Spring 4-10-2018

Next Generation TCP/IP Side Channels

Xu Zhang

Xu Zhang

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Jedidiah Crandall , Chairperson

Patrick Bridges

James Plusquellic

Benjamin Edwards

Next Generation TCP/IP Side Channels

by

Xu Zhang

B.A., Mathematics, Dalian University of Technology, China, 2009
M.E., Computer Engineering, Daegu University, South Korea, 2011
M.S., Computer Science, University of New Mexico, 2013

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May 2018

Acknowledgments

I would like to thank my advisor, Professor Jedidiah R. Crandall for supporting and guiding me on my doctoral research over the years. I would also like to thank my dissertation committee members: Professor Patrick Bridges, Professor James Plusquellic, and Dr. Benjamin Edwards for valuable feedback. I would like to thank Jeffrey Knockel for his contribution of a technique for detecting ingress filtering in Section 3.3.4 and insight of Linux IPID behavior in Chapter 5.
I would like to thank my fiancee, Lengge Se for her company and encouragement, and my parents for teaching me to pursue my dreams and never give up.

Next Generation TCP/IP Side Channels

by

Xu Zhang

B.A., Mathematics, Dalian University of Technology, China, 2009

M.E., Computer Engineering, Daegu University, South Korea, 2011
M.S., Computer Science, University of New Mexico, 2013

Ph.D., Computer Science, University of New Mexico, 2018

Abstract

Side channel techniques have been developed in recent years to fulfill various tasks in modern computer network measurements. However, due to their nature, these techniques are typically limited in terms of both fidelity and their ability to be used on the real Internet without raising ethical concerns because of packet rates. I propose the next generation of TCP/IP side channel techniques that exploit information flow in modern systems' network stacks to overcome weaknesses in previous techniques. The proposed work is novel, non-intrusive, and can carry out measurements with high fidelity. I achieved this by deeply understanding the behaviors of modern systems' network stacks and balancing the trade-offs (e.g. packet rate and fidelity) by applying suitable mathematical models. My work comprises three novel tools which each solve different challenges in current network measurement.

Firstly, I propose 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, the technique can still find the machine. Secondly, I present an improved off-path round-trip time (RTT) measurement technique based on [11] that can, with high fidelity, measure the RTT between essentially any two machines (A and B) on the Internet without having special access to A or B or having any presence in the path between A and B. Finally, I proposed a new scanning technique that can perform network measurements such as: inferring TCP/IP-based trust relationships off-path, stealthily port scanning a target without using the scanner's IP address, or detecting off-path packet drops between two international hosts. The thesis statement of my dissertation is: *Previous side channel techniques can be improved and used to solve new challenges in current network measurement based on deeply understanding the modern systems' network stack behavior and building corresponding mathematical models to balance trade-offs between fidelity and ethical concerns related to packet rates.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Penetration testers and network researchers have a variety of tools they can use to gather information about target networks or hosts, such as: traceroute to understand routing information or nmap to scan open ports of a host. However, 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 nowadays. Networks are no longer defined by routing alone, but also by trust relationships, firewall rules, and policies. Unfortunately, none of the existing tools are sophisticated enough to address the aforementioned new challenges.

Imagine that a digital forensics expert needs to know information about a host on a network of interest. Current tools, such as nmap, are totally blind to machines behind the firewall which do not answer to any kind of TCP, UDP, or ICMP scans. Therefore, the digital forensics expert has no way to clearly understand the full topology of the network.

Also imagine that a network researcher wants to know the general network delay between two countries in Africa (*e.g.* Niger and Chad), or to learn the trust relationship between any two international nodes. However, he or she does not have shell access to the operating systems on any hosts located in those countries, and it is hard to find enough nodes for collecting data on these two continents using network services such as Planetlab [21] and RIPE Atlas [5]. The rudimentary routing measurement tool traceroute simply does not work in this case.

I propose the next generation of TCP/IP side channel techniques to assist penetration testers, digital forensics experts, and general network researchers for modern network measurements. The proposed techniques are novel, non-intrusive, and have high fidelity compared to previous side channel techniques.

In Chapter 3, I present an Internet measurement technique for finding machines that are hidden behind firewalls, a critical first step towards being able to measure modern networks. 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, it can still find the machine. The technique is based on the technique of Ensafi *et al.* [28], but does not require the SYN backlog to be filled to infer information, and SYNs are sent at a very low rate without causing denial-of-service. I also make no assumptions about globally incrementing IPIDs, as do idle scans. Thus I was able to carry out real measurements on the real Internet, as described in Chapter 3.

In Chapter 4, I present a technique to measure off-path round-trip time (RTT) with high fidelity. Round-trip time measurement is an important staple of Internet measurements and sees application in everything from IP geolocation to performance analysis. The ability to measure RTTs completely off-path, *i.e.*, to know the RTT between virtually any given machines A and B on the Internet without having any special access to A or B, would enable measurements in parts of the world where there is not good measurement infrastructure (such as Africa, South America, and other regions where there are not many machines to measure from such as PlanetLab [21] nodes and RIPE Atlas [5]).

In chapter 5, I describe a new technique to map out off-path trust relationships between two arbitary machines. Port scanning is a critical first step for penetration testers to understand network structure, in which a measurement machine sends probes to a target and, *e.g.*, determines if a given port is open or closed based on the received responses. The idle scan is a special kind of port scan that appears to come from a third machine that is not under the control of the penetration tester, called a zombie. Idle scans can be used to map out trust relationships in a firewall, such as a client that has a port open to only a backup server, or other information for planning attacks. Unfortunately, it assumes that the zombie has a discouraged behavior called a globally incrementing IP identifier (IPID), meaning that its usage is extremely limited and it is very rarely used in practice. The technique I propose is based on a much more advanced and prevalent IPID generation scheme, that of the Linux kernel. Although Linux's IPID generation scheme is specifically intended to reduce information flow, I show that using Linux machines as zombies to test off-path trust relationship is still possible. The technique has 87% accuracy, which is comparable to nmap's implementation of the idle scan at 86%. Its much broader choice of zombies will enable it to be a widely used technique which can fulfill various network measurement tasks.

Chapter 2

Related Work

TCP/IP side channels are a nascent area of research, but there has been a considerable amount of work. Antirez [12] first proposed the idle scan method in 1998. Morbitzer [57] explores idle scans in IPv6. Qian *et al.* [63,64] infer the TCP sequence number of a connection and perform off-path TCP/IP connection hijacking using a firewall-based side channel. Some work uses global IPID fields to perform inference for Internet measurement purposes. Chen *et al.* [20] 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 [16] describes a technique to detect NATs and count the number of hosts behind them. Kohno *et al.* [50] use the IPID to perform remote device fingerprinting. Knockel and Crandall [48] demonstrated that it was possible in a previous iteration of the Linux IPID generation algorithm to count packets sent to a specific destination by a remote server, and subsequently Cao *et al.* [19] demonstrated that related techniques can be used to interfere with connections completely off-path. Quach *et al.* [65] performed a comprehensive measurement of the impact of the ACK limiting vulnerability. The work of Gilad and Herzberg in this area is also notable. [35–37]

Some work uses SYN backlogs as side channels to perform the network measurement. Ensafi *et al.* [28] demonstrated that their SYN backlog TCP/IP side channel could be used to determine open *vs.* filtered ports on certain hosts, but the host-based firewall configurations that make this possible are not a common case. Follow-on work [26, 27] presented a hybrid method that combined the idle scan with the SYN backlog idle scan to improve the results and determine in which direction packets were being blocked by a firewall. The hybrid method still assumes that the zombie has a globally incrementing IPID, and is intended for global-scale measurements of national firewalls rather than port scans of local networks. Augur [60] is a technique to measure reachability between two Internet locations based on [27]. Zhang *et al.* [73] showed that the SYN backlog side channel can be used to find hidden machines behind firewalls, but did not attempt to make inferences about open ports. SYN backlogs are used to measure off-path network latency in both Alexander and Crandall [11] and Zhang *et al.* [74]. Spoofed return IP addresses and side channel inferences [27, 28, 63] have been shown to be very useful for Internet measurement, see, *e.g.*, Chen *et al.* [20]'s inferences based on IPIDs, reverse traceroute [47], or PoiRoot [44], or Flach *et al.* [30]'s use of spoofed IP addresses to locate Destination-Based Forwarding rule violations.

Once a set of firewall rules are known, there is a body of work to reason about those rules. Firmato [14], Fang [56] and Lumeta [71] are query-based firewall analysis systems. They interact with users on queries about firewall rules. Liu *et al.* [53] improves the query processing algorithm by using a tree representation. Other work [15, 40, 41] also focus on developing high-level specification languages to specify firewall rules. Some firewall design methods have been proposed [13, 29, 38, 42]. These works focus on detecting every pair of conflicting rules in a firewall. Gouda and Liu [38] use decision diagrams to reduce the size of a firewall's configuration. Firewall Policy Advisor [10] and FIREMAN [72] are techniques for detecting anomalies. FIREMAN [72] is implemented by using binary decision diagrams (BDDs) which can detect anomalies among multiple rules.

Port scanning is an active research area. Nmap's FTP bounce Attack [1] is able to make FTP servers port scan a target server. Modern FTP servers are configured by default to prevent this. Staniford *et al.* [68] and Gates *et al.* [34] focus on large enterprise network protection. Leckie and Kotagiri [51] use a probabilistic approach to detect port scans. Treurniet [69] aims to detect stealthy scans using classification schema. Muelder *et al.* [58] proposes a visualization for port scan detection. Jung *et al.* [45] develop a fast port scanning detection method using the theory of sequential hypothesis testing. Other work [18, 52, 67] use a neural network approach to detect malicious port scanning. Gates [32, 33] and Kange *et al.* [46] 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 [2, 6, 8, 9].

