On the Difficulty of Scalably Detecting Network Attacks

Kirill Levchenko
UC San Diego
klevchen@cs.ucsd.edu

Ramamohan Paturi
UC San Diego
paturi@cs.ucsd.edu

George Varghese
UC San Diego
varghese@cs.ucsd.edu

ABSTRACT

Most network intrusion tools (e.g., Bro) use per-flow state to reassemble TCP connections and fragments in order to detect network attacks (e.g., SYN Flooding or Connection Hijacking) and preliminary reconnaissance (e.g., Port Scans). On the other hand, if network intrusion detection is to be implemented at high speeds at network vantage points, some form of aggregation is necessary. While many security analysts believe that such per-flow state is required for many of these problems, there is no clear proof that this is the case. In fact, a number of problems (such as detecting large traffic footprints or counting identifiers) have scalable solutions. In this paper, we initiate the study of identifying when and how a security attack detection problem can have a scalable solution. We use tools from Communication Complexity to prove that the common formulations of many well-known intrusion detection problems (detecting SYN Flooding, Port Scans, Connection Hijacking, and content matching across fragments) *require* per-flow state. Our theory exposes assumptions that need to be changed to provide scalable solutions to these problems; we conclude with some systems techniques to circumvent these lower bounds.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General—*security and protection*; F.2 [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

General Terms

Security, Theory

Keywords

Network Intrusion Detection, Communication Complexity

1. INTRODUCTION

As the story is commonly told, the Internet began as a collection of mutually-trusting networks. While the trust implicit in the original design remained after the phenomenal success of the Internet, the trustworthiness of the hosts did not. Thus, today we are faced with the problem of extracting utility from an Internet seemingly populated by hostile agents.

The first approach to this problem is to secure Internet hosts using anti-virus software and personal firewalls. Since it is often hard to ensure that all hosts are running the latest versions of such software and many attempts are from insiders, a second approach has been to secure campus and enterprise networks against attacks using firewalls and Network Intrusion Detection Systems (NIDS).

Intrusion Detection Devices. While firewalls have their place, they only check for allowed header patterns; many catastrophic attacks such as worms have tunneled through firewalls using the headers of legitimate traffic (e.g., HTTP) that must be allowed in. Thus, despite a number of technical issues, almost every major enterprise buys NIDS devices, and the market is growing: Table 1 offers a sampling of vendors offering network security products.

Taking this approach a step further, perhaps intrusion detection devices could be deployed deeper inside the network, say, at the ISP level, to provide security for all hosts connecting through the service provider.[1] This vantage point provides cost savings compared to deployment closer to the end-hosts, due both to the smaller number of devices that need to be deployed and to the reduced administrative overhead required to manage these systems. A second well-known reason to deploy a NIDS deeper within the Internet is to provide more discriminating blocking when source addresses are being spoofed. For example, in response to a TCP SYN Flood attack, a NIDS may rate-limit traffic to the targeted. However, while this mitigates the effects of the attack, it also disrupts service for legitimate users: by placing the NIDS deeper inside the network, and as close to the attacker as possible, fewer legitimate users are affected.

Per-Flow State and Its Effect on Speed. Current intrusion detection and prevention systems from vendors such as NetScreen [21], Cisco [5], and Checkpoint [4] seek to detect a wide class of network intrusions (e.g., denial-of-service attacks, worms, port scans) within enterprise or campus networks. Two big challenges for the next generation of IDS devices are (**i**) to provide real-time or wire-speed intrusion

[1]Of the NIDS vendors we surveyed, only one—Fortinet [10]—actually markets its NIDS to service providers.

	Detection Claim			
Vendor	*SYN Flooding*	*Port Scans*	*Conn. Hijacking*	*Content Matching*
Checkpoint [4]	●	●	●	●
Cisco [5]	●	●	●	●
ForeScount [9]		●		●
Fortinet [10]				●
Juniper [14]	●	●		
Mazu Networks [18]	●		●	●
NetScreen [21]		●		●
Network Associates [20]	●	●		●
TippingPoint [26]				●

Table 1: Vendors offering network intrusion detection systems, or components thereof. For each vendor, we indicate which of the four attacks considered in this paper their products claim to detect. A blank entry indicates that we could not find a specific claim to detect the attack in the literature provided on the vendor's web site, though the product may detect the attack.

detection so as not to miss attacks at high speeds and (**ii**) to reduce false positives.

Running a NIDS device at the edge of a network or even deeper inside the network requires that the NIDS operate at higher link speeds. Such vantage points also expose the detection system to a larger number of flows, and in this setting, it becomes infeasible to maintain per-flow or per-host state, that is, to track each active "conversation" taking place across the instrumented link. But this is precisely the information used by most NIDS to detect large classes of attacks. For example to detect Port Scans, Snort 2.0 [25], a popular open source NIDS, maintains a per-target 65536-bit vector to track the ports probed by a suspected attacker, and of the vendors in Table 1, several (e.g., Cisco [5] and Network Associates [20]) mention maintaining per-flow state in their product literature.

Now, it is well known that high speed implementations of other tasks in the forwarding path of network devices (such as routers) have relied on the use of small memory footprints to fit into cache or on-chip SRAM. For example, Internet lookups in routers use prefix aggregation to store around 150,000 prefixes for the entire Internet. Similarly, DiffServ uses class aggregation to avoid per-flow state in core routers. This is fundamentally because the number of connections/flows at network vantage points can easily scale into the millions, and this does not scale with the increases in the size of high speed memory. Thus, in the wire-speed implementation community the notion of per-flow state has been anathema for a long time.

