Original SYN: Finding Machines Hidden Behind Firewalls

Xu Zhang
Department of Computer Science
University of New Mexico
xuzhang@cs.unm.edu

Jeffrey Knockel
Department of Computer Science
University of New Mexico
jeffk@cs.unm.edu

Jedidiah R. Crandall
Department of Computer Science
University of New Mexico
crandall@cs.unm.edu

***Abstract*—We present an Internet measurement technique for finding machines that are hidden behind firewalls. That is, if a firewall prevents outside IP addresses from sending packets to an internal protected machine that is only accessible on the local network, our technique can still find the machine. We employ a novel TCP/IP side channel technique to achieve this. The technique uses side channels in "zombie" machines to learn information about the network from the perspective of a zombie. Unlike previous TCP/IP side channel techniques, our technique does not require a high packet rate and does not cause denial-of-service. We also make no assumptions about globally incrementing IPIDs, as do idle scans.**

This paper addresses two key questions about our technique: how many machines are there on the Internet that are hidden behind firewalls, and how common is ingress filtering that prevents our scan by not allowing spoofed IP packets into the network. We answer both of these questions, respectively, by finding 1,296 hidden machines and measuring that only 23.9% of our candidate zombie machines are on networks that perform ingress filtering.

I. Introduction

With the advent of Software Defined Networking and increasing concerns about cybersecurity and the ability to enforce policies on networks connected to the Internet, it is becoming increasingly difficult to understand the structure of networks. Networks are no longer defined by routing alone, but also by trust relationships, firewall rules, and policies. In this paper we propose a method for finding hidden machines behind firewalls, a critical first step towards being able to measure modern networks.

An idle scan is a port scanning technique that exploits TCP/IP side channels. Antirez [1] proposed the original idle scan method in 1998. By implementing Antirez's idle scan, a scanner can scan a target machine without sending a single packet to the target using his or her own return IP address. The scan selects a "zombie" machine which has a global IP identifier (IPID), which is a unique identifier for each IP packet. A Global IPID means that the IPID associated with every packet sent by the machine shares the same global counter. There are basically three steps for the idle scan:

1) The measurement machine queries the IPID of the zombie.
2) The measurement machine sends a SYN packet to a port of the target using the zombie's IP address.
3) The measurement machine queries the IPID of the zombie again.

By measuring the changes in the IPID between the first and last query, the status of the port (either open or closed) on the target is known. This is because the open port case causes the zombie to send packets and increment its IPID (responding to unsolicited SYN-ACKs with RSTs), while the closed port case does not. However, Antirez's idle scan method requires the zombie machine to have a global IPID, which is relatively rare. The scan also assumes that the zombie is idle, hence the name "idle scan." Internet-connected hosts are seldom idle.

In 2010, Ensafi *et al.* [2] proposed a different TCP/IP side channel based on information flow in the TCP/IP SYN backlog. Ensafi *et al.*'s technique's main advantage over Antirez's idle scan was that the measurement machine does not need to send any packets at all (not even spoofed packets) to the target. Hence, if a firewall prevents the attacker from reaching the target at all the attacker can still infer the existence of the machine. However, Ensafi *et al.*'s technique cannot be used ethically for Internet measurements because it fills the SYN backlog of the zombie, causing the possibility of denial-of-service. The SYN backlog is a buffer that stores information about half-open connections where a SYN has been received and a SYN-ACK sent but the ACK reply to the SYN-ACK has not been received. Ensafi *et al.*'s technique fills this backlog with spoofed SYNs (that have the return IP address of the target) and SYNs with the return IP address of the measurement machine, and infers whether the target is responding to the zombie's SYN-ACKs with RSTs based on whether the SYNs from their machine are responded to with SYN cookies. SYN cookies [3], [4] are a type of SYN-ACK that require no state to be kept and are only ever transmitted once. They are used when the SYN backlog is full to mitigate SYN flooding denial-of-service attacks. Most SYN cookie implementations do not allow for a scaled flow control window, and filling the SYN backlog requires a high rate of SYN packets to be sent, thus Ensafi *et al.*'s technique cannot be used ethically for Internet measurement purposes.

In this paper, we present an Internet measurement technique for finding machines that are hidden behind firewalls. That is, if a firewall prevents outside IP addresses from sending packets to an internal protected machine that is only accessible on the local network, our technique can still find the machine. Our

technique is based on the technique of Ensafi *et al.*, but does not require the SYN backlog to be filled to infer information, and SYNs are sent at a very low rate.

We summarize our major contributions as follows:

1) We present a novel scan that uses a TCP/IP side channel to find hidden machines behind firewalls without causing denial-of-service. Our method also does not require a global IPID on the zombie machine, nor does it assume that the measurement machine can send packets to the target. We demonstrate our scan's effectiveness by discovering 1,296 hidden machines.
2) We propose a comprehensive direct host discovery scan which is comprised of five scans: SYN scan, SYN-ACK scan, UDP scan, ICMP scan, and ICMP fragmentation scan. The new scan we implemented was used to compare with our SYN backlog scan, meanwhile it could find more hosts up than the Nmap host discovery scan.
3) We present a novel method for testing whether the network that a machine is on performs ingress filtering to prevent spoofed IP packets from entering the network with return IP addresses within the network. Using this method, we determined that only 23.9% of networks we attempted to measure perform this kind of filtering, meaning that our novel TCP/IP side channel scan is widely applicable.

The rest of the paper is organized as follows. Section II gives some background information that is necessary for understanding our scan. Section III describes the implementation of our technique. We then describe our experimental methodology for assessing the effectiveness and applicability of our technique in Section IV, and how we perform quantitative analysis of the raw results in Section V. Results are presented in Section VI, followed by discussion in Section VII, related works in Section VIII and the conclusion in Section IX.