Traditional round-trip time (RTT) measurements use distributed servers to perform direct measurements. See, for example, IDMaps [31] or iPlane [55]. King [39] estimates off-path round trip time by using recursive DNS queries to DNS servers topologically close to each end point. Queen [70] uses a similar technique to measure packet loss rate. By contrast, the technique presented in Chapter 4 [74] and the technique it is based on [11], assume only that one machine is a Linux server with an open port and the other responds to SYN/ACKs with RSTs. Coordinate systems [22, 43, 59, 75] are another approach to estimating the RTT between two hosts, where the basic idea is for a given host to get a good estimate of latency to other hosts from itself for performance reasons. By contrast, my technique [74] can measure the RTT between two remote hosts that a scanner has no special control over, using TCP/IP side channels.

Chapter 3

Finding Machines Hidden Behind Firewalls

3.1 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 chapter we [1] propose a method for finding hidden machines behind firewalls, a critical first step towards being able to measure modern networks.

In 1998, Antirez [12] proposed the idle scan method. An idle scan is a port scanning technique that exploits TCP/IP side channels. In the idle scan, the measurement machine spoofs the return IP address of probes so that the scan appears

[1] I use “we” instead of “I” in this chapter and next chapter since they are published collaborative work.

to be coming from another machine. Side-channel information is then used by the measurement machine to infer how the target responded. Side channels are necessary because the *zombie*, which is the machine used as the return IP address of the probe, is not under the scanner's control. Thus, the network scanner has no direct way of knowing what packets the zombie receives from the target. 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. 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. We further discuss Antirez's idle scan Section 5.2.3.

In 2010, Ensafi *et al.* [28] 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 by sending SYN packets at a high rate, 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 [7, 17] 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 chapter, we describe our work on 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.

Our work 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.

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 this chapter is organized as follows. Section 3.2 gives some background information that is necessary for understanding our scan. Section 3.3 describes the implementation of our technique. We then describe our experimental methodology for assessing the effectiveness and applicability of our technique in Section 3.4, and how we perform quantitative analysis of the raw results in Section 3.5. Results are presented in Section 3.6, followed by discussion in Section 3.7 and the conclusion in Section 3.8.

3.2 Background

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

3.2.1 TCP basics

There are some rules that TCP follows [23], 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.

3.2.2 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.* [23] and Lyon [54].

TCP SYN scan

In a TCP SYN scan [23], 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.

SYN-ACK scan

In the TCP SYN-ACK scan [54], 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.

FIN scan

The FIN scan [23] 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.

Xmas Tree, Null scan

Xmas Tree and Null scanning [23] 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.

UDP scan and ICMP scan

The UDP scan [54] 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 [54] is implemented by sending an ICMP echo or timestamp request and waiting for the ICMP reply packet.

Fragment scan

In this chapter, 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.

3.3 Implementation

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

3.3.1 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.

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" [66]. This feature causes information flow before the resource is exhausted, so that we can make inferences without causing denial-of-service.

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.

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 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 3.1:

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.

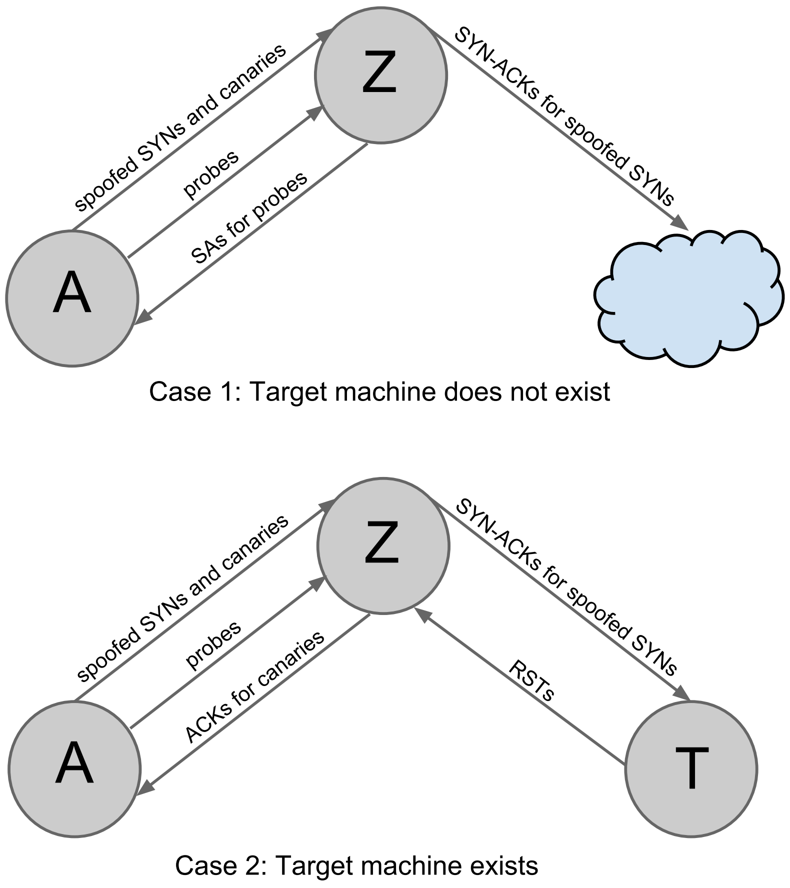
Z
A
spoofed SYNs and canaries
probes
SAs for probes
SYN-ACKs for spoofed SYNs
Case 1: Target machine does not exist
Z
A
T
spoofed SYNs and canaries
probes
ACKs for canaries
SYN-ACKs for spoofed SYNs
RSTs
Case 2: Target machine exists

Figure 3.1: Two cases in our scan.

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.

3.3.2 Nmap direct scan

We implemented Nmap's host discovery scan [3] *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.

3.3.3 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.

3.3.4 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.

3.4 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 3.3, 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) [4] 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 3.3.

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 3.3 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.

3.5 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

$$\begin{aligned} H_0 &: \text{the address was never alive during our test} \\ H_a &: \text{a machine at that address reset some SYN-ACKs.} \end{aligned}$$

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 3.4, 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 3.3, 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 chapter. 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.

3.6 Results

In this section we describe the results of our experiments.

3.6.1 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.

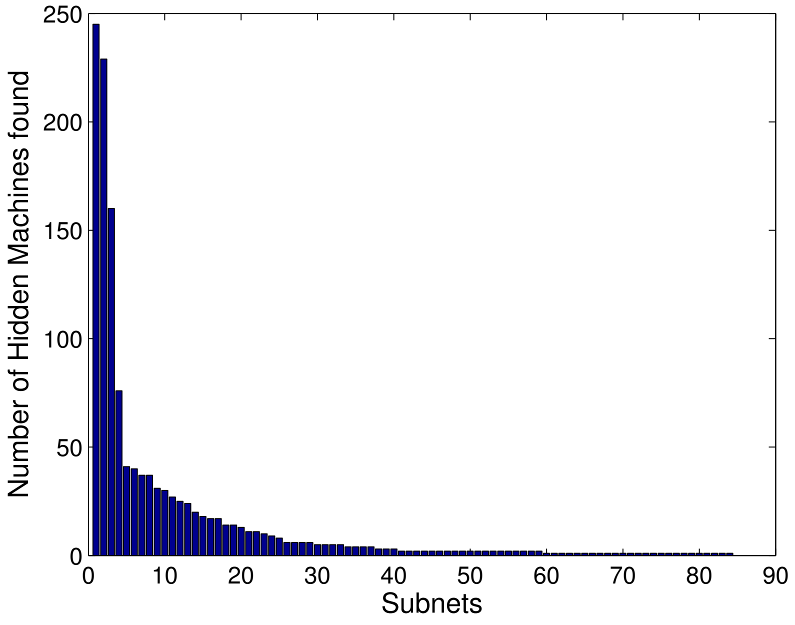

Figure 3.2: Distribution of the number of hidden machines per subnet.

3.6.2 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 3.2 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

Table 3.1: 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%

subnets (which ranges from 1 to 245). Most subnets (about 74%) had less than 10 hidden machines.

3.7 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.

3.7.1 Nmap *vs.* our direct 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 3.1 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%.

3.7.2 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.

3.7.3 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.

3.8 Summary of this Chapter and Future Work

In this chapter, 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.

Chapter 4

High Fidelity Off-Path Round-Trip Time Measurement

4.1 Introduction

In this chapter, we describe our work on improving existing off-path round trip time measurements *via* TCP/IP side channels.

Off-path round-trip time (RTT) measurement has many potential applications, including: improved geolocation capabilities, measuring the performance of parts of the Internet where there is not much measurement infrastructure (*e.g.*, PlanetLab), and providing data plane measurements to better understand global Internet routing. Off-path means that the measurement machine is not on the path being measured. More specifically, we can measure the RTT between essentially any two machines (A and B) on the Internet without having special access to A or B or having any presence in the path between A and B. Instead, we use packets with spoofed return IP addresses (*i.e.*, appearing to be from A to B or from B to A) and TCP/IP side channels that infer information about the state of either machine's network stack.

In 2002, Gummadi *et al.* [39] proposed King, a method to measure RTT off-path. Their technique was based on the DNS system and therefore was relatively limited in terms of what machines or networks it could measure between. In 2015, Alexander and Crandall [11] proposed a technique that is based on TCP/IP side channels and therefore can be applied between any Linux server and any client that responds to SYN-ACKs with RSTs. Thus, their technique is widely applicable across many parts of the Internet. However, the accuracy of their technique was greatly reduced when either the RTT being measured was low or the packet loss rate during the measurement was high. In this chapter, we propose an improved technique that overcomes both of these limitations.

Our new technique is shown to have 82.95% of the RTT measurement results within 10% of the actual RTT, and 91.18% of the results within 20% of the actual RTT; while the previous technique by Alexander and Crandall only had 60.7% of the results within 10% and 81.33% of the results within 20%.