While the requirement for small amounts of control memory may seem like an irrational prejudice in a world of plentiful memory, it does have a technical basis. Roughly, this is because while fairly fast DRAM memories (e.g., DDR) are available, the highest speed memories (on-chip and off-chip SRAM) are still limited and growing slowly. In particular, programmability is often required for security implementations because attack technology is constantly evolving, and most programmable chips such as FPGAs have even smaller amounts of memory (say 1 Mbit) compared to custom chips. In particular, these chips cannot hold the state for the millions (or even hundreds of thousands) of concurrent connections that traverse a fairly large enterprise network.

The current state-of-the-art is to use load-splitters to allow several slower NIDS devices to protect a single high-speed link. However such installations are expensive and inhibit the detection of attacks split between several devices.

Is Per-Flow State Necessary? Given these constraints, we may ask whether per-flow state is truly necessary for common network intrusion detection tasks, or whether there is hope for finding clever, memory-efficient algorithms. Consider, for example, the following four detection problems:

1. Estimate the number of distinct destinations to within 1%.
2. Determine if any WWW site was accessed more than three times.
3. Estimate the number of DNS queries to within 1%.
4. Detect a TCP SYN Flooding attack.

Of the four, which require per-flow or per-host state in a NIDS? It turns out that the first problem has an efficient, but non-trivial solution [1, 8] which does not require per-host state. On the other hand, the second problem *does* require state proportional to the number of WWW sites, which can be shown using the technique in Section 4. The third problem, estimating the number of DNS queries is quite easy and can be solved exactly: we need only count the number of UDP packets to port 53. Finally, detecting a TCP SYN Flooding attack requires per-flow state under some conditions and does not under others; we consider it in Section 3.

In this paper, we formalize the study of which attack detection tasks require per-flow or per-destination state. Like all "impossibility" results, they are best regarded by practitioners as labor-saving devices: by precisely stating the problems and assumptions under which per-flow state is required, a designer can either choose to use memory technology that allows the appropriate amount of memory with corresponding speed tradeoffs, or *change the assumptions* to invalidate the lower bounds and hence make low-memory implementations possible.

Toward this end, we consider four common network intrusion detection tasks we presented in Table 1: detecting TCP SYN Flooding, detecting Port Scans, detecting attempts to hijack a TCP connection, and matching a payload split across two fragments (evasion attack). For each, we prove a

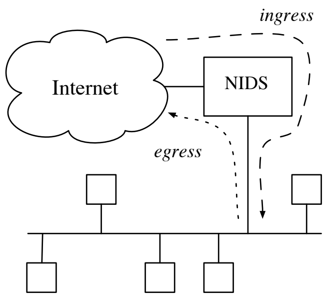

Figure 1: Our setting is a Network Intrusion Detection System (NIDS) on a router connecting a network to the Internet. We say a detector is an *ingress* detector if it only uses traffic entering the protected network. A detector is an *egress* detector if it also uses traffic leaving the protected network.

lower bound on detection and then comment on its practical implications.

We continue with some definitions in Section 2 which we will need for formalizing our arguments, and a description of the Set Disjointness problem of Communication Complexity, a powerful tool we will use to establish our lower bounds. The following sections will consider in turn each of the four NIDS tasks listed above. We will begin each section with a description of the problem, followed by the lower bound. Thus, Section 3 covers TCP SYN Flooding, Section 4 covers Port Scans, Section 5 covers TCP Connection Hijacking, and Section 6 covers Evasion by fragmentation.

Our results and their implications are summarized in Table 2, on the next page.

2. THEORETICAL TOOLS

In this section, we introduce the main theoretical tools we use to prove lower bounds on attack detection. In Section 2.1 we describe our setting and introduce a simple Turing Machine model. The reader who is used to implementations using today's computers and chip sets should not be alarmed: space lower bounds in the Turing Machine model apply to RAM based and other more commonly used computation models. Next, in Section 2.2 we intuitively describe our main tool for lower bounds: the classical Set Disjointness problem from Communication Complexity. In the rest of the paper, the game will be to recast seemingly different attacks as versions of the Set Disjointness problem.

2.1 Setting and Model

Our setting is a detection system monitoring traffic on a network router connecting a network to the Internet (see Fig. 1). We classify detection schemes as *ingress* or *egress* based on whether the detector uses only traffic entering the protected network (ingress) or traffic entering *and leaving* the network (egress). We make this distinction because a NIDS may only have access to incoming traffic due to asymmetric routing, and because this distinction becomes crucial for some problems.

We model the detection system as a Turing Machine with a work tape and a sequentially accessed read-only input tape. The input consists of a *packet sequence*, a finite sequence over some problem-specific alphabet P. We take the space complexity of an algorithm to be the maximum space (in bits) used over all finite-length packet sequences.[2] We allow the Machine to make probabilistic choices (e.g., to take random samples), and require only that it fail with some fixed probability less than a half.

Finally, as a notational convenience, denote the set $\{1, \ldots, n\}$ by $[n]$.

2.2 Set Disjointness as a Tool

The Set Disjointness problem is a fundamental problem in Communication Complexity that addresses the difficulty (in terms of bits exchanged) of knowing whether two sets at two ends of a link have an element in common. This construction was used by Alon, Matias, and Szegedy [1] to prove a lower bound on the *space* required by any device that wishes to determine the number of occurrences of the most frequent element (e.g., the element could be the content of a packet) in a data stream.

The earlier result of Alon *et al.* used Set Disjointness to prove that certain database streaming queries require large amounts of memory. By contrast, in this paper we will use the Set Disjointness problem to prove that many formulations of attack detection problems also require large amounts of memory in the worst case.