II. Background

In this section, we briefly review TCP basics and give some background information about different port scanning techniques which we use in this paper.

A. TCP basics

There are some rules that TCP follows [5], which are exploited by our scan:

1) A SYN packet sent to an open port will be accepted and replied to with a SYN-ACK.
2) A SYN packet sent to a closed port will be dropped, and a RST-ACK will be sent back.
3) A FIN packet sent to an open port will be dropped.
4) A FIN packet sent to a closed port will be answered a RST.
5) A SYN-ACK packet will be dropped by a machine if that machine did not send the original SYN, and a RST response will be sent back.

B. Port scan methods

Various methods can be used to implement port scanning. Generally, port scanning techniques can be classified into two types: vertical scans and horizontal scans. The former means scanning some or all ports on a single host, the latter means scanning a specific type of service in a range of IP addresses. In this section, we will discuss the most popular scans. Some of the definitions are from De Vivo *et al.* [5] and Lyon [6].

1) TCP SYN scan: In a TCP SYN scan [5], the scanner sends SYN packets to a certain port of the target machine. If the target machine replies with a SYN-ACK, it means that port is open. If the scanner receives a RST response, this means the port is closed. In this way, the scanner can learn the status of a given port. The advantage of this scan is that it does not need to establish a full TCP connection. Because of this feature of SYN scanning, it is also called half-open scanning. The disadvantage is the scanner has to use its own return IP address and has to be able to send a packet to the target, which might be prevented by a firewall.

2) SYN-ACK scan: In the TCP SYN-ACK scan [6], the scanner sends SYN-ACK packets to the target machine. If the target machine replies with RSTs, that means the target machine is up. This scan is often used to detect firewalls.

3) FIN scan: The FIN scan [5] is rarely logged (*e.g.*, by an intrusion detection system) compared to the original SYN scan because it does not consist of a normal TCP 3-way handshake. As mentioned above, a FIN packet arriving at a closed port will get a RST back; if a FIN packet arrives at an open port, it is dropped.

4) Xmas Tree, Null scan: Xmas Tree and Null scanning [5] are variations of FIN scanning. The same behavior that FIN scanning observes can also be seen with all FIN/PSH/URG flags enabled (Xmas Tree scan) in a TCP segment or no flags turned on (Null scanning). Certain firewalls focus on preventing FIN scanning but are susceptible to these two kinds of scans.

5) UDP scan and ICMP scan: The UDP scan [6] is a very different scanning method used to detect UDP open ports. It uses the fact that when a UDP packet arrives at a closed port, an ICMP unreachable message will be sent back. An ICMP scan [6] is implemented by sending an ICMP echo or timestamp request and waiting for the ICMP reply packet.

6) Fragment scan: In this paper, we introduce a scan called a *Fragment scan*. Hosts typically store fragments in a data structure called a *fragment cache* so that a fragment's datagram can be reassembled after the rest of its fragments arrive. Fragment scanning utilizes the notion that many hosts will send ICMP "reassembly time exceeded" messages when they evict entries from their fragment cache. To perform the scan, we send an IP address the first fragment of a large ICMP echo request. We then wait up to 120 seconds for it to expire from a host's fragment cache and to receive an ICMP error message. We found that, although the Windows Firewall filters the previously mentioned scan techniques, on Windows Vista and later, the default firewall settings still permit "reassembly time exceeded" messages to be sent unfiltered. Thus, this scan

is useful for detecting Windows machines even if they are running the Windows Firewall.

III. Implementation

In this section, we describe the implementations of our indirect and direct scans.

A. Our backlog scan

In this subsection, we present the details of our SYN backlog scan in three parts: First we give a brief description about the SYN backlog and how it is implemented in Linux, in particular how it behaves as the number of half open connections increase. Then we explain what we do to infer the SYN backlog size of a Linux machine. Finally, we give the details of our SYN backlog scan based on the understanding of the previous two parts.

1) SYN backlog preliminaries: Our scan relies on a TCP/IP side channel in the SYN backlog to make inferences. The SYN backlog is a buffer to store half open connections. The status when a machine receives a SYN and answers with a SYN-ACK, but has not received an ACK reply to its SYN-ACK to finish the "TCP three way handshake", is called "half open". A half-open connection stays in the SYN backlog until it receives an ACK to complete the normal handshake process or a RST, ICMP error, or ARP timeout to drop the connection. If no answer comes back, the SYN-ACK is retransmitted some fixed number of times (typically between 3 and 5 times) until the half-open connection times out (typically between 30 and 180 seconds) and is then aborted.

In the Linux kernel versions 2.3 and later, if the SYN backlog is more than half full, some of the older entries in the backlog will be evicted to reserve half of the backlog for the *young* requests. A young request is a request that has not been retransmitted yet. The idea of SYN backlog management is to "keep most of the young entries and remove old ones from the queue which have been there for quite some time and have not yet been accepted or acknowledged" [7]. This feature causes information flow before the resource is exhausted, so that we can make inferences without causing denial-of-service.

2) Inferring SYN backlog size: The first inference we make based on information flow in Linux is about the backlog size of a Linux machine. Below we will talk about how to calculate the possible SYN backlog size of a Linux machine. Then we will give a method to infer the SYN backlog size based on the range obtained from the previous calculation.