We summarize our major contributions as follows:

1. We propose a significantly improved off-path TCP/IP side channel RTT measurement technique *via* active probing. Our technique is more robust to packet loss and more accurate across different RTT ranges compared to previous off-path RTT measurement techniques. Overall, 91.18% of our RTT measurement results are within 20% of the actual RTT, while the previous techniques of Alexander and Crandall [11] and Gummadi *et al.* [39] achieved 81.33% and 75%, respectively.

2. We perform a detailed analysis of sources of error across different RTT ranges when using our TCP/IP side channel off-path RTT measurement technique, and discuss certain challenges in our work as well as directions for future improvement.

4.2 Background

Next, we briefly outline TCP behavior specified by RFC 793 [61] relevant to our technique. Then we describe Linux's TCP/IP implementation and summarize the off-path measurement technique by Alexander and Crandall [11].

4.2.1 TCP behavior

Our technique utilizes the following TCP behavior:

1. When a client sends a SYN packet to a server, this creates a half-open connection in the `SYN_SENT` state on the client to the server.
2. When a SYN packet is received from a client on an open port on the server, it will create a half-open connection in the `SYN_RCVD` state on the server to the client, and a SYN-ACK will be sent in response.
3. A SYN-ACK packet received by a machine with no corresponding half-open connection will respond with a RST.
4. A RST received by a machine in response to a SYN-ACK will close the corresponding half-open (`SYN_RCVD`) connection.

4.2.2 Linux's TCP/IP implementation

On Linux, each listening socket has its own *SYN backlog*, a data structure that stores that socket's half-open (`SYN_RCVD`) connections. The SYN backlog can only store a finite number of entries, and its maximum capacity is determined by three variables:

1. The *backlog* argument of the listen() system call

2. The runtime kernel parameter *net.core.somaxconn*

3. The runtime kernel parameter *net.ipv4.tcp_max_syn_backlog*

We represent the above three variables by m_1, m_2, and m_3, respectively. Linux determines maximum backlog capacity m as

$$m = 2^{\lceil \log_2(\max(8,\min(m_1,m_2,m_3))+1)\rceil}.$$

Argument m_1 is large in most programs. Parameter m_2 is by default 128, and parameter m_3 is, by default, determined by the memory of the system but typically ≥ 128. Thus, m is commonly 256.

To compensate for packet loss, for each SYN still in the SYN backlog, Linux will send five SYN-ACK packet retransmissions, waiting $2^{i-1}t$ seconds after the previous transmission to send the ith retransmission, where t is the initial timeout for sending the first retransmission. On Linux kernels < 3.1, $t = 3$, and so these timeouts are 3, 6, 12, 24, and 48 seconds; and on Linux ≥ 3.1, $t = 1$, and so these timeouts are 1, 2, 4, 8, and 16 seconds.

To maintain service under high load or during a denial of service attack, Linux $\geq 2.3.41$ employs an additional mechanism to evict entries from the SYN backlog to prevent it from completely filling. Linux distinguishes between *young* entries, or entries whose SYN-ACKs have yet to be retransmitted, and *mature* entries, or entries whose SYN-ACKs have been retransmitted. Every 200ms, Linux checks if the backlog is at least half-full. If it is, it first determines a threshold number T as

$$T = \begin{cases} \max(2, 5 - \lfloor \log_2 \lfloor n/y \rfloor \rfloor) & \text{when } y > 0 \\ 2 & \text{otherwise} \end{cases}$$

where y is the number of young entries in the backlog and n is the number of entries currently stored in the backlog. It will then remove up to $\left\lfloor \frac{2m}{5t} \right\rfloor$ entries whose SYN-ACKs have been retransmitted at least T times, where t is again the timeout in seconds before retransmitting the first SYN-ACK and m is the size of the SYN backlog. The intuition here is that the more mature entries there are in the backlog, the more room Linux will make for young entries.

4.2.3 Previous work

Below we summarize the previous off-path measurement technique by Alexander and Crandall [11]. As mentioned earlier, this technique assumes that the server is a standard Linux machine with an open port and the client responses to unsolicited SYN-ACKs with RSTs. In general, to estimate the round-trip time between the server and the client using an off-path measurement machine, they use a binary search algorithm. In each round, a midpoint in the binary search is selected as a new round trip time estimate ($eRTT$). Then, a result about whether the $eRTT$ is too small or too large is obtained by running the below 3 steps.

1. Send SYN packets to the server, using the measurement machine's own IP address as the source IP address, without answering ACKs to complete the three way handshake. Each SYN packet uses different source ports to ensure that it is stored in a separate backlog entry. The total number of SYN packets sent is close to half of the backlog size in the server.

2. Wait until all the packets sent in Step 1 become *mature* (see Section 4.2), then send spoofed SYN packets to the server, using the client's IP address as the source IP address, at a specific rate determined as a function of the $eRTT$. The server, after receiving these packets, will create an entry for each spoofed SYN and then reply with SYN-ACKs to the client. The client machine will answer with RSTs to the server to reset those spoofed SYNs.

3. Infer how many of the SYN packets in Step 1 were evicted by counting the SYN-

ACK retransmission packets sent for them to the measurement machine. (If a SYN entry is evicted from the backlog prematurely, it will have made fewer transmissions.) Based on the number of evicted SYNs, the server's backlog status (whether half full or not) is inferred. The backlog status implies whether the *eRTT* is too small or too large.

4.3 Implementation

In this section, we describe the steps to perform our off-path RTT measurement.

4.3.1 Pre-requisite

Selecting clients

To perform off-path RTT measurement between a server and a candidate client machine, we first need to determine whether the client machine meets the assumptions of our experiment. As laid out in Section 4.2.1, we require that our client machines reply to unsolicited SYN-ACKs with RSTs as per RFC 793 [61]. To determine this behavior, we simply send SYN-ACKs to a candidate client machine using the server's open port as the source port (but with our measurement machine's address as the source address) to mimic as closely as possible the SYN-ACKs that will be sent from server during the experiment. If we receive RSTs in response, we consider our technique applicable to the machine.

Determining SYN backlog size of a Server

Determining a Server's SYN backlog size is another critical step before the off-path RTT could be measured. We implement the technique as described in Section 3.3.

The advantage of active probing, comparing to the SYN-ACK retransmission counting method used in [11], is that it is not influenced by SYN-ACK packet loss. In the previous technique, if one of the five retransmitted SYN-ACKs accidentally gets lost, the backlog would be falsely considered more than half full. While if duplicate SYNs get lost, we can still resend them and make sure to get the status of backlog.

4.3.2 Evaluating an estimated RTT

Now we will set out how we utilize the server's SYN backlog as a side channel and use it to measure the RTT between the server and the client. First, we will show three steps to evaluate whether the RTT between a server and a client is less than or greater than an estimated RTT. Then we will show how to use this evaluation to measure the actual RTT.

In the first step, we fill the server's backlog with SYNs (with the measurement machine's return IP address) to use as a side channel for testing an estimated RTT between the server and the client. For simplicity, we call these SYNs *canaries.* We nearly half-fill the backlog with $c = m/2 - (\log_2(m) - 2)$ canaries. We subtract out a $(\log_2(m) - 2)$ term as a buffer to make room for any connection requests from real clients. We want this term to be at least two in order to be as accommodating as possible no matter the size of the backlog, but we do not want the subtracted term to be too large in order to minimize packet rates in the next step. Each of the c canaries has a different source port and is sent three times to ameliorate possible packet loss. We send canary packets at 100 packets per second (pps).

In the second step, we wait to ensure the number of SYN-ACK retransmissions for each canary is at least 2. Then we send spoofed SYNs to the server, using the client machine's IP address as the source address, at a rate of $3(c-1)/eRTT$ packets per second for ten seconds. Each spoofed SYN has a unique source port and sequence number. We multiply the rate by three because each packet is sent three times to ameliorate packet loss. The server will reply to each SYN with a SYN-ACK to each client machine. The client machine, after receiving from the server each unsolicited SYN-ACK, will respond with a RST to the server. These RSTs will remove their corresponding entries in the backlog. The time it takes for a RST to remove each spoofed SYN from the backlog is the RTT between the server and the client machine. If more than $(\log_2(m) - 2)$ spoofed SYNs arrive before the first RST from the client machine arrives, then the server's backlog will become more than half full and Linux will begin evicting mature entries.

In the final step, instead of counting SYN-ACK retransmissions for each canary as in [11], we send "duplicate SYNs" that we call *probes* to test for the canaries' presence in the backlog. These probes have the same TCP header values as the original canaries, except each of their sequence numbers is the sequence number of their corresponding canary minus one. Each probe is sent three times to ameliorate possible packet loss. We send probe packets at the same rate as in step one. We have two possible results, as illustrated in Figure 4.1:

1. In the case that, for every canary, we received ACKs in response to at least one of the probes sent to test its presence in the backlog, none of the canaries were evicted, and thus we conclude that the server's backlog was never at least half full because each spoofed SYN has been reset before up to $(\log_2(m) - 2)$ spoofed SYNs arrived. Therefore, the chosen estimated RTT is larger than the actual RTT.