More precisely, the Set Disjointness problem is as follows. Consider two parties, Alice and Bob, each with a set of numbers between 1 and n. Alice does not know Bob's set of numbers, and Bob does not know Alice's. They would like to find out if there is some number that they both have by communicating as few bits of information as possible. We allow them unlimited computational resources and measure only the number of bits they send to each other. One strategy might be for Alice to just say "yes" or "no" n times, once for every number starting at 1. Then Bob will know whether they have any number in common, and tell Alice, "yes" or "no." However this requires $n + 1$ bits to be communicated, and we may well ask if we can do better. It turns out the answer is "no," and we will use this fact as a basis for proving asymptotic lower bounds on the amount of space required to detect attacks.

3. TCP SYN FLOODING

We now begin our study of attack scalability with TCP SYN Flooding attacks.

A TCP SYN Flood [2] is a denial-of-service attack that exploits the way a TCP connection [23] is established between a sender U and a receiver V. TCP's mating dance begins with U sending a so-called `SYN` packet, and V responding with a so-called `SYN+ACK` response. Unfortunately, after sending the `SYN+ACK` response, V allocates resources to remember the pending connection for a pre-specified amount of time, roughly a minute, waiting for sender U to establish the connection with an `ACK` packet.

A TCP SYN Flood occurs when a malicious host repeatedly sends `SYN` packets, typically with forged source ad-

[2]Another reasonable model might be to measure space in words, where the word size is logarithmic in the length of the input.

Detection Problem	Scalable	Comment
SYN Flooding (§3) ingress	No	Egress detection relies on
— egress	Yes	trust of protected network.
Port Scans (§4) ingress	No	Scalable egress detection is possible
— egress	Yes*	by estimating the no. of distinct items.
Conn. Hijacking (§5) ingress	No	No scalable detection possible if attacker
— egress	No	is outside the protected network.
Evasion (§6) ingress	No	Workaround exists for IP fragmentation,
— egress	No	but not for TCP segmentation.

Table 2: A summary of our results and their practical implications. "Ingress" refers to detection using only traffic entering the network, "egress" to detection using traffic entering *and leaving* the network. "Scalable" means that there is a detection scheme that does not use per-flow state. See the appropriate section for discussion. *Additional empirical evidence is needed to test the effectiveness of this approach.

dresses, causing the listening host to allocate many half-open connections. Because of the generous timeout and the small number of connections allowed to be in the half-open state, it is easy for an attacker to deplete a server's connection resources, preventing service to legitimate clients. Several modern implementations address or mitigate this problem by using techniques such as SYN-cookies or by simply allocating more resources. Nevertheless, this attack is still used and is useful to detect.

At the link level, this attack may be characterized as a packet sequence containing a SYN packet from U to V without a subsequent ACK packet. However instead of considering this attack, we consider a slightly different problem: detecting unclosed connections. That is, connections consisting of a an opening SYN packet without a closing FIN packet. Detecting unclosed connections (versus half-open connections) requires us to also consider RST packets, as these cause the connection to close as well. However since this only makes the problem *harder*, it does not affect our lower bound.

3.1 Ingress SYN Flood Detection

In this section, we consider the problem of detecting unclosed connections using only traffic entering the protected network. At first glance, it may seem sufficient to count the number of SYN packets and the number of FIN packets, declaring the packet stream to contain a unclosed session if there are more of the former that the latter. Indeed, this is the approach advocated by [27]. However an attacker can circumvent this strategy by sending FIN packets ahead of the SYN packets, or sending FIN packets with a different address. Without matching FIN packets to preceding SYN packets, there is no efficient way to determine if there are unclosed sessions, as we show next.

Abstract Problem Formulation. To apply our lower bound technique to this problem, define the packet set P to be $[m] \times \{\texttt{SYN}, \texttt{FIN}\}$: the set of tuples consisting of a session identifier and a packet type field, either SYN or FIN. Call a packet $(x, \texttt{SYN})$ in a packet sequence *matched* if $(x, \texttt{FIN})$ occurs in the remainder of the sequence. Call a packet *unmatched* if it is not matched. Intuitively, unmatched SYN packets correspond to unclosed connections.

Example: To model TCP SYN Flooding, the session identifier x in the abstract formulation would be the TCP 4-tuple consisting of the source address, source port, destination address, and destination port.

Let SYNMATCH be the problem of detecting a packet sequence containing one or more unmatched SYN packets. In practice, detecting one unmatched SYN packet in the whole stream may be too strict. Let us allow the algorithm to fail if there are some—but not too many—unmatched SYN packets. Formally, we guarantee (to any algorithm for detecting unmatched SYN packets) that the input contains either no unmatched SYN packets or that the unmatched SYN packets constitute at least an α fraction of the whole packet sequence. Call this variant α-SYNMATCH. We are now ready to prove a lower bound on ingress detection.

Lower Bound. Any algorithm for SYNMATCH must use $\Omega(m)$ space. Any algorithm for α-SYNMATCH where $\alpha m \geq 1$ must use $\Omega(1/\alpha)$ space.

Proof of Lower Bound. We will prove the first statement; see the Appendix for the proof of the second. As promised, the proof is by reduction from DISJ, the Set Disjointness problem of Communication Complexity; see the Appendix for a formal description of this problem and its lower bounds. We will show that two parties, Alice and Bob, can decide if two sets, $X \subseteq [n]$ and $Y \subseteq [n]$, held by Alice and Bob respectively, are disjoint using only S bits of communication, where S is the space used by the SYNMATCH detection algorithm. The lower bound on the communication complexity of Set Disjointness will imply $S = \Omega(m)$.

The reduction work as follows. Alice forms the packet sequence

$$(x_1, \texttt{SYN}), (x_2, \texttt{SYN}), \ldots, (x_{|X|}, \texttt{SYN}),$$

where $x_1, x_2, \ldots, x_{|X|}$ are the elements of X. She runs the SYNMATCH detection algorithm (with parameter m set to n) on this sequence, and suspends it immediately after reading the last element. She then sends its state to Bob. Bob forms the packet sequence

$$(\bar{y}_1, \texttt{FIN}), (\bar{y}_2, \texttt{FIN}), \ldots, (\bar{y}_{|\bar{Y}|}, \texttt{FIN}),$$

where $\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_{|\bar{Y}|}$ are the elements of $\bar{Y} = [n] \setminus Y$. He then resumes the algorithm using the state received from Alice, providing the remainder of his sequence as the rest of the input. To the algorithm, the input appears as a concatenation of their two sequences.