The Linux SYN backlog size depends on three kernel variables:

1) The "backlog" argument of the listen() system call
2) The kernel variable *net.core.somaxconn*
3) The kernel variable *net.ipv4.tcp_max_syn_backlog*

To calculate the backlog size, the kernel takes the first variable (an argument passed to the listen() call), adds 1 and then picks the next power of two, which is the final backlog size. The lower bound of this variable is hard coded to 8 in the kernel, and the upper bound depends on the minimum value of the second and third variables. Although Linux sets *tcp_max_syn_backlog* based on the memory of the system (minimum is 128), the default value of *somaxconn* is 128. Thus, the typical range for the SYN backlog size of a Linux system is 16 to 256.

To implement our inference technique to find out the backlog size of a given machine on the Internet, we make the assumption that "the SYN backlog size of the machine is x" and iteratively increase x. We start from the smallest possible size (16) and work up from there, to be non-intrusive. We send $3/4 \cdot x$ SYN packets, without answering ACKs to SYN-ACKs from the machine. If the machine's backlog size is x, more than half of it is full and some of SYNs we sent will be evicted. If the backlog size is greater than x, no eviction behavior will be observed, so the guessed size of the backlog is doubled to $2x$ and we repeat the test. The experiment is run until it successfully returns the backlog size of the Linux machine or it reaches the threshold 256 (typically the largest size of the backlog). If the backlog size appears to be greater than 256, we abort and do not use that machine as a zombie.

Below we explain a method to test whether an entry stayed in the SYN backlog or not. For every original SYN packet that we send, we create another duplicated SYN. The duplicated SYN has the exact same information (source and destination port, source and destination IP address) corresponding to the original SYN, except it has a different sequence number that is less by one. For Linux, the duplicated SYN we send may have two kinds of answers:

1) If the original SYN still stays in the SYN backlog, an ACK packet will be answered to it.
2) If the original SYN has been evicted, an SYN-ACK packet will be answered to the newly arrived duplicated SYN.

Therefore we can find out the status of previously sent SYN packets by observing the machine's answers to duplicated SYN packets.

3) Implementing the backlog scan: Now we talk about how to exploit the SYN backlog side channel and use it to find hidden machines on the Internet.

Our backlog scan also involves a third machine called a "zombie". The procedure of our scan is as follows. We assume that we have already performed the scan from the previous subsection and we therefore know the zombie's backlog size s. We then fill $3/4$ of the zombie's backlog again. However, this time the packets contain two parts: **1.** Spoofed SYN packets sent to the zombie which use the target machine's IP address as the source IP address. **2.** SYN packets from our scan machine to the zombie, using the return IP address of the scan machine, and we call these packets *canaries*. Spoofed SYN packets and canaries are mixed and shuffled to be sent in a completely random order, and then sent at a rate of 5 packets per second to the zombie machine. Three quarters of the zombie's SYN backlog are now filled by an even number of spoofed SYNs and canaries, so each of them has a number of packets equal to $3/8$ of the zombie's SYN backlog size. In order to ensure that the Linux kernel evicts SYNs independently and without treating canaries and probes

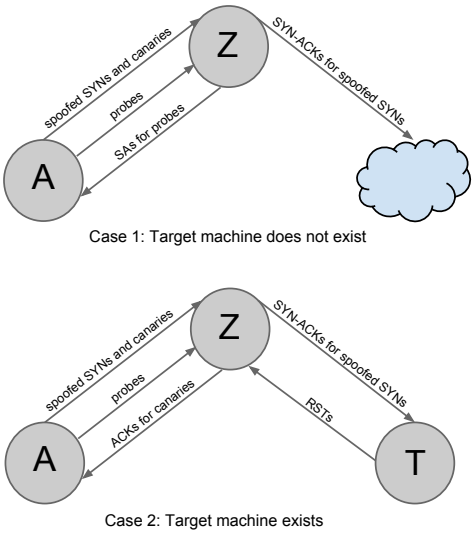

Fig. 1. Two cases in our scan.

differently, we use random (without replacement) source port numbers for all SYN packets created.

Next we send duplicates of the canaries to test their status in the backlog. We call these duplicates *probes*. As discussed above, probes are the exact same as canaries, except the sequence number of each corresponding packet is smaller by 1. Two kinds of answers may come back, as shown in Figure 1:

1) If the target machine does not exist, the SYN backlog is filled with spoofed SYN packets and canaries. Some of the canaries will be evicted. We will therefore observe SYN-ACKs as answers to probes.
2) If the target machine exists, it sends RSTs to SYN-ACKs from the zombie. The SYN backlog is less than half full because only the canaries stay. We will therefore observe ACKs as answers to canaries.

Two special cases may affect the result of our technique when the target machine does not exist.

1) Tested target is in the same subnet with the zombie: In this case the zombie will send an ARP request to the target. Spoofed SYNs will be removed from zombie's SYN backlog because of the ARP request timeout. Thus the SYN backlog will be less than half full. However, with Linux versions 3.2 and earlier, this behavior is rate-limited to 1 per second. With this rate limitation, our scan can still fill more than half of the backlog at a rate of 5 packets per second when using these zombies with SYN backlog sizes of at least 256.
2) Tested target is not in the same subnet with the zombie: Some gateway routers may send ICMP host unreachable messages back to the zombie. The SYN backlog of the zombie will thus be less than half full. There is typically a rate-limit for sending this type of ICMP message of 1 per second.

B. Nmap direct scan

We implemented Nmap's host discovery scan [8] *via* "nmap -n -sn". These options let Nmap run its built-in host discovery scan without resolving DNS. We execute this command using a privileged account on the measurement machine. By default, nmap sends an ICMP echo request, an ICMP timestamp request, a TCP SYN to port 443, and a TCP ACK to port 80 on each machine.