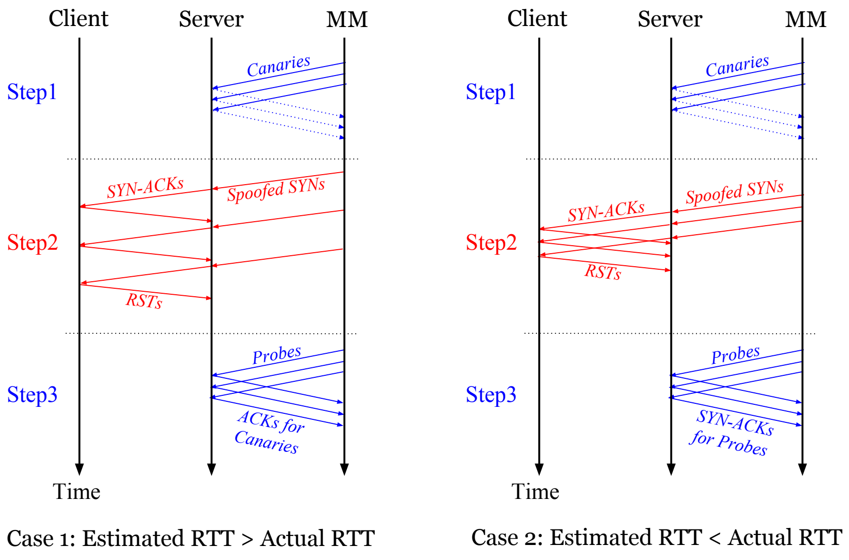

Figure 4.1: Two cases in our estimated RTT evaluation.

2. In the case that at least one canary's probes only responded with SYN-ACKs, at least one canary was evicted, and so the server's backlog was at some time at least half full because up to $(\log_2(m) - 2)$ new spoofed SYNs arrived to the server's backlog before the previous had been reset. Therefore, the chosen estimated RTT is less than the actual RTT.

4.3.3 Determining the RTT

We will now show how to use our technique to evaluate an estimated RTT to determine the actual RTT between a server and a client. By performing binary search over the estimated RTT [11], we can converge on the actual RTT. However, during our experiments, we found that the actual RTT may vary. The problem of using binary search is that once a bisection decision is made, it cannot be revisited. If a wrong decision is made in early stage, the influence can be severe.

We replace the binary search with a heuristic search algorithm so that it can backtrack. Instead of testing the midpoint between the lower and upper bounds as an estimated RTT, we perform a ternary search such that we test two estimated RTTs, each evenly dividing the remaining search space into thirds. We also use two stacks to keep track of the trace points on both sides. For the left (lower) point, if the estimation is too small, the left point will be kept; if the estimation is too big, the previous left point will be returned as a new left point. The case for the right (upper) point is similar. This way we are able to backtrack when experiencing variations of actual RTT. We keep iterating this algorithm until the distance between left point and right point is less than 5ms.

4.4 Experimental Setup

We duplicated the experimental setup used by Alexander and Crandall in order to make a direct comparison with their results. The measurement machine we used ran Ubuntu Server 14.04 with Linux kernel 3.13 installed. It was directly connected to an Internet backbone without any stateful firewall or egress filtering in between. We selected 15 PlanetLab nodes as our servers, evenly distributed in Asia, North America, South America, Europe, Australia, three nodes for each continent. We used PlanetLab nodes as servers in order to directly compare our off-path RTT measurement results with on-path RTTs recorded on the servers themselves; however, we did not use the servers to assist in our off-path RTT measurement technique–they are only used to verify results. The PlanetLab servers that we selected were all running Linux with kernel 2.6.x. We opened an unprivileged TCP port 15216 on every Planet lab node and confirmed that every backlog size was 256 before running our off-path RTT scan.

Once an off-path RTT measurement began, the measurement machine first copied necessary scripts to every PlanetLab server. We used these scripts to record on-path RTTs which serve as a ground truth to verify our off-path technique results. Then our measurement machine sent a remote ssh command to activate the server program to listen on port 15216 on each PlanetLab server. We used an unprivileged port 15216 to help setup tcpdump filters and also to avoid conflicts with other PlanetLab users. After that, the measurement machine created multiple threads, each using one of our PlanetLab servers to perform our off-path measurement.

Each thread first tested if a randomly generated IPv4 address replied to SYN-ACKs with RSTs. Then it activated previously copied scripts on its corresponding PlanetLab server to:

1. Start a tcpdump to record traffic between the server and the client

2. Run traceroute to the client during the experiment

The tcpdump output was used to capture the SYN-ACKs and RSTs between the server and the client. This traffic was created by our off-path technique, but the purpose of keeping it was to directly compare with our off-path result. The traceroutes were used to verify if there were any routing changes during our off-path RTT measurement. After all the environmental setup was done, we executed our off-path experiment. We used a search algorithm discussed in section 4.3 to search the space between 0 and 3000ms. When a result RTT range was found, ssh commands were sent to terminate tcpdump and any running traceroute. Each off-path testing took about an hour to complete. We used tcpdump output to calculate an average RTT between the server and the client during our off-path measurement, and compared it with the mean of our off-path RTT result range.

Dataset	Within 10%	Within 20%
Overall	82.95% (89.74%)	91.18% (96.92%)
RTT > 25ms	83.36% (90.6%)	91.39% (97.39%)
RTT < 25ms	42.86% (42.86%)	71.43% (71.43%)
RTT > 100ms	86.05% (93.15%)	92.77% (97.82%)
RTT < 100ms	63.92% (73.91%)	81.44% (92.75%)
25ms< RTT < 100ms	65.56% (77.42%)	82.22% (95.16%)

Table 4.1: Percent of measurements within given percent of actual round trip time. Values for measurements with no packet loss are in parentheses.

Dataset	Within 10%	Within 20%
Overall	82.95% (60.7%)	91.18% (81.33%)
RTT > 25ms	83.36% (63.6%)	91.39% (83.7%)
RTT < 25ms	42.86% (18.0%)	71.43% (46.1%)
RTT > 100ms	86.05% (67.1%)	92.77% (87.2%)
RTT < 100ms	63.92% (35.5%)	81.44% (58.06%)
25ms< RTT < 100ms	65.56% (43.5%)	82.22% (63.5%)

Table 4.2: Percent of measurements within given percent of actual round trip time. Previous measurement results are in parentheses.

4.5 Results

We ran off-path RTT measurement using a single measurement machine from 1 October 2015 to 6 October 2015. 728 round trip time estimates were collected during this period. We determined by tcpdump output that 36 (4.5%) experiments showed no RST traffic from the client to the server, so we excluded these experiments from our analysis, leaving 692 data points. Using the method we discussed in Section 4.3, we calculated the corresponding on-path RTT for each of our off-path RTT estimates. Figure 4.2 shows the CDF plot for the estimated RTTs divided by actual RTTs.

Our technique has higher accuracy than previous off-path RTT measurement techniques. 82.95% of RTT measurement results are within 10% of the actual RTT,

Dataset	Within 10%	Within 20%
Overall	89.74% (68.87%)	96.92% (90.84%)
RTT $>$ 25ms	90.6% (72.1%)	97.39% (92.7%)
RTT $<$ 25ms	42.86% (16%)	71.43% (60.0%)
RTT $>$ 100ms	93.15% (75.9%)	97.82% (96.0%)
RTT $<$ 100ms	73.91% (39.3%)	92.75% (69.05%)
25ms$<$ RTT $<$ 100ms	77.42% (49.3%)	95.16% (72.9%)

Table 4.3: Percent of measurements within given percent of actual round trip time with no packet loss. Previous measurement results are in parentheses.

and 91.18% of the results are within 20% of the actual RTT. In contrast, the previous technique by Alexander and Crandall has 616 round-trip time estimates, 60% of them within 10% and 81.33% of the results within 20%. Table 4.2 and Table 4.3 show a direct comparison of our result to the previous technique developed by Alexander and Crandall. King, another off-path latency estimation tool that used DNS, has less than 20% of error in over three quarters (75%) of estimates and 10% of error in two-thirds (67%) of the estimates. Below we investigate the accuracy of our technique as well as analyzing the sources of error.

4.5.1 Performance for low RTTs and high RTTs

Figure 4.3 shows the accuracy of our estimates for actual RTTs less than 25ms. In this RTT range, our measurement has 42.86% within 10% of actual RTTs, and 71.43% within 20% of actual RTTs. Comparing with previous results, we have a 24.86% and 25.33% increase of accuracy, respectively. Our estimates for small RTTs are bounded by 50% of actual RTTs, within an error less than 7.77ms. Meanwhile, our results show the capability of measuring off-path RTTs accurately even for a RTT less than 10ms, *e.g.*, the smallest RTT we measured had actual RTT 7.16ms, and our estimate was 7.01ms. We found that 85.7% of our low RTT results overestimates the actual

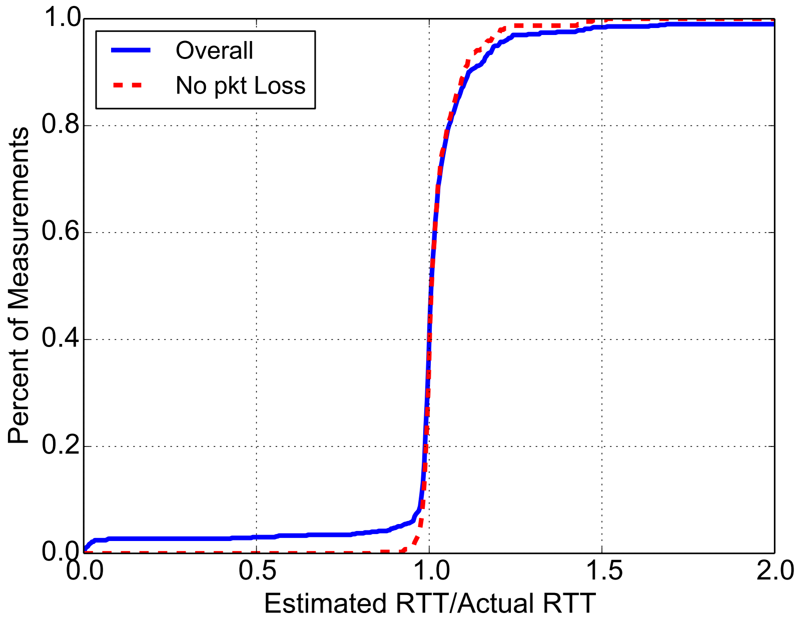

Figure 4.2: Overall accuracy of Off-Path RTT estimates.