We observe that if X and Y are disjoint, then for all $x \in X$ there will be a matching $x \in \bar{Y}$, so there will be a closed connection consisting of $(x, \texttt{SYN})$ and $(x, \texttt{FIN})$ in the aggregate sequence seen by the algorithm. Conversely, if X and Y intersect, say at some element c, then there will be a packet $(c, \texttt{SYN})$ without a matching $(c, \texttt{FIN})$ packet. Thus, X and Y intersect if and only if the aggregate packet sequence seen by the algorithm contains an unmatched `SYN`. Using the result of the algorithm, Bob can determine if X and Y are disjoint.

Note that the only communication between Alice and Bob are the S bits of the state of the algorithm. Since the communication complexity is $\Omega(n)$ and $m = n$, it follows that $S = \Omega(m)$. $\square$

We have shown that any detection system must effectively maintain per-flow state in order to detect an unmatched `SYN` packet.

Practical Implications. The lower bound means that we can detect a TCP SYN Flood using traffic entering the network only if it is a sizable fraction of all the (`SYN` and `FIN`) traffic. Effectively, the best we can do is to pick random `SYN` packets in the packet stream and watch if they are followed by a matching `ACK`; see [7] for this type of approach. Note that even this scheme is vulnerable to evasion as described by Paxson in [22]: an attacker can set the IP TTL field of the `ACK` so that the packet is seen by the NIDS, but not by the victim.

In practice, a detection scheme may keep per-flow state, but only within a reasonable round-trip time (RTT), say 1 second. That is, if an incoming `SYN` is not followed by a matching `ACK` within a second, we may reasonably assume the `ACK` will never come. However a simple back-of-the-envelope calculation shows that even this relaxation requires considerable memory, because a 1 Gbit/sec link may see up to 3 million `SYN` packets per second.

3.2 Egress SYN Flood Detection

If the attacker cannot inject packets into the traffic leaving the protected network, we can detect a TCP SYN Flood by considering the difference between the number of `SYN` packets entering the network and the number of `FIN` (or `SYN+ACK` packets leaving the network [16, 27]. Since the attacker cannot tamper directly with this packet sequence, we can expect a large number of incoming `SYN` packets without corresponding `FIN` packets to be a reasonable indicator of an attack. Implicit in this detection scheme, however, is trust of hosts with access to this packet stream. A cooperating host on the inside (e.g., a zombie) could spoil this scheme by sending spoofed `FIN` packets through the NIDS. The NIDS would be deceived into thinking the `SYN` packets resulted in successful connections (see Fig. 2). Note, however, that this scenario is somewhat unusual, as a compromised host on the inside could attack the victim directly.

The above scenario illustrates an important point about egress detection, and that is that *an egress detection system relies on the credibility the packet stream leaving the network*; it is up to the administrator and the users of the scalable NIDS to decide whether this trust is justified.

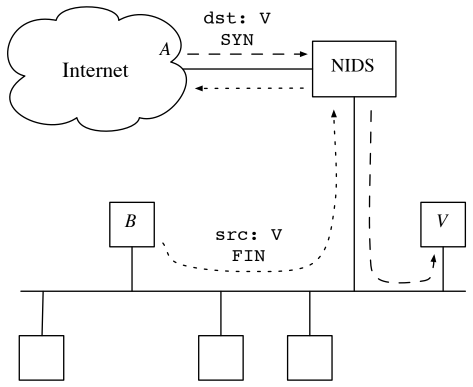

Figure 2: A SYN Flooding scenario in which the NIDS is fooled by an attacker A with control over a compromised host B on the inside. The NIDS considers a TCP SYN Flood to be in progress if there are significantly more `SYN` packets entering the network than `FIN` packets leaving the network. The attacker sends `SYN` packets to V, and at the same time, a compromised host B inside the network sends spoofed `FIN` packets through the NIDS, fooling it into ignoring the attack.

4. PORT SCANS

We continue our study of the state required for detecting attacks by considering Port Scans. Though not an attack in itself, a Port Scan is usually a precursor to one. It consists of an attacker identifying the open ports on a machine to determine which services are available. (With this knowledge, the attacker can attempt to exploit a vulnerability in one of these services) To determine if a port is open, an attacker sends a packet to the target host attempting to connect to the desired port. If the target host is listening on that port, it will respond by opening a connection. The exact details of how a host responds to a connection attempt will be examined in Section 4.2, when we consider egress detection. Let us start by seeing whether we can detect this phenomenon scalably based only on traffic entering the network.

4.1 Ingress Port Scan Detection

Abstract Problem Formulation. For this problem, define $P = [m] \times [m]$ to be the set of tuples consisting of a source address and destination address. Call a set $\{(x, i_1), (x, i_2), \ldots, (x, i_k)\}$ a scan of size k, if the $i_1, i_2, \ldots, i_k \in [m]$ are distinct and $x \in [m]$. Let k-SCAN be the problem of determining if a packet sequence contains a scan of size k. Note that this corresponds to the scenario where a scanner probes a set of hosts (typically on the same port, and sometimes called a *horizontal scan*).

Sometimes a Port Scan is taken to be a scan probing a range or ports/services on a single machine, which could be looked on as a *vertical scan*. Our formal definition captures both these scenarios as well as more complex scans that can scan a number of hosts and a number of ports. The key abstraction is that one one host is sending the packets. After proving a lower bound for this variant, we will consider distributed Port Scans.