C. Our own direct scan

We also implemented a new comprehensive direct scan. This scan is a hybrid scan to test the liveness of an IP address. It is comprised of six types of scans: TCP SYN Scan, TCP SYN-ACK Scan, UDP Scan, ICMP Echo Scan, ICMP Timestamp Scan, and Fragment Scan. In the TCP SYN Scan and SYN-ACK Scan, we target more ports (21, 22, 80, 135, 139, 443, 445, 631) than Nmap's host discovery scan. In the UDP Scan, we choose the probably unused port 5000 and wait for an ICMP Port unreachable error. In the Fragment Scan, we send only the first fragment of an ICMP Echo request and then wait for the reassembly timeout ICMP error message as a response. This is effective for windows machines because some Windows versions have a firewall that blocks SYNs but still allows the reassembly timeout message to pass.

D. Ingress filtering

Many zombies are on networks subject to *ingress filtering*, *i.e.*, they are on networks that filter incoming packets from outside their network if those packets have a source address from inside their network. Since our scanning technique relies on spoofing packets from other hosts on the zombie's network, if the zombie's network performs ingress filtering, our scan will not work and will spuriously report all hosts on the zombie's network as alive.

To test if a zombie's network performs ingress filtering, we use the zombie's fragment cache as a side channel to determine if packets spoofed from other hosts on the zombie's network are reaching the zombie. Since our zombies are all Linux machines, we adapt our test to Linux's fragment cache implementation. Namely, we use the time it takes to fill Linux's fragment cache to determine if there is ingress filtering.

Linux limits the size of the fragment cache according to the number of bytes used for storing fragments. When the cache is full and the storage of an incoming fragment would exceed its maximum size, the kernel begins *pruning* fragments from its cache in FIFO order, typically until $1/4$ of the cache is free, although this number is configurable.

Our test begins by performing a *fragment cache size measurement* by measuring the maximum number of 1420-byte fragments that fit into the target Linux machine's fragment cache. We will call this number the *size* of the fragment cache. We perform this measurement by sending large TCP SYN datagrams to the open port of the zombie that we fragment into two halves. As Linux ignores extraneous TCP payloads in SYN datagrams, once these datagrams are completed in the

zombie's fragment cache, the zombie will respond with the appropriate SYN-ACK.

One might naively measure the number of fragments the Linux fragment cache can hold by splitting each datagram d_i of n datagrams $d_1, \ldots, d_n$ into two fragments, a first-half fragment f_i and a second-half fragment s_i, and then sending $f_1, \ldots, f_n$ followed by $s_n, \ldots, s_1$, for some n larger than the fragment cache size. To avoid quickly kicking out any fragments that might be in the zombie's fragment cache, we might send these packets out evenly over an up to 30 second interval, as after 30 seconds, Linux times out fragment cache entries, evicting them. However, the number of SYN-ACKs received from this method would not be the size of the cache. Rather, it would be the number of fragments remaining in the cache after the kernel prunes its entries.

Thus, instead we, over the span of 29 seconds, send fragments from $2n$ different datagrams and in a different order than before. We send f_1 and f_2, then s_1, followed by f_3 and f_4, then s_2, and so on, until we have sent all of $f_1, \ldots, f_{2n}$ and $s_1, \ldots, s_n$. Thus, we take turns between placing two new datagram entries in the fragment cache and attempting to complete the oldest datagram we have not tried completing by sending its missing second-half fragment. When the kernel prunes its fragment cache for the first time, any future second-half fragments will only try to complete fragments that were already evicted, and so we will cease receiving any more SYN-ACKs from the host. At this moment, the number of entries that were in the fragment cache will equal the number of SYN-ACKs that we have received.

To then measure if fragments spoofed from some address are reaching the zombie, we perform a modified version of the fragment cache size measurement we call the *spoofed size measurement*. This measurement is similar to the fragment cache size one, except every time we send either a first-half or second-half fragment from us, we also send an analogous one from the spoofed host. If our spoofed packets from that host are reaching the zombie, then the fragment cache will fill twice as quickly, and we will measure the fragment cache to be half of its size.

To test if the network filters a certain address, we perform a *filtering test*. In a filtering test, we alternatively perform both the fragment cache size and spoofed size measurements 10 times. Let $\bar{x}$ be the average result of the fragment cache size measurements and $\bar{y}$ be the average result of the spoofed size measurements. If $\bar{y} < 0.55\bar{x}$, then we conclude that incoming packets from the spoofed address are filtered. If $\bar{y} > 0.95\bar{x}$, then we conclude that they are not. Otherwise, we consider the result inconclusive.

To decide if there is ingress filtering on a zombie's network, if the zombie's address is $a.b.c.d$, we perform the filtering test to determine if $a.b.c.(d \oplus 0\text{x}01)$ and $a.b.c.(d \oplus 0\text{x}80)$ are filtered on the zombie's network. These two tests effectively test whether the zombie is on a subnet of size /31 or larger that performs ingress filtering and whether the zombie is on a subnet of size /24 or larger that performs ingress filtering, respectively.

Thus far, we have assumed that we know some appropriate value of n to use larger than the fragment cache size. Although we could use some very large value of n surely larger than any fragment cache size, if n is too large, we will be sending packets and filling the zombie's fragment cache at an unnecessarily high rate. Thus, to find an appropriate value of n, we first perform a fragment cache size measurement with $n = 11$, and we continue doubling n until we find an n such that the measured fragment cache size is less than $9n/10$, and then we use that value of n for all future measurements on that zombie.

IV. Experimental Setup

All of the measurement machines we used were Linux machines running Ubuntu 14.04. To avoid the influence of ARP timeouts as discussed in Section III, we chose Linux machines with kernel version 3.2 and earlier as zombies. In this section we describe how we selected zombies and ran experiments. The two main purposes of our experiments were:

1) Demonstrate the efficacy of our technique by locating hidden machines, *i.e.*, machines that a very comprehensive direct scan cannot find.
2) Determine how common ingress filtering is, to assess the applicability of our technique.