RTTs. The reason for this is that the packet intervals in low RTT estimates are usually small, and any small delay of RSTs (such as a packet loss of the SYN-ACK or RST) may cause the server's backlog to become more than half full unexpectedly. As a result, the search algorithm will falsely believe that the estimate is too small, and try a larger value in the next round.

Our technique performs better when the actual RTTs are greater than 100ms. Among our RTT estimates, 86.05% are within 10% and 92.77% are within 20%, compared to the previous results of 67.1% within 10% and 87.2% within 20%. Figure 4.4 shows the accuracy of our estimates for this RTT range. One major source of error in our results is from the variations of actual RTTs. Large RTTs typically have more variations than small RTTs. We found that for the 7.23% of results that are beyond 20% of actual RTTs, 58.14% of them have a standard deviation of RTTs 25ms or larger; while for the 92.77% results that are within 20%, only 11.96% of

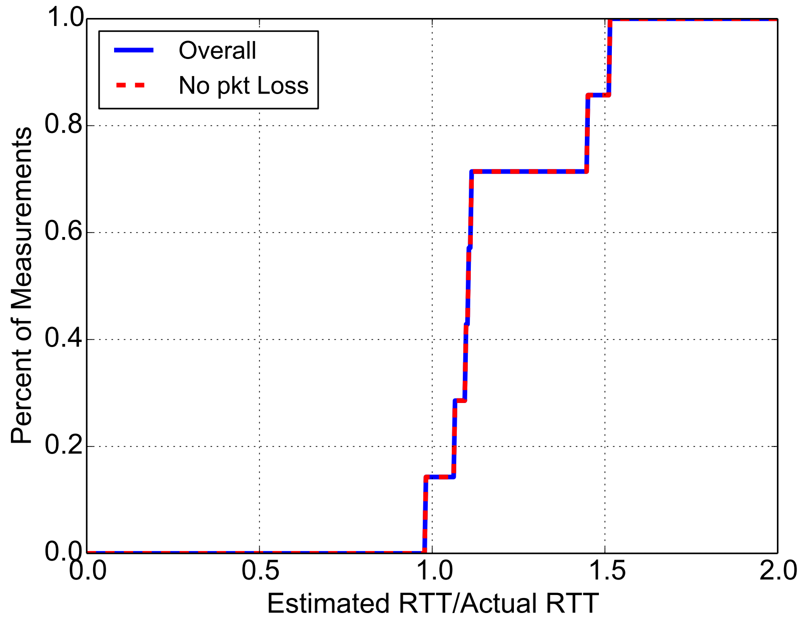

Figure 4.3: Effects of packet loss on RTT estimates (actual RTT <25ms).

them have a standard deviation of RTTs 25ms or larger. For cases when a standard deviation of RTTs is 250ms or more, only 10% of results are within 20%; while for cases when a standard deviation of RTTs 25ms or less, 96.43% of results are within 20%. One possible reason for large variation in RTTs is route change. We found in our data that, although route changes existed in many of our experiments, it happened most frequently for large RTTs.

4.5.2 Effects of packet loss

We used active probing to ameliorate the influence of packet loss between the measurement machine and server, as previously discussed. For the purpose of this chapter, we are interested in packet loss between the server and client. Figures 4.2, 4.3, 4.4, 4.5 show a comparison between the full data set case and the case without packet

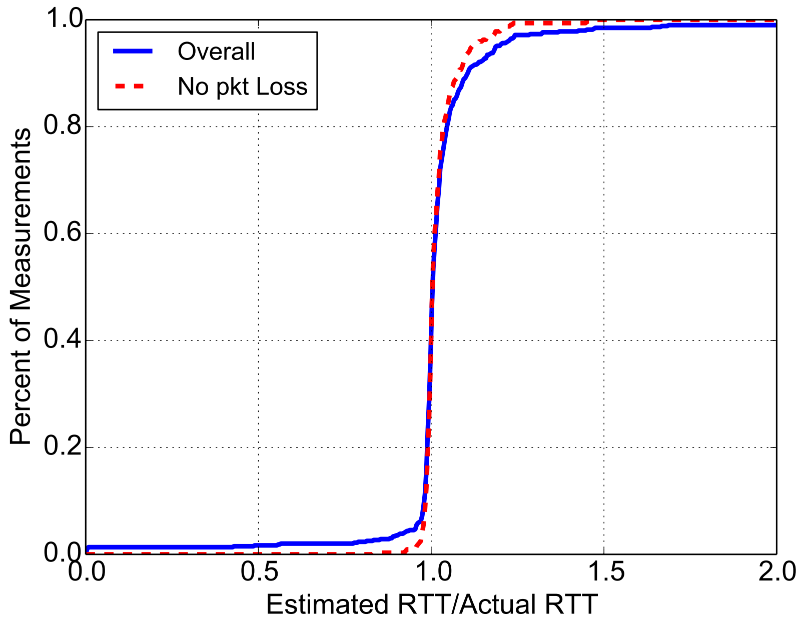

Figure 4.4: Effects of packet loss on RTT estimates (actual RTT >100ms).

loss. The sample for our low RTT estimates is small and did not experience packet loss, so in Figure 4.3, the no packet loss case shows the exact same pattern as a normal case. In other RTT ranges, the no packet loss case had an increase of accuracy from about 5% to 12%. To understand how packet loss between server and client influences our results, we consider a case where the server sent SYN-ACKs and the RSTs from client are lost. As a result of that, the server's backlog is more than half full and it evicts canaries. From the measurement machine's point of view, the estimate is less than the actual RTT.

4.6 Discussion

For cases where actual RTTs are greater than 25ms and less than 100ms, our measurement performs better than the case of small RTTs, with 65.56% within 10% and

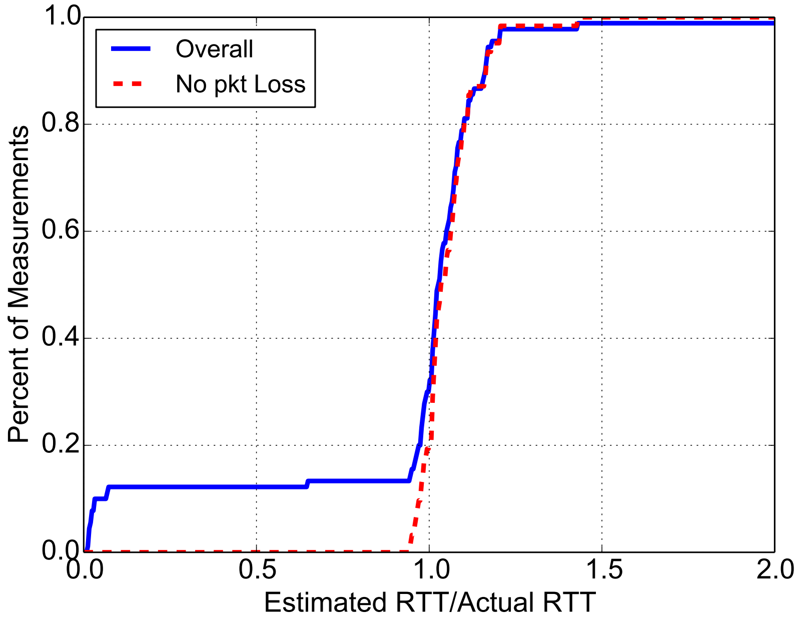

Figure 4.5: Effects of packet loss on RTT estimates (25ms <actual RTT <100ms).

82.22% within 20%, as shown in Figure 5. For comparison, Alexander and Crandall had 43.5% within 10% and 63.5% within 20%. Note that about 10% of our results in this range underestimate the actual RTTs. The reason for this is that the backlog was completely filled by the first arrival of spoofed SYNs and refused any newly arriving spoofed SYNs. At the same time, RSTs from the client reset the spoofed SYNs currently in the backlog before the server's eviction happened. That is to say, this problem happens only when 1) The search algorithm makes an aggressive move to the left, causing the server's backlog to quickly fill; and, 2) The actual RTT is less than 200ms (the SYN-ACK timer), which means RSTs could reset spoofed SYNs before eviction happens. For large RTTs, the second requirement cannot be met because RSTs come back after the server starts the eviction; for small RTTs, the first requirement cannot be met because RSTs come back quickly enough and the server's backlog will not be full. Therefore, we saw these kinds of results in this

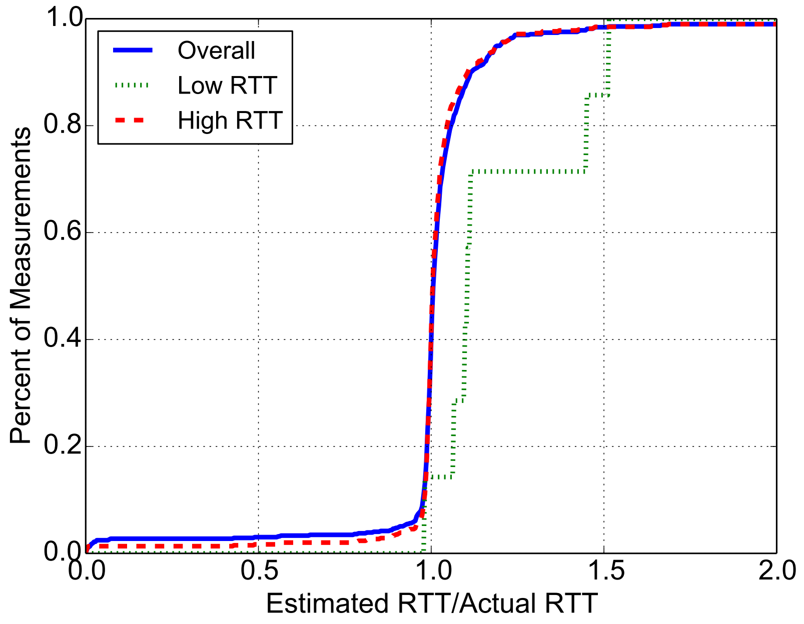

Figure 4.6: Performance comparison between different RTT ranges.

range.