Example: To model a general scan in the abstract formulation, x could represent the IP source address of the scanner, and i_1, i_2 etc., could represent the combination of the IP address and port that is scanned at each point in the sequence.

Lower Bound. Any algorithm for k-SCAN must use $\Omega(m/k^5)$ space.

Proof of Lower Bound. We will give the proof for the the case $k = 2$. The proof of the general case uses a different reduction; it is given in the Appendix.

As before, this proof is by reduction from DISJ, the Set Disjointness problem of Communication Complexity, to 2-SCAN. Let Alice have a set $X \subseteq [n]$ and Bob have a set $Y \subseteq [n]$. Alice forms the packet sequence

$$(x_1, 1), (x_2, 1), \ldots, (x_{|X|}, 1),$$

where $x_1, x_2, \ldots, x_{|X|}$ are the elements of X. She runs the 2-SCAN detection algorithm (with parameter m set to n) on this sequence and suspends it when it reaches the last element, but before it terminates. She then sends the state of the algorithm to Bob, who resumes the algorithm providing the following sequence as the remainder of the input:

$$(y_1, 2), (y_2, 2), \ldots, (y_{|Y|}, 2),$$

where $y_1, y_2, \ldots, y_{|Y|}$ are the elements of Y.

If X and Y are disjoint, then the input sequence to the algorithm will not contain a scan, and conversely, if X and Y intersect, say at an element c, then the input sequence will contain a scan consisting of $(c, 1)$ and $(c, 2)$. Thus, Bob can determine if X and Y are disjoint using the output of the 2-SCAN detection algorithm. Since Alice sent Bob S bits (the size of the state of the algorithm), it follows that $S = \Omega(m)$. $\square$

Practical Implications. The lower bound implies that ingress detection of a Port Scan for any constant k does indeed require per-flow state. The reduction hinges on the difficulty of determining whether, among many hosts each probing exactly one target, there is one who probes two or more (k or more in the general case).

Nevertheless, an increase in the number of distinct targets probed (not necessarily from the *same* host) may in some cases be a reasonable indicator of suspicious activity. In fact, this is exactly how a distributed Port Scan would appear to the NIDS. It turns out that this quantity can be estimated efficiently; see, for example [8], who also propose using it as an indicator of a Port Scan. However this approach has several problems. First, the estimator is not accurate enough to detect a scan hidden in a large traffic stream, as would be found inside the network. Second, an increase in the number of distinct hosts accessed may have a benign explanation (a route change is one example). It is doubtful, therefore, that an increase in the estimate of the number of distinct destinations is a viable indicator of a Port Scan.

4.2 Egress Port Scan Detection

The TCP specification [23] requires a host to send a `RST` packet in response to a connection attempt on a port without a listening process. Similarly, a packet sent to a UDP port may generate an ICMP "port unreachable" response [24].[3]

[3]Unfortunately, some firewalls might block such packets for security reasons.

In view of this, an increase in the number of distinct destination addresses generating such responses may be a better indicator of a Port Scan, as it implicitly excludes packets addressed to existing services. However "noise" (e.g., due to mistyped addresses) or poor detector sensitivity could still hinder detection; therefore, it remains to be verified experimentally whether this is an accurate detector.

Note, however, that unlike TCP SYN Flooding, an insider cannot fool the NIDS, because he cannot prevent the victim's `RST` packets from reaching the NIDS.

5. TCP CONNECTION HIJACKING

The Transmission Control Protocol provides the abstraction of a reliable communication channel for two processes on (usually district) network hosts, say U and V. One of the mechanisms used by TCP to ensure this is the segment sequence number, which identifies where a packet payload belongs within the interprocess data stream. The same mechanism also provides some protection against an attacker wishing to inject packets into an existing connection, because the sequence number must be within the window accepted by V.

An attacker A wishing to do so must be able to correctly number the forged packets to V. Without knowing any of the sequence numbers used by U, the attacker—after narrowing down the possibilities—may attempt to guess the correct sequence number, injecting packets with incorrect sequence numbers into the stream. To an observer seeing traffic to V, this will appear as a TCP session containing packets whose sequence numbers are not strictly increasing. Can we efficiently detect this attack signature at the NIDS?

5.1 Ingress Connection Hijacking Detection

Abstract Problem Formulation. In this section, we consider the problem of detecting a session with sequence numbers that are not increasing. Formally, let $P = [m] \times [\ell]$ be the set of tuples consisting of a session identifier and a sequence number. Define a *session* to be the subsequence of packets with the same session identifier. Call a packet *out-of-order* if it is not the first packet in the session and if the sequence number of the preceding packet in the session is greater than its own. Note, however, that packet reordering and retransmission occurs in the normal operation of the network. To exclude this benign phenomenon, it would be more correct to consider an out-of-order packet as anomalous if its segment sequence number differs substantially from that of its session predecessor. Let k-SEQMATCH be the problem of determining if the packet stream contains a session with k or more out-of-order packets, or whether there are no out-of-order packets in any session. That is, we guarantee (to any detection algorithm) that if a session contains any out-of-order packets, it contains at least k such.

Example: To model TCP Connection Hijacking, in the abstract formulation the session identifier would represent the TCP 4-tuple connection identifier, and the sequence number would represent the TCP sequence numbers in a received packet.

Given this formulation, we have:

Lower Bound. Any algorithm for k-SEQMATCH must use $\Omega(m/k)$ space.

Proof of Lower Bound. See the Appendix.

Practical Implications. The lower bound tells us that detecting attempts to guess sequence numbers based solely on the stream of sequence numbers entering the network requires per-flow state. In fact, any scheme that relies on somehow relating two packets in a session will require per-flow state. For example, suppose we could tell if two packets came from the same host, even if the source addresses were forged (for example, by means of OS fingerprinting [11]). Could we efficiently detect a Connection Hijacking then? No, because it requires comparing two packets in the same session, and therefore requires per-flow state.