One measurement machine was used to generate random IP addresses. We sent SYNs to ports 21, 22, 80, 443, and 631 of each randomly generated IP and sniffed for the response for 3 seconds. IP addresses which responded with SYN-ACKs were recorded. Then we took the IP addresses collected in the first step as input and removed duplicates. We ran "nmap -O" (Nmap Operating System Detection) [9] with a timeout of 60 seconds to all these IP addresses. Basically, Nmap sends TCP and UDP packets and tests TCP Initial Sequence Number (ISN) sampling, TCP option support and ordering, IPID sampling, initial window size check, *etc.*, and then compares the results with its own operating system database to see if there is a match. IP addresses determined to be running the Linux operating system were recorded. Some answers returned by Nmap were not accurate, so we discarded those answers and only recorded answers with 100% certainty from Nmap that the machine was a Linux machine. We wanted to find Linux machines with versions earlier than 3.3. We found that Linux version 3.0 and earlier has its TCP timeout period (the amount of time it takes for a SYN to time out and be removed from the SYN backlog) hardcoded to 3 times longer than version 3.1 and later. We used this feature to fingerprint Linux machines with kernel version 3.0 and earlier (thus ensuring they were older than 3.3) by testing TCP timeout periods. After collecting all the Linux zombie candidates, we tested their SYN backlog sizes using the method described in Section III.

For each zombie machine we chose, the machines in the same /24 subnet were considered target machines for us to try to discover the liveness of with both direct and indirect scans. Before scanning, we queried reverse DNS entries and looked up all the domain names for all target machines. Then

we ran our SYN backlog scan, the Nmap host discovery scan, and our comprehensive direct scan on every target machine. We re-ran the SYN backlog indirect scan three times for each experiment to minimize the effects of bursty packet loss. After finishing testing all machines in same /24 subnet with a zombie, we ran the ingress filtering test on that zombie. The scans ran on six measurement machines using multiprocessing. The whole scan period was about 15 days in length. In Section III we discussed two possible cases that may affect our scan. To ensure that there are enough SYN's in zombie's backlog to withstand those removed via linux's arp timeout behavior and any possibly removed via rate-limited ICMP host unreachable errors, we selected our final data from zombies with a minimum backlog size of 256.

V. Analysis

We use statistical hypothesis testing to determine whether a target address is alive on the zombie's network, using as our null hypothesis and our alternative hypothesis

H_0 : the address was never alive during our test

H_a : a machine at that address reset some SYN-ACKs.

If we are able to reject the null hypothesis and accept the alternate hypothesis with high statistical significance, then we can safely assume that the target machine is up, *i.e.*, alive.

We send both spoofed SYNs and canaries to fill $3/4$ of the SYN backlog of a zombie machine. When the null hypothesis is true, then no machine answers at the target address with any RST in response to any of the zombie's SYN-ACKs. The SYN backlog is more than half full and some old entries are evicted because half of the SYN backlog is reserved for young entries. When the null hypothesis is false, then that address is alive and, in an ideal case, its machine responds to every one of the zombie's SYN-ACKs with RSTs and our experiment would show no evicted old entries. However, in practice, machines may not be consistently up and there is the possibility of packet loss in all directions of links and machines. Moreover, sometimes packet loss changes the number of evicted canaries that we measure. It might be obvious to say a machine is up when our evicted number is, for example, 0. But for numbers such as 4 or 5, it may not be clear how to decide whether we can assume that the machine is up, which is why we apply a hypothesis test. We need to calculate the point at which we can consider the machine to be up, which is the critical number, c. Meanwhile, we also need to decide how critical we will be, *i.e.*, how much statistical significance we will require to assume that the machine is up. For example, in this case, $c = 0$ is more critical than $c = 5$. In other words, the decision we make on c determines how often we would falsely reject the null hypothesis (Type 1 error). In our experiment, we selected the maximum acceptable probability of Type 1 error (also called the significance level) $\alpha = 0.05$. We calculate critical value c depending on this significance level.

As discussed above, if we assume that the null hypothesis is true, some entries in the SYN backlog will be evicted because more than half of it is full. Whether a specific entry is evicted or not is based on the location it is hashed to in the SYN backlog hash table. The spoofed SYN packets and canaries we created have random source port numbers; therefore, the whole process of evicting packets can be simulated by randomly selecting entries from all SYN packets in the SYN backlog. The evicting process stops after the number of entries in the SYN backlog drops under half. By sending probes, we can know how many canaries are evicted.

When n draws are taken, without replacement, from a finite population size of N that contains exactly K successes and where each draw is either a success or a failure, the probability of k successes has a hypergeometric probability distribution. Thus, the number of evicted canaries is hypergeometrically distributed. More specifically, the number of evicted canaries k (the test statistic) follows a hypergeometric distribution. If we define s to be the zombie machine's SYN backlog size, then the number of successes statistic K in the population is simply the number of canaries that arrived at the zombie machine,

$$K = C - L_C = 3/8 \cdot s - L_C,$$

where C is number of canaries we sent and L_C is the number of these canaries that were lost due to packet loss.