Our technique ameliorates packet loss by sending each packet three times. However, the two additional SYNs should not cause extra resource allocations since together they create at most one connection on the server. Each SYN packet's size is 60 bytes, and the network burst of traffic created by our scan is less than 150 kilobytes per second per server. During each round of the scan, the server's SYN backlog could only possibly be more than half full (but not completely full) for less than 200ms, and so it should not cause denial of service. However, in some rare cases, as we discussed above, the server's backlog was full due to the search algorithm making an aggressive move. To improve our technique, the search algorithm needs to be less aggressive when testing RTTs less than 100ms.

4.7 Summary of this Chapter and Future Work

We have presented an improved method for off-path RTT measurement that is more robust to packet loss and more accurate for low RTTs than Alexander and Crandall's method. Opportunities for improvement to the technique include accounting for Linux's SYN backlog pruning timing, adapting the technique to a broader set of server OSes, and decreasing the packet rate so that the traffic is not perceived as invasive. Our results presented in this chapter show that managing SYN backlog entries can enable significant improvements in accuracy over Alexander and Crandall's technique.

Chapter 5

ONIS: Inferring TCP/IP-based Trust Relationships Completely Off-Path

5.1 Introduction

We briefly mentioned the idle scan in Section 3.1. The idle scan, although originally intended for learning the status of a port, in general can be used to learn the trust relationship between two arbitrary hosts that a network researcher does not control. For example, consider a network researcher in country X who wants to learn if network traffic from a host in country Y can connect to a Tor server in country Z. Performing this measurement off-path is necessary when vantage points (VPNs, Planet Lab nodes, *etc.*) are limited or unavailable in some countries. Ensafi *et al.* detail this off-path trust relationship testing by using the idle scan in [27]. Specifically, they measured packet drops from clients to Tor directory servers by using machines with global incrementing IPIDs as vantage points without those machines being under their control.

Unfortunately, use of the idle scan has two major issues.

1. It requires that the zombie has a globally incrementing IPID. Many modern network stacks are specifically designed to prevent information flow through IPIDs. One of the most advanced network stacks in this respect is Linux. The Linux IPID generation algorithm is described in Section 5.2.4.

2. It also assumes that the zombie is idle, hence the name "idle scan". Internet-connected hosts are seldom idle. Ensafi *et al.* proposed using an autoregressive moving average (ARMA) model to handle the noise on zombie machine in [27], but sometimes the process of fitting an ARMA model to the data fails if the zombie machine is often not idle (*e.g.*, web servers).

Motivated by the goal of overcoming both of these drawbacks of the idle scan, in this chapter we propose **ONIS**: **O**NIS is **N**ot an **I**dle **S**can, which uses an up-to-date Linux machine as the zombie. ONIS extends the choices of zombies of the idle scan, by using up-to-date linux machines as zombies. According to our estimate in Section 5.6, 17% of web servers are potential zombies that can be used. Unlike an idle scan, ONIS does not require the zombie to be completely idle. Although ingress filtering prevents our scan by not allowing packets with spoofed IP addresses into a network, only 23.9% of the networks on the Internet actually perform this [73].

We summarize our major contributions as follows:

1. We propose ONIS: a novel indirect scanning technique using TCP/IP side channels. The idle scan requires a global incrementing IPID zombie machine, which has been gradually phased out in many major OSes. ONIS uses Linux machines with kernel 3.16 or later as zombies and does not require the zombie

to be idle. ONIS achieves 87% accuracy, which is roughly as accurate as the idle scan at 86%. In the meantime, it allows a much broader choice of zombies.

2. We propose a new technique to do IPv4/IPv6 alias resolution on Linux machines with kernel 3.16 or later.

3. We perform a detailed analysis of noise in our scan technique and propose an effective model selection method to handle noise.

The rest of this chapter is structured as follows: Section 5.2 gives a review of what an IPID is and how Linux generates IPIDs. Section 5.3 talks about the methodology of ONIS, as well as a new technique to perform IPv4 and IPv6 alias resolution and a model selection method called "AIC". Section 5.4 describes the details of our experimental setup. We provide a direct comparison of results between ONIS and nmap implementation of the idle scan in Section 5.5. In Section 5.6 we discuss the applicability of ONIS, ethics concerns and possible defenses against ONIS. We present our conclusion in Section 5.7.

5.2 Background

In this section, we briefly review IP identifiers and then discuss Linux's changing approach to generating them in response to different attacks.

5.2.1 IP Identifiers

Every IPv4 packet contains a 16-bit field known an *IP identifier* (IPID). When an IP datagram is too large to be transmitted over a link, a router can break it up into smaller packets called *fragments*. The datagram's final destination can reassemble the original datagram by collecting each incoming fragment until all fragments have been received. The final destination uses each fragments' IPID to determine which datagram it belongs to.

IPv6 is different from IPv4 in this matter in that every IPv6 packet does not necessarily have an IPID. If fragmentation of a datagram is required for it to reach its final destination, then the original sender fragments the datagram, adding to each fragment an IPv6 extension header containing a 32-bit identifier, a field 16 bits larger than an IPv4 packet's IPID.

5.2.2 Early Linux IPID generation

The Linux kernel originally determined each IPv4 datagram's IPID by using a globally incrementing counter. Every time a host sends a datagram, the value of the counter is incremented (mod 2^{16}) and then used as that datagram's IPID.

In 1998, a technique called an *idle scan* was discovered to port scan machines off-path by exploiting globally incrementing counters as a side-channel [12]. In response, kernel developers switched to having a separate counter for each IP destination.

5.2.3 The idle scan

A measurement machine can use the idle scan technique to port scan a target machine completely off-path by performing the procedure described in this section.

Before the scan, the measurement machine identifies a suitable *zombie* machine with the following characteristics: the measurement machine can communicate with the zombie, the zombie can communicate with the target, the zombie responds to SYN-ACKs with RSTs, the zombie has a globally incrementing IPID counter, and its network communication is *idle*, *i.e.*, aside from the scan, it is not otherwise sending any datagrams.

The measurement machine first probes the current value of the zombie's IPID counter by sending it a SYN-ACK packet. The zombie responds with a RST packet with the current value of its IPID counter. Next, the measurement machine sends a SYN packet to the target with the source address spoofed to be that of the zombie. If the destination port number of the spoofed SYN packet is open on the target, the target will send a SYN-ACK to the zombie, and the zombie, not expecting the SYN-ACK since it did not send the spoofed SYN, sends a RST to the target, incrementing the zombie's IPID counter. Otherwise, if the port on the target is closed, the target sends a RST to the zombie, which does not cause the zombie to send any packets, and so the zombie's IPID counter is unaffected. Finally, the measurement machine sends another SYN-ACK probe to the zombie to once again measure the current value of its IPID counter. If the IPID of the responding RST packet is one greater (mod 2^{16}) than that of the last probe, then the destination port of the spoofed SYN must be closed on the target. However, if the IPID of the RST is two greater (mod 2^{16}) than that of the original probe, then the port is open, since the SYN-ACK the target sent to the zombie incremented its counter in between the probes.

5.2.4 Recent Linux IPID generation

In 2014, in Linux 3.16, the kernel developers recognized performance issues with using a separate counter for each IP destination [24]. Since having a globally incrementing counter was still undesirable, they adopted a hybrid approach consisting of 2048 globally incrementing counters. To determine which counter to use for an IP datagram, that datagram's destination address is hashed with a secret value randomly generated at system startup. The resulting hash (mod 2^{11}) is used to determine the index of the counter. Each counter is 32 bits to accommodate IPv6, and for IPv4 the IPID is taken from only the lower 16 bits of the counter.

During the same time, a side-channel technique was discovered that could count the number of datagrams sent between machines off-path [48]. The technique worked by inferring the values of per-destination IPID counters off-path but could also be extended to work for the kernel's new hybrid approach [25]. In response, the kernel developers made additional changes to make each counter less predictable. Every time a counter is used to assign an IPID, instead of incrementing it by one, the kernel adds to it a number uniformly distributed between 1 and the number of system ticks since the counter was last used.

In light of defending against machines being used as zombies in the idle scan, this new hybrid approach was identified as partially having the problems of per-destination counters and partially having the problems of a globally incrementing counters [49]. It has the problems of per-destination counters in that the counters would still partially isolate information about which hosts a zombie is sending packets to, since the probability of any two destination machines hashing to the same counter is only 1/2048. This means that a zombie need not necessarily be completely idle, only the counter that it uses to send packets to the target need be idle. Moreover, it partially has the problems of globally incrementing IPID counters in that, if a measurement machine has an address hashed to the same counter on the zombie as that of the target, then that counter shares information between the target and the measurement machine the same way a single globally incrementing counter would. However, the probability of this occurring between any one measurement machine address and any one target machine address is only 1/2048.

The effort to make IPID counters less predictable by adding a value uniformly chosen at random also makes the idle scan more difficult to perform due to the random noise being added. The addition of this random noise results in the number of datagrams the zombie sends no longer significantly affecting the expected value of the observed increase to its IPID counter. However, it has been noted [49] that, although the expected value does not significantly change, the number of packets sent by the zombie still changes the distribution by which the IPID counter is observed to have increased. This is because, if $\mathcal{U}(a, b)$ is a discrete uniformly distributed observation between a and b, then $\mathcal{U}(1, n)$, the observed increase when no datagram from some counter has been sent in between IPID probes received n system ticks apart, has a different distribution than $\mathcal{U}(1, n/2) + \mathcal{U}(1, n/2)$, which would be the approximate observed increase if one datagram were sent exactly in between the two IPID probes having been received. Note that for large k and n, $\sum_{i=1}^{k} \mathcal{U}(1, n/k)$ approximates a normal distribution due to the central limit theorem.