5.2 Egress Connection Hijacking Detection

The TCP specification [23] defines the appropriate response to a segment sequence number outside the receiver's window to be an acknowledgment of the last valid packet payload. Although responses from V provide no indication (to the NIDS) of how far the sequence number fell outside its window, an excessive number of `ACK` packets may be an indicator to U—the real sender—that an attack is taking place. That is, if U notices that that the number of `ACK` packets from V is significantly greater than the number of packets sent by U, then U has reason to suspect that someone is injecting packets into the session.

6. EVASION BY FRAGMENTATION

A number of intrusion detection systems attempt to detect an attack by attempting to find a substring (e.g., of a known worm payload) in a packet. Fragmentation, the mechanism by which the Internet Protocol splits a large packet into several smaller ones, allows the attacker to conceal the payload in several fragments; an analogous mechanism, called segmentation, is used by TCP to divide the interprocess data stream into discrete packets. With the substring sought by the intrusion detection system spread across several packets in the packet sequence, space-efficient detection becomes impossible. In this section, we prove a lower bound on determining if a packet split into two fragments contains some string.

6.1 Evasion Detection

Abstract Problem Formulation. Define $P = [m] \times \{\mathtt{1ST}, \mathtt{2ND}\} \times \{\mathtt{Q}, \mathtt{R}\}$ to be the set of tuples containing a packet identifier, a marker indicating whether the fragment is the first or second fragment in a two-fragment packet, and one of two parts of a string for which the intruder hopes to avoid evasion. We say that a packet stream contains the string `QR` if there are two packets $(c, \mathtt{1ST}, \mathtt{Q})$ and $(c, \mathtt{2ND}, \mathtt{R})$, occurring in this order, for some $c \in [m]$. Call the problem of detecting such a packet stream EVASION.

Example: To model evasion via IP fragmentation, the packet identifier in the abstract formulation would represent the combination of the packet identifier and source IP address. For TCP segmentation, the packet identifier would represent the TCP connection 4-tuple. For IP fragmentation, `1ST` would represent a fragment with offset 0, while `2ND` would a represent a fragment with the "more fragments" bit being zero. For TCP segmentation, the first and second markers represents the byte sequence numbers and the payload lengths, from which these markers can be inferred. The string `QR` can be any malicious payload the NIDS is trying to detect, such as a known virus signature.

Lower Bound. Any algorithm for EVASION must use $\Omega(m)$ space.

Proof of Lower Bound. See the Appendix.

Practical Implications. The lower bound means that any algorithm attempting to detect a particular string in the reconstructed interprocess data stream must maintain per-flow state. Effectively, any NIDS must have traffic normalizer component, as described in [12], to reassemble fragmented packets.

Fortunately, there is a workaround for IP fragmentation, implemented in some NIDS, and that is to simply drop very small fragments, as there are so few legitimate reasons to see them on the Internet. The NIDS can now detect if a fragment contains a sufficiently large substring of a harmful payload. TCP segments, however, can be quite small, allowing the attacker to spread the payload across several packets far apart in the packet stream, making detection without per-flow state impossible.

7. CONCLUSION

Table 2 summarizes our results on detecting TCP SYN Flooding, Port Scans, TCP Connection Hijacking, and Evasion by fragmentation, as well as their implications for the intrusion system designer. Our results indicate that common stream processing problems are created equal, and therefore the coarse characterization that they all require per-flow state is not accurate.

For Port Scans and TCP SYN Flooding, the practical implication of our results is that while scalable ingress detection (as stated) is impossible, egress detection is possible under certain, reasonable assumptions. For example, for Port Scans, we must assume that failed scans produce a measurable response (e.g., `RST` packets) that are not produced normally, so that the signal-to-noise ratio is sufficient for detection. In fact, a general form of the latter approach is to recognize an attacker as someone trying to access nonexistent hosts or services by using a network telescope [19] or a honey pot [6]; at least one vendor—NetScreen [21]—has already incorporated the latter into their product. More than any other, the Port Scan problem illustrates that a phenomenon may have several manifestations, and while scalably detecting one might be hard, scalably detecting another may be feasible.

The results for detecting Evasion attacks and TCP Hijacking are more grim (no solution without keeping state for either ingress or egress detection), confirming the intuition of IDS designers. Despite this, we emphasize that these results are *under worst case conditions!* In practice, the packet stream may be disposed much more favorably toward the NIDS designer. However the fact that worst case conditions *can* arise, means that the designer must decide how the NIDS should fail when they *do* arise.

Furthermore, there may be some information available to the NIDS not captured by our model. For example, Jin, Wang, and Shin [13] use the TTL field to determine if the source address hash been spoofed: a sure sign of illicit activity.

We began by observing that economic and logistical forces may drive the network intrusion system deeper and deeper into the network. The technical implications of this new setting place an increased demand on the system to perform quickly and efficiently, demands that current NIDS state of the art would be hard-pressed to meet. Clearly, NIDS designs must change. As a step in this direction, we have shown the hardness of detecting a number of common attack classes, bringing to light the fundamental obstacles that must be overcome. More than a vindication of the use of per-flow state in existing NIDS, we hope that our results guide the NIDS algorithm designer in fruitful directions, some of which we have tried to point out throughout our analysis. We believe that a new breed of fast, scalable IDS systems with low false positive rates will be born of a new fellowship between algorithmic, network, and security research.

8. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. "The Space Complexity of Approximating the Frequency Moments." *STOC* 1996, pp. 20–29.

[2] CERT. "CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks." 1996.

[3] CERT. "CERT Advisory CA-1997-28 IP Denial-of-Service Attacks." 1997.