The population size N is all of the spoofed SYNs and canaries that arrived at zombie machine. In this case, we cannot observe the number of packets lost for spoofed SYNs L_S, because answers to spoofed SYNs are off-path. We estimate that $L_S = L_C$ because spoofed SYNs and canaries are sending aggregately in the same time period and because in our experiment $C = S$, where S is the number of spoofed SYNs we sent. So for our population size, we have

$$N = (C - L_C) + (S - L_S) = 2 \cdot (C - L_C) = 2 \cdot K.$$

The number of draws n is how many entries are evicted. As we discussed above, the evicting process stops when the total number of entries in the backlog drops under half, and it does not necessarily stop when it reaches exactly half the size of the SYN backlog. In other words, it might drop even more SYNs after it reaches the half-full threshold. And so we have

$$n \geq N - 1/2 \cdot s.$$

The number of successes k is simply the number of evicted entries that we observed for canaries. We measure the packet loss of evicted canaries by counting answers to probes. If the number of probes answered is fewer than the number of probes we sent, then there is packet loss. There are two types of answers: ACKs for canaries (meaning that the canary stayed in the SYN backlog) and SYN-ACKs for probes (meaning that the canary was evicted). Packet loss could occur in the probes we sent or in the two types of answers we get. Without making guesses about where exactly the packet loss happened, we want to be conservative and bias the result to H_0. That is to say, we assume that the answers that get lost are always SYN-ACKs for probes. This way we count more evicted canaries, which makes it harder to reject H_0. This can only make the

result more statistically significant. So we calculate k as

$$k = R - A,$$

where R is the number of probes sent and A is the number of ACKs received from our probes.

Here the p-value, or probability of seeing data at least as extreme as what we measured, is simply $P(x \leq k)$. Since k is geometrically distributed, our p-value is

$$P(x \leq k) = \sum_{x=1}^{k} \frac{\binom{K}{x}\binom{N-K}{n-x}}{\binom{N}{n}}.$$

We chose the possible smallest value of n, which is $n = N - 1/2 \cdot s$, because we want to be conservative about H_0, and a smaller n results in a bigger P value, which makes it harder to reject H_0.

As we discussed in section IV, each experiment is repeated 3 times to avoid the influence of packet loss. When selecting the results, there are two cases:

1) At least one of the three results does not have packet loss in the traffic we can observe.
2) All the three results have packet loss in the traffic we can observe.

For case 1, we would select one result that is without packet loss. If there is more than one result which does not have packet loss, we chose the result with the highest evicted number of canaries k, to remain conservative and bias us towards H_0. For case 2, we would select the one result which has the smallest packet loss rate, so as to minimize the influence of packet loss. If the one we select still has a high packet loss rate, (greater than 30% in this case, because it would cause population of less than half of the SYN backlog size) we throw out the data and return a failure error message for this target machine.

In Section III, we discussed two cases that may affect our scan. To give allowance of our model to handle these cases, we adjusted certain variables in our model. The zombie's SYN backlog size is always 256 for the results presented in this paper. At a rate 5 packets per second, our experiment takes less than 40 seconds to fill 3/4 of backlog. Assuming the target does not exist, the maximum number of evicted SYNs due to ARP request timeouts or ICMP unreachable messages is 40. Therefore, we subtracted both the number of successes statistic K and number of draws n by 40, respectively.

VI. Results

In this section we describe the results of our experiments.

A. Ingress filtering results

We were able to collect data from 289 zombie machines with backlog size 256 during a 15 day experiment. We scanned machines in the same /24 subnet for each zombie. After collecting the scan results, we performed the ingress filtering test on each zombie machine. We found that 69 (23.9%) of the zombies had ingress filtering. Among them, 55 (79.7%) had ingress filtering on a /24 or larger network; 14 (20.3%) experienced ingress filtering on a /31 or larger network (*e.g.*, /27) but not on a /24 or larger network. Their backlog scan results for the /24 showed almost no evicted canaries in some smaller subnet. 176 (60.9%) of the zombies did not have ingress filtering on a /24 or larger network, and 44 (15.2%) of the zombies' ingress filtering status could not be determined. The results show that since most (60.9%) of zombie machines we selected do not have ingress filtering on their network, our technique is widely applicable on the Internet.

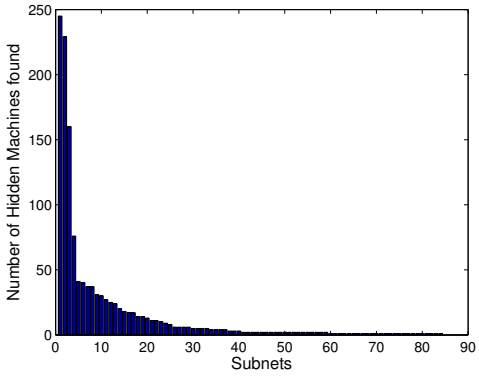

Fig. 2. Distribution of the number of hidden machines per subnet.

B. Indirect scan result

For the 176 zombies we found that did not experience ingress filtering, we applied statistical analysis to the results and calculated a p-value for each individual experiment. There are 14,503 addresses for which we rejected the null hypothesis, which means that these addresses were alive. Comparing to our direct scan, we found 1,351 more machines that were hidden to direct scans. Finally, we removed hidden machines if there are ICMP unreachable error message collected by our direct scan. The number of hidden machines we found is 1,296, distributed across 84 different subnets out of the 176 tested. Figure 2 shows the distribution of those 1,296 machines in terms of how many such machines existed on each subnet. The x axis is the different subnets sorted by the rank of how many hidden machines were found on that subnet, and the y axis is the number of hidden machines found. From the figure, we can see the variety in the number of hidden machines found in different subnets (which ranges from 1 to 245). Most subnets (about 74%) had less than 10 hidden machines.

VII. Discussion

In the previous section, we demonstrated the efficacy of our indirect scan technique, based on a TCP/IP side channel in the Linux SYN backlog. In this section, we discuss how our direct scan compares to Nmap's direct scan, and present some limitations of both our indirect and direct scan techniques. We also discuss how we informed network operators of our experiments and gave them the ability to opt out.