5.3 Implementation

In this section, we describe how ONIS works by using Linux machines with kernel 3.16 or later as zombies. We start by describing a general approach to perform ONIS. Then we propose a new technique to do IPv4 and IPv6 alias resolution on Linux machines with kernel 3.16 or later. Next, we introduce our implementation of ONIS which uses dual-stack Linux machines as zombies. Finally, we talk about how to process the result by using a model selection technique called “Akaike information criterion” (AIC).

5.3.1 Overview of ONIS

ONIS requires that the zombie machine is running Linux with kernel version 3.16 or later and replies to unsolicited SYN-ACKs with RSTs as per RFC 793 [61]. Similar to the idle scan described in Section 5.2.3, there are also three steps for ONIS.

In the first step, the measurement machine sends a SYN-ACK packet to the zom-

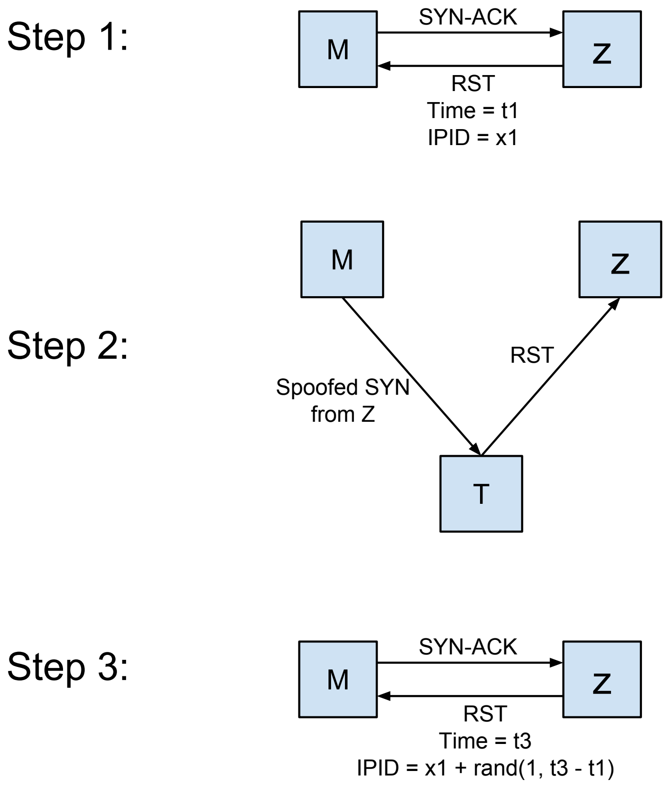

Figure 5.1: Scan of closed port on the target using ONIS.

bie machine, using the measurement machine's IP address as its source IP address. According to RFC 793 [61], the zombie will reply with a RST packet back to the measurement machine, since the zombie did not send any SYN. Let the IPID in the RST packet be x_1 and the time in system ticks when the zombie generates x_1 be t_1. After receiving the RST packet from the zombie, the measurement machine records x_1.

In the second step, the measurement machine sends a spoofed SYN packet to a port of the target machine using the zombie machine's IP address as the source IP address. Depending on the status of the port (open, closed/filtered), the target will

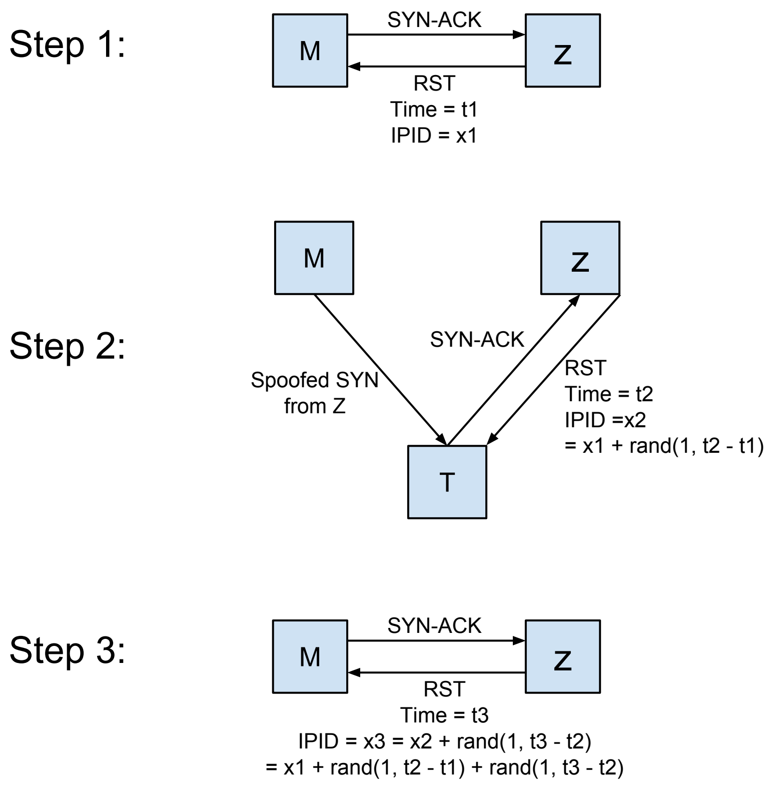

Figure 5.2: Scan of an open port on the target using ONIS.

respond differently.

1. In the case that the port on the target is closed (see Figure 5.1), the target will reply with a RST packet to the zombie. The zombie will simply ignore the RST and not send any packets in response. For a filtered port in the target, the SYN packet spoofed from the zombie is silently filtered and thus there is no traffic between the zombie and the target. The result is the same as a closed port scenario, since the zombie will not generate any new IPIDs.

2. In the case that the port on the target is open (see Figure 5.2), the target will reply with a SYN-ACK packet to the zombie. The zombie, which did not send

any SYN to the target, will send a RST packet to reset the handshake. Let the IPID in this RST packet be x_2 and the time in system ticks it was generated at be t_2. Here we assume that x_2 draws from the same Linux IPID counter as x_1, as we will later show how to ensure this, which we discuss in Section 5.3.2. Then, according to the Linux behavior of generating IPIDs (see Section 5.2.4), we have $x_2 = x_1 + \mathcal{U}(1, t_2 - t_1)$, where $\mathcal{U}(a, b)$ is a discrete uniformly distributed random variable as before.

In the third step, similarly to step one, the measurement machine sends a SYN-ACK to the zombie to collect x_3, the IPID in the following RST. Let the time in system ticks when the zombie generates x_3 be t_3. Then the distribution of x_3 will differ according to the status of the port of the target.

1. If the port is closed, the zombie only generates an IPID in the first and third steps, thus yielding $x_3 = x_1 + \mathcal{U}(1, t_3 - t_1)$.
2. If the port is open, as seen in step 2, the zombie has an additional access to the IPID counter in which case $x_3 = x_2 + \mathcal{U}(1, t_3 - t_2) = x_1 + \mathcal{U}(1, t_2 - t_1) + \mathcal{U}(1, t_3 - t_2)$.

Whether x_2 is generated is not directly known to the measurement machine. However, from repeated measures of the values of x_1 and x_3, it is possible to infer the status of the port on the target by analyzing the distribution of their differences, $x_3 - x_1$. Figure 5.3 shows the contrast of two random number distributions generated by $\mathcal{U}(1, 2n)$ and $\mathcal{U}(1, n) + \mathcal{U}(1, n)$ where $n = 50$.

5.3.2 Finding dual-stack Linux machines

As described in Section 5.2.4, Linux 3.16 or later uses 2048 global counters to generate IPIDs. The scan method we talked about in the previous section relies on the fact

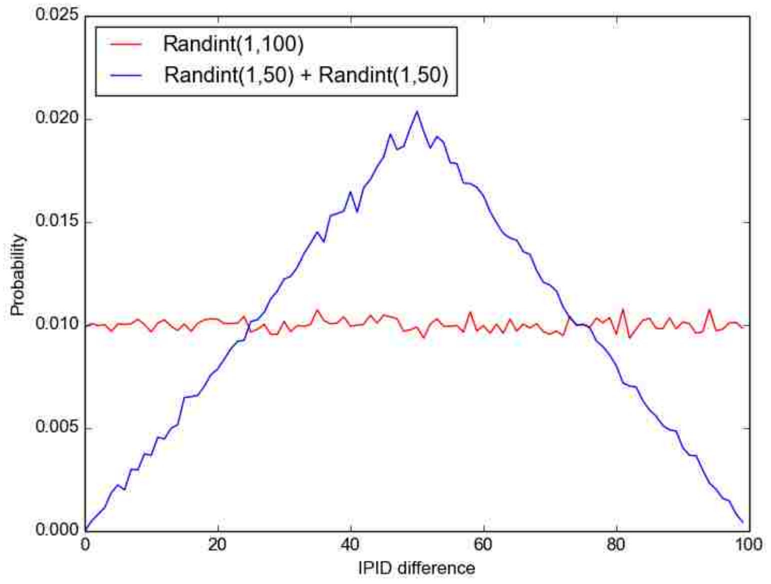

Figure 5.3: Distributions of observed IPID increases in two cases of ONIS.

that the measurement machine has an address that hashes to the same IPID counter on the zombie as that of the target. It is possible to try differing measurement machine addresses until a collision is found with the target's IP address. Each time, we have a possibility of 1/2048 for the hashes to collide. If we have 10,000 IP addresses to try, the chance to have a collision of a certain IP address at least once is more than 99%. $(1 - (\frac{2048-1}{2048})^{10000} \approx 99.2\%)$

Such resources are usually within the capabilities of network researchers, especially considering how easy to obtain a /64 of IPv6 addresses nowadays (a /114 of IPv6 addresses would be sufficient). In our experiment, we demonstrate how ONIS works by using multiple IPv6 addresses in our measurement machine since the same 2048 IPID global counters are used by both IPv4 and IPv6.