[4] Check Point Software Technologies, Ltd. `http://www.checkpoint.com/`

[5] Cisco Systems. `http://www.cisco.com/`

[6] F. Cohen. "A Mathematical Structure of Simple Defense Network Deceptions." *Computers & Security* 19 (2000), pp. 520–528.

[7] C. Estan, G. Varghese. "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice." *ACM Transactions on Computer Systems* 21 (3), pp. 270–313.

[8] C. Estan, G. Varghese, M. Fisk. "Bitmap Algorithms for Counting Active Flows on High Speed Links." *Internet Measurement Conference*, 2003.

[9] ForeScout Technologies. `http://www.forescout.com/`

[10] Fortinet, Inc.. `http://www.fortinet.com/`

[11] Fyodor. "Remote OS detection via TCP/IP Stack FingerPrinting." `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`

[12] M. Handley, V. Paxson. "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics." *10th USENIX Security Symposium*, pp. 115–131.

[13] C. Jin, H. Wang, and K. Shin. "Hop-Count Filtering: An Effective Defense Against Spoofed DDoS Traffic." *ACM Conference on Computer and Communication Security (CCS).* October 2003.

[14] Juniper Networks. `http://www.juniper.net/`

[15] B. Kalyanasundaram and G. Schnitger. "The Probabilistic Communication Complexity of Set Intersection." *SIAM Journal on Discrete Mathematics* 5 (4), pp. 545–557, 1992.

[16] R. Kompella, S. Singh, G. Varghese. "On Scalable Attack Detection in the Network." *ACM SIGCOMM Internet Measurement Conference*, 2004.

[17] E. Kushilevitz and N. Nisan. *Communication Complexity*, Cambridge University Press, 1997.

[18] Mazu Networks. `http://www.mazunetworks.com/`

[19] D. Moore, G. Voelker, and S. Savage. "Inferring Internet Denial-of-Service Activity." *10th USENIX Security Symposium*, pp. 9–22.

[20] Network Associates, Inc. `http://www.nai.com/`

[21] NetScreen Technologies, Inc. `http://www.netscreen.com/`

[22] V. Paxson. "Bro: A System for Detecting Network Intruders in Real-Time." *7th USENIX Security Symposium*, pp. 31–52.

[23] J. Postel. "Transmission Control Protocol." *RFC 793.*

[24] J. Postel. "Internet Control Message Protocol." *RFC 792.*

[25] Snort. `http://www.snort.org/`

[26] TippingPoint Technologies. `http://www.tippingpoint.com/`

[27] H. Wang, D. Zhang, and K. Shin. "Detecting SYN Flooding Attacks." *IEEE INFOCOM*, 2002.

APPENDIX

Appendix

The Set Disjointness Problem

The Communication Complexity (see [17]) Set Disjointness problem, DISJ, goes as follows. Two parties with unlimited computational resources, canonically, Alice and Bob, each have a set $X \subseteq [n]$ and $Y \subseteq [n]$, respectively. They would like to determine whether and X and Y are disjoint or intersect, while exchanging as few bits as possible.

Trivially, Alice can send Bob n bits, where the i^{th} bit is 1 if $i \in X$ and 0 otherwise. Bob can then determine whether $X \cap Y = \emptyset$, and communicate this to Alice. We may ask if they can do better by communicating fewer than $\Theta(n)$ bits. It turns out the answer is "no," even if we allow Alice and Bob a randomized protocol that errs with a fixed probability less than a half. See [17] for the deterministic lower bound, and [15] for the probabilistic one.

The fact that this problem remains hard even under randomization makes it a powerful source of reductions. It was first used in the stream setting by Alon, Matias, and Szegedy [1], whose technique we use here.

The Multi-Party Set Disjointness Problem of Alon *et al.*

For the proof of the SCAN lower bound, we rely on another problem in Communication Complexity. In [1], Alon, Matias, and Szegedy introduce a multi-party Set Disjointness problem, which we call MDIS, involving σ players, each holding a subset of $[n]$ of cardinality τ. The subsets $X_1, X_2, \ldots, X_\sigma$ are either pair-wise disjoint, or intersect at exactly one element, i.e., for any $i \neq j$,

$$X_i \cap X_j = \bigcap_{r=1}^{\sigma} X_r = \{c\}.$$

Alon, Matias, and Szegedy showed that for a fixed failure probability less than a half, and $\tau \geq \sigma^4$, the communication complexity of this problem is $\Omega(\tau/\sigma^3)$.

Omitted Proofs

Lower Bound (see §3). Any algorithm for SYNMATCH must use $\Omega(m)$ space. Any algorithm for α-SYNMATCH where $\alpha m \geq 1$ must use $\Omega(1/\alpha)$ space.

PROOF. The first half of the claim was proven in Section 3.1; it remains to prove the lower bound for the relaxed variant. In fact, the proof is the same, with the exception that Alice and Bob must use the correct instance of α-SYNMATCH.

Let Alice and Bob have sets $X \subseteq [n]$ and $Y \subseteq [n]$, respectively. Set α to $1/2n$ and set m arbitrarily to satisfy $\alpha m \geq 1$. Following the protocol given in the proof of the first claim, we note that each element of the packet sequence seen by the algorithm is at least an α fraction of the total input size. This allows Alice and Bob to use their instance of the α-SYNMATCH detection algorithm as a drop-in replacement for the SYNMATCH detection algorithm used in the proof of the first claim. $\square$

Lower Bound (see §4). Any algorithm for k-SCAN must use $\Omega(m/k^5)$ space.

PROOF. We reduce MDIS, the multi-party Set Disjointness problem of Alon, Matias, and Szegedy (see appendix 7) to k-SCAN. Set $\sigma = k$, and let τ be such that $n = (2\tau - 1)\sigma + 1$ as required. We will demonstrate a protocol for MDIS using an algorithm for deciding k-SCAN.