TABLE I
ADDITIONAL HOSTS FOUND VERSUS NMAP VIA DIRECT SCAN BY DIRECT SCAN TYPE

Scan method	Additional hosts found	Percentage
SYN	569	55.7%
SYN-ACK	233	22.8%
ICMP-ECHO	25	2.45%
ICMP-TS	32	3.13%
ICMP-FRAG	351	34.4%
UDP	173	16.9%

A. *Nmap* vs. *our scan*

We compared the results of our comprehensive direct scan with Nmap's host discovery scan. Our direct scan found 39,163 machines that were up, while the Nmap host discovery scan found 38,252 machines. There are 1,131 differing results between our direct scan and Nmap's host discovery scan, 1,021 (90.3%) of them are reported as "up" only by our direct scan, and only 110 (9.7%) are reported as "up" only by Nmap's host discovery scan. Table I shows details in terms of the percentage of each technique to help find hosts which were blind to Nmap's host discovery scan. Because a machine could be found by multiple scans, the sum of the percentages in the table is above 100%.

B. *Limitations of our SYN backlog scan*

There are limitations of our technique. First, our technique to exploit the Linux SYN backlog side channel is non-intrusive if the zombie we scan is in a normal status. However our current technique does not consider the case that the zombie is scanned by other scanners at the same time. Our technique requires sending at a rate 5 packets per second for about 60 seconds. If a scanner scans the same machine at the same time using our technique, the packet rate will reach up to 10 packets per second. Based on the result of a simulation experiment we set in a virtual environment, any packet rate faster than 9 packets per second will fill the Linux SYN backlog because the kernel will not be able to drop old entries as fast as the SYN backlog is filling. Therefore, in this case the machine's backlog will be totally full and the server will send SYN cookies. SYN cookies still allow other clients to connect to the server, but the Linux implementation of SYN cookies does not support window scaling so the flow control of the connection may be more limiting. This is a rare case, and Internet hosts typically have their bandwidth limited by congestion control rather than flow control. Nonetheless, in future work we plan to develop an adaptive scan that backs off if it is detected that others are also sending SYNs to the zombie and leaving them in a half-open state. Second, although our statistical hypothesis model has allowance for some special cases (that SYNs could be removed because of ARP request timeouts or ICMP unreachable error messages when a target machine does not exist), we have not thoroughly tested the assumptions we made about applicable rate limits for a variety of operating systems and versions for the host and gateway router.

Another two limitations are worth noting for our comprehensive direct scan. Our direct scan targets a limited number of popular ports in the SYN scan and SYN-ACK scan. However, some other ports might be open in a target machine and a firewall is preventing outside connections except on that open port. For example, we found one target machine with port 25 open for an SMTP service, which appeared to be down to our direct scan but was found by our indirect scan. Furthermore, we did not implement multiple experiments in our direct scan, which makes it susceptible to packet loss. This can be seen in the comparison of our direct scan with Nmap's host discovery scan. Although all the techniques used in Nmap's host discovery scan are included in our direct scan, Nmap still found 110 (9.7%) machines to be up which appeared to be down according to our direct scan. Because of these two limitations, some of the machines not located by our direct scan that were located by our indirect scan may not be "hidden" in the sense that they are completely invisible from outside the firewalled network, but note that our direct scan is more comprehensive than existing direct scans and still such machines cannot be found via our direct scan. Also, our direct scan includes a SYN-ACK scan while our indirect scan is based on the target replying to unsolicited SYN/ACKs, meaning that with respect to SYN-ACKs the target is definitely hidden behind a firewall.

C. *Opting out of measurements*

During our scans, the scanning machines all were serving web pages with an explanation of our scan and contact information for network operators who wanted us to exclude their networks from our experiments. At no time during our experiments were we contacted by any network operators about our experiments. Because of the low rate at which we send SYN packets, our technique is non-intrusive.

VIII. RELATED WORK

Staniford *et al.* [10] and Gates *et al.* [11] focus on large enterprise network protection. Leckie and Kotagiri [12] use a probabilistic approach to detect port scans. Treurniet [13] aims to detect stealthy scans using classification schema. Muelder *et al.* [14] proposes a visualization for port scan detection. Jung *et al.* [15] develop a fast port scanning detection method using the theory of sequential hypothesis testing. Other works [16],[17], [18] use a neural network approach to detect malicious port scanning. Gates [19], [20] and Kange *et al.* [21] consider stealthy port scans that are based on using many distributed hosts. There has also been some research on improving port scans, such as port scan techniques that increase the speed of horizontal scans based on techniques that use the same principle as SYN cookies [22] [23] [24] [25].

Our work is based on the work of Ensafi *et al.* [2], [26], [27], since we use the SYN backlog as a side channel for making inferences. There are other works using different types of side channels. Morbitzer [28] explores idle scans in IPv6. Qian *et al.* [29], [30] infer the TCP sequence number of a connection and perform off-path TCP/IP connection hijacking using a

firewall-based side channel. Some works use global IPID fields to perform inference for Internet measurement purposes. Chen *et al.* [31] explore new uses of the IPID to infer the amount of internal traffic generated by a server, the number of servers in a large scale server complex, and one-way delays to a target computer. Bellovin [32] describes a technique to detect NATs and count the number of hosts behind them. Kohno *et al.* [33] use the IPID to perform remote device fingerprinting. Our work is based on the SYN backlog TCP/IP side channel, not globally incrementing IPIDs.