The i^{th} player having input set X_i, forms the following input sequence to the k-SCAN algorithm (with parameter $m = n$):

$$(x_{i1}, i), (x_{i2}, i), \ldots, (x_{i\tau}, i),$$

where $x_{i1}, x_{i2}, \ldots, x_{i\tau}$ are the elements of X_i. The first player provides his sequence to the algorithm, suspends it after reading the last tuple, and sends the S bits of state to the second player. The second player resumes the algorithm providing his sequence as the continuation of the input, suspends it before reading the last tuple, and sends the state to the next. This continues until the last player receives the state and finishes the computation by providing his sequence as the remainder of the input.

If the sets $X_1, X_2, \ldots, X_\sigma$ intersect at one element, say c, then the scan $(c, 1), (c, 2), \ldots, (c, k)$ will occur in the packet sequence. Conversely, if the sets are pair-wise disjoint, no element $c \in [n]$ will occur more than once in the packet sequence. This allows the last player to determine whether the sets are disjoint using the output of the k-SCAN algorithm. The number of bits exchanged is $S(\sigma - 1)$, since the state of the algorithm was communicated $\sigma - 1$ times. From the $\Omega(\tau/\sigma^3)$ complexity of MDIS [1], it follows that $\sigma = \Omega(m/k^5)$. $\square$

Lower Bound (see §5). Any algorithm for k-SEQMATCH must use $\Omega(m/k)$ space.

PROOF. We establish the lower bound by reduction from DISJ. As usual, let $X \subseteq [n]$ be the set held by Alice and let $Y \subseteq [n]$ be the set held by Bob. Alice forms the sequence

$$(x_1, k+1), (x_2, k+1), \ldots, (x_{|X|}, k+1),$$

where $x_1, x_2, \ldots, x_{|X|}$ are the elements of X. She runs the k-SEQMATCH detection algorithm (with parameter m set to n) on this sequence, suspending it immediately after reading the last element, and then sends its state to Bob. Bob forms the sequence

$$(y_1, 1), (y_2, 1), \ldots, (y_{|Y|}, 1),$$

where $y_1, y_2, \ldots, y_{|Y|}$ are the elements of Y. Bob runs the algorithm, providing the above sequence as input, suspending it immediately after reading the last element. He then sends the state of the k-SEQMATCH algorithm back to Alice, who resumes it on the sequence

$$(x_1, k+2), (x_2, k+2), \ldots, (x_{|X|}, k+2),$$

again suspending it immediately after reading the last element, and then sends the state to Bob. Bob resumes th! e simulation on the sequence

$$(y_1, 2), (y_2, 2), \ldots, (y_{|Y|}, 2).$$

They do this k times altogether, with Bob finishing the execution of the algorithm and determining its output. If X and Y are disjoint, then the aggregate packet sequence provided to the algorithm as input contains sessions whose sequence numbers are strictly increasing. However if X and Y intersect, say at an element c, then the packet sequence will contain the session

$$(c, k+1), (c, 1), (c, k+2), (c, 2), \ldots, \\ (c, k+k), (c, k),$$

which contains exactly k out-of-order packets. Thus, Bob can use the result of the k-SEQMATCH to determine if X and Y are disjoint. Since Alice and Bob exchanged the state of the algorithm $2k - 1$ times, we have $S(2k - 1) = \Omega(n)$, so $S = \Omega(m/k)$. $\square$

Lower Bound (see §6). Any algorithm for EVASION must use $\Omega(m)$ space.

PROOF. The proof is by reduction from DISJ. As before, Alice and Bob have sets $X \subseteq [n]$ and $Y \subseteq [n]$, respectively. Let $\bar{X} = [n] \backslash X$ and let $\bar{Y} = [n] \backslash Y$. Alice forms a packet sequence of the form

$$(x_1, \texttt{1ST}, \texttt{Q}), (x_2, \texttt{1ST}, \texttt{Q}), \ldots, (x_{|X|}, \texttt{1ST}, \texttt{Q}), \\ (\bar{x}_1, \texttt{1ST}, \texttt{R}), (\bar{x}_2, \texttt{1ST}, \texttt{R}), \ldots, (\bar{x}_{|\bar{X}|}, \texttt{1ST}, \texttt{R}),$$

where $x_1, x_2, \ldots, x_{|X|}$ are the elements of X and $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_{|\bar{X}|}$ are the elements of $\bar{X}$. She runs the EVASION detection algorithm (with parameter m set to n), suspending it immediately after reading the last element. She then sends the state of the algorithm to Bob. Bob forms a packet sequence of the form

$$(y_1, \texttt{2ND}, \texttt{R}), (y_2, \texttt{2ND}, \texttt{R}), \ldots, (y_{|Y|}, \texttt{2ND}, \texttt{R}), \\ (\bar{y}_1, \texttt{2ND}, \texttt{Q}), (\bar{y}_2, \texttt{2ND}, \texttt{Q}), \ldots, (\bar{y}_{|\bar{Y}|}, \texttt{2ND}, \texttt{Q}),$$

where $y_1, y_2, \ldots, y_{|Y|}$ are the elements of Y and $\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_{|\bar{Y}|}$ are the elements of $\bar{Y}$. He then resumes the algorithm and provides the remainder of his sequence as input.

If X and Y intersect, then there will be a pair of packets $(x, \texttt{1ST}, \texttt{Q})$ and $(y, \texttt{2ND}, \texttt{R})$, occurring in this order, in the sequence. Conversely, if X and Y do not intersect, there will not be such a pair of packets in the packet sequence. Using the result of the EVASION detection algorithm, Bob can determine if X and Y are disjoint. It follows that $S = \Omega(m)$. $\square$