IX. Conclusion and Future Work

In this paper, we presented a new Internet measurement technique that uses TCP/IP side channels to find machines hidden behind firewalls. Our technique can find machines which are behind a firewall that prevents outside IP addresses from sending packets to the internal network. Our technique was shown to be widely applicable on the Internet by our novel ingress filtering test, and is also resistant to packet loss due to the use of our statistical analysis model. The results show the existence of hidden machines on the Internet by comparing with our comprehensive direct scan. Planned future work includes using a slower packet rate to implement our technique, to make it non-intrusive even when there are other scanners scanning the same machine. Also the direct scan can be improved by targeting more common ports and doing multiple experiments to be robust to packet loss.

With respect to Internet measurement, our proposed technique is a first step towards being able to measure firewall rules, trust relationships, and all of the complexities that define today's Internet.

X. Acknowledgements

We would like to thank the anonymous reviewers for valuable feedback. This material is based upon work supported by the U.S. National Science Foundation under Grant Nos. #0844880, #0905177, #1017602, #1314297, and #1420716. Jed Crandall is also supported by the DARPA CRASH program under grant #P-1070-113237.

References

[1] Antirez, "new tcp scan method," Posted to the bugtraq mailing list, 18 December 1998.

[2] R. Ensafi, J. C. Park, D. Kapur, and J. R. Crandall, "Idle port scanning and non-interference analysis of network protocol stacks using model checking," in *Proceedings of the 19th USENIX Security Symposium*, ser. USENIX Security'10. USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929843

[3] D. J. Bernstien, "SYN Cookies," http://cr.yp.to/syncookies.html.

[4] "SYN Cookies," http://en.wikipedia.org/wiki/SYN_cookies.

[5] M. De Vivo, E. Carrasco, G. Isern, and G. O. de Vivo, "A review of port scanning techniques," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 2, pp. 41–48, 1999.

[6] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[7] S. Seth and M. A. Venkatesulu, *TCP/IP ARCHITECTURE, DESIGN, AND IMPLEMENTATION IN LINUX*. Hoboken, New Jersey: John Wiley & Sons, Inc, 2008.

[8] "Nmap Host Discovery," http://nmap.org/book/man-host-discovery.html.

[9] "Nmap OS Detection," http://nmap.org/book/man-os-detection.html.

[10] S. Staniford, J. A. Hoagland, and J. M. McAlerney, "Practical automated detection of stealthy portscans," *Journal of Computer Security*, vol. 10, no. 1, pp. 105–136, 2002.

[11] C. Gates, J. J. McNutt, J. B. Kadane, and M. I. Kellner, "Scan detection on very large networks using logistic regression modeling," in *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*. IEEE, 2006, pp. 402–408.

[12] C. Leckie and R. Kotagiri, "A probabilistic approach to detecting network scans," in *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*. IEEE, 2002, pp. 359–372.

[13] J. Treurniet, "A network activity classification schema and its application to scan detection," *Networking, IEEE/ACM Transactions on*, vol. 19, no. 5, pp. 1396–1404, 2011.

[14] C. Muelder, K.-L. Ma, and T. Bartoletti, "Interactive visualization for network and port scan detection," in *Recent Advances in Intrusion Detection*. Springer, 2006, pp. 265–283.

[15] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*. IEEE, 2004, pp. 211–225.

[16] B. Soniya and M. Wiscy, "Detection of TCP SYN scanning using packet counts and neural network," in *Signal Image Technology and Internet Based Systems, 2008. SITIS'08. IEEE International Conference on*. IEEE, 2008, pp. 646–649.

[17] J. Cannady, "Artificial neural networks for misuse detection," in *National information systems security conference*, 1998, pp. 368–81.

[18] J. Li, G.-Y. Zhang, and G.-C. Gu, "The research and implementation of intelligent intrusion detection system based on artificial neural network," in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 5. IEEE, 2004, pp. 3178–3182.

[19] C. Gates, "Co-ordinated port scans: a model, a detector and an evaluation methodology," 2006.

[20] ——, "Coordinated Scan Detection." in *NDSS*, 2009.

[21] M. G. Kang, J. Caballero, and D. Song, "Distributed evasive scan techniques and countermeasures," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2007, pp. 157–174.

[22] "Scanrand," https://www.sans.org/security-resources/idfaq/scanrand.php.

[23] "Unicorn Scan," http://www.unicornscan.org/.

[24] "Zmap," https://zmap.io/.

[25] "MASSCAN: Mass IP port scanner," https://github.com/robertdavidgraham/masscan.

[26] R. Ensafi, J. Knockel, G. Alexander, and J. R. Crandall, "Detecting intentional packet drops on the Internet via TCP/IP side channels."

[27] ——, "Detecting intentional packet drops on the Internet via TCP/IP side channels: Extended version," *CoRR*, vol. abs/1312.5739, 2013, available at http://arxiv.org/abs/1312.5739.

[28] M. Morbitzer, "TCP Idle Scans in IPv6," Master's thesis, Radboud University Nijmegen, The Netherlands, 2013.

[29] Z. Qian and Z. M. Mao, "Off-path TCP sequence number inference attack," in *Security & Privacy*. IEEE, 2012.

[30] Z. Qian, Z. M. Mao, and Y. Xie, "Collaborative TCP sequence number inference attack: how to crack sequence number under a second," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 593–604. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382258

[31] W. Chen, Y. Huang, B. F. Ribeiro, K. Suh, H. Zhang, E. de Souza e Silva, J. Kurose, and D. Towsley, "Exploiting the IPID field to infer network path and end-system characteristics," in *Proceedings of the 6th international conference on Passive and Active Network Measurement*, ser. PAM'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 108–120.

[32] S. M. Bellovin, "A technique for counting NATted hosts," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 267–272.

[33] T. Kohno, A. Broido, and K. C. Claffy, "Remote physical device fingerprinting," *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 93–108, 2005.