Now we present a new technique to do IPv4 and IPv6 alias resolution on a Linux machine with kernel 3.16 or later. Previous alias resolution techniques are either

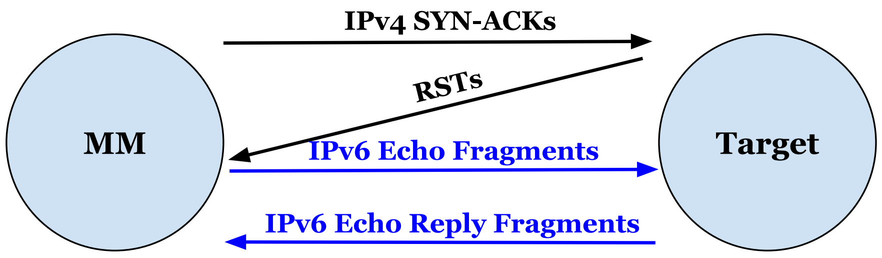

Figure 5.4: IPv4 and IPv6 alias resolution.

IPv4 or IPv6 only, and so our alias resolution technique is novel. We use a TCP/IP side channel discussed above in Section 5.2.4 to achieve this, as shown in Figure 5.4. Here we call the machine we perform IPv4 and IPv6 alias resolution on the “target”, although for ONIS this target will become the zombie.

Given an IPv4 address and an IPv6 address, in each round we simultaneously send an IPv4 SYN-ACK packet and a large IPv6 Echo Request to the target to collect its IPv4 IPID and IPv6 fragment ID. We fill the IPv6 Echo Request’s body such that the unfragmented size of the datagram is 2000 bytes and so the subsequent reply will require fragmentation to reach the measurement machine, ensuring that it will contain an IPv6 extension header containing an IPv6 fragment ID. Then, we vary the source IPv6 address with another, and resend the same type of packets.

If an IPv4 address (a_4) and an IPv6 address (a_6) are aliases of the same Linux 3.16 or later machine, then it is possible to find a measurement machine IPv6 address that hashes to the same IPID counter receiving probe replies from a_6 as our measurement machine’s IPv4 address does receiving probe replies from a_4. We test 10,000 different measurement machine IPv6 addresses. For each measurement machine IPv6 address we would like to test, we generate ten IPv4 SYN-ACK probes and ten large, 2000 byte IPv6 Echo Requests, large enough so that the probed machine will need to

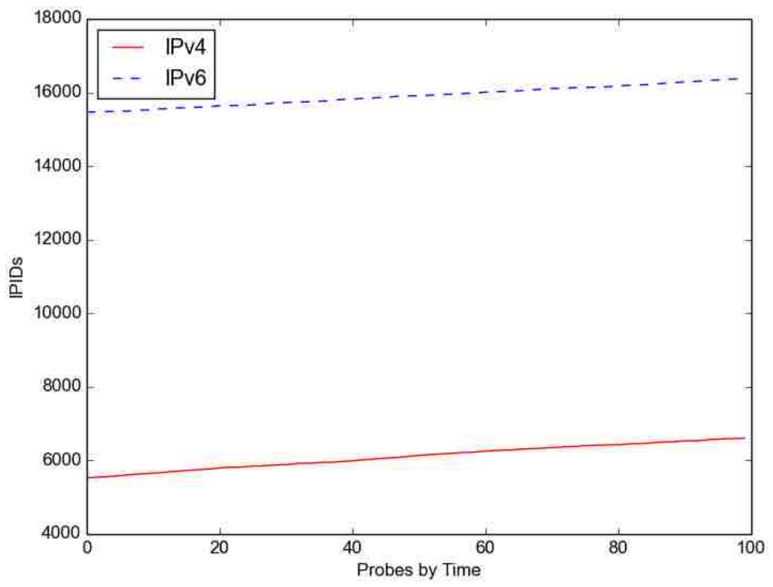

Figure 5.5: Alias resolution result (not dual stack).

reply adding IPv6 fragment extension headers containing fragment IDs. We alternate sending each IPv4 and IPv6 probe, waiting 0.05 seconds between sending each probe.

If x_i stands for the ith result of the IPv4 probe, and y_i stands for the ith result of the IPv6 probe (mod 2^{16}), then if we have a strictly increasing sequence $x_1 < y_1 < x_2 < y_2 \ldots x_n < y_n$ and $y_i - x_i \geq 1$ and $\sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i > n$, where $n = 10$, then we conclude that the machine is dual-stacked. (In this analysis, we say that $X < Y$ if Y occurs in X's upcoming half of the 16-bit sequence space.)

As shown in Figure 5.5, the values of x_i and y_i are incrementing irrespective of each other, which means that they are hashed to different buckets. In contrast, in Figure 5.6, the values of x_i and y_i take turns incrementing the same counter, which means that the addresses being tested hash to the same IPID counter. Our novel alias resolution technique which uses TCP/IP side channels is able to find Linux 3.16 or later IPv4 and IPv6 dual stack machines to be used as zombies in ONIS.

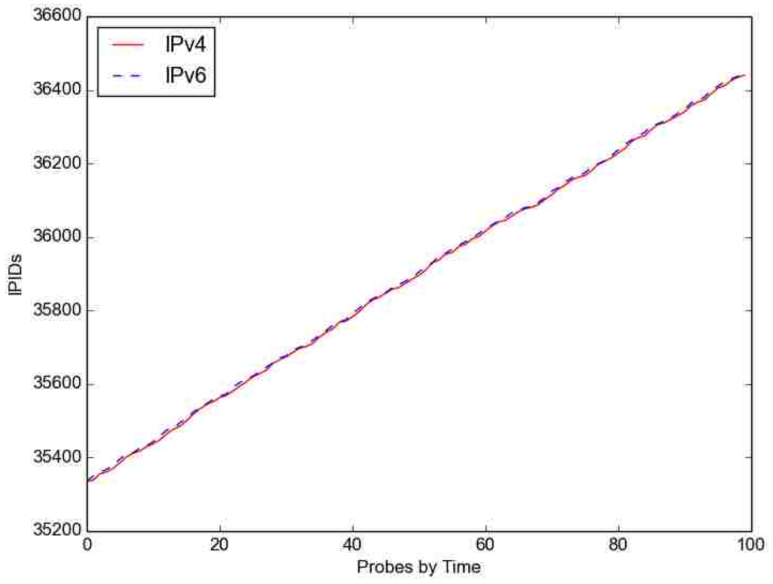

Figure 5.6: Alias resolution result (dual stack).

5.3.3 Finding a collision with target's IP address

In the previous section, we discussed a new technique to do IPv4 and IPv6 alias resolution on Linux machines with kernel 3.16 or later. Assuming we will use an IPv4 and IPv6 dual stack machine as the zombie machine in ONIS, we also need to know the exact source IPv6 address that causes the collision with the target machine's IPv4 address. Similar to the previous method, we pick a source IPv6 address and send a 2000 byte IPv6 Echo Request to the zombie at t_1. Then we send a spoofed IPv4 SYN-ACK packet using the target's return address to the zombie at t_2. Finally, we send another 2000 byte IPv6 Echo Request at t_3.

We want our three probes to arrive in the order in which we sent them, but also, to eliminate the random noise that Linux adds to its IPID counters, we want them to arrive within one system tick from each other. This eliminates all noise because when they arrive one tick after each other, the kernel will increase the IPID counter

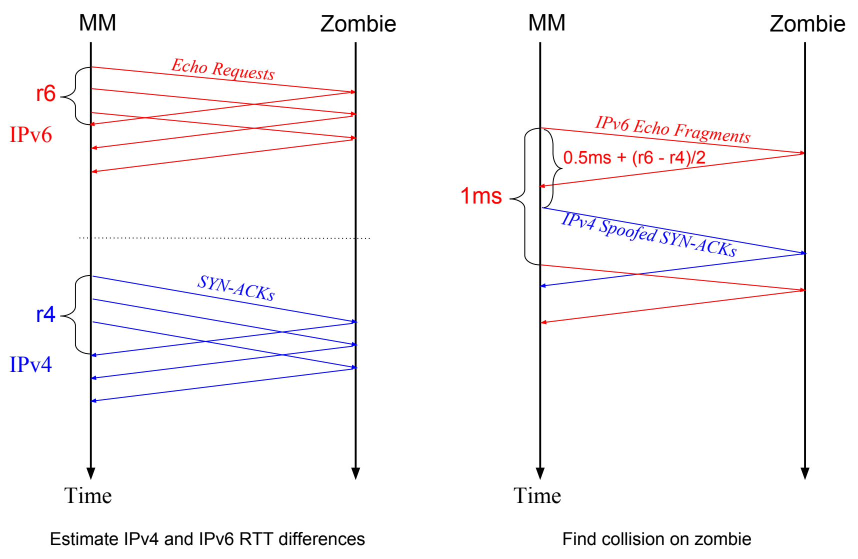

Figure 5.7: Efficient way to find collision on zombie.

by $\mathcal{U}(1,1) = 1$, but if they arrive within zero change of the system clock, the kernel still increments by one. If we can have the packets arrive in order and within one system tick of each other, then, for the open port case, we will observe an IPID increase of $1 + 1 = 2$ and for the closed port case, we will observe an IPID increase of only one.

Naively, we might want to send the probes at $t_2 = t_1 + 0.5$ms, and $t_3 = t_1 + 1$ms. (We choose milliseconds because, while the tick rate of the kernel is never faster than one tick per millisecond, it may be slower.) However, often the round trip time (RTT) between the measurement machine and the zombie are different for IPv4 versus IPv6 routes and so the order of IPv4 packets and IPv6 packets arriving at the zombie might be at different times than we expect.

To overcome this, we first send IPv4 and IPv6 probe packets to the zombie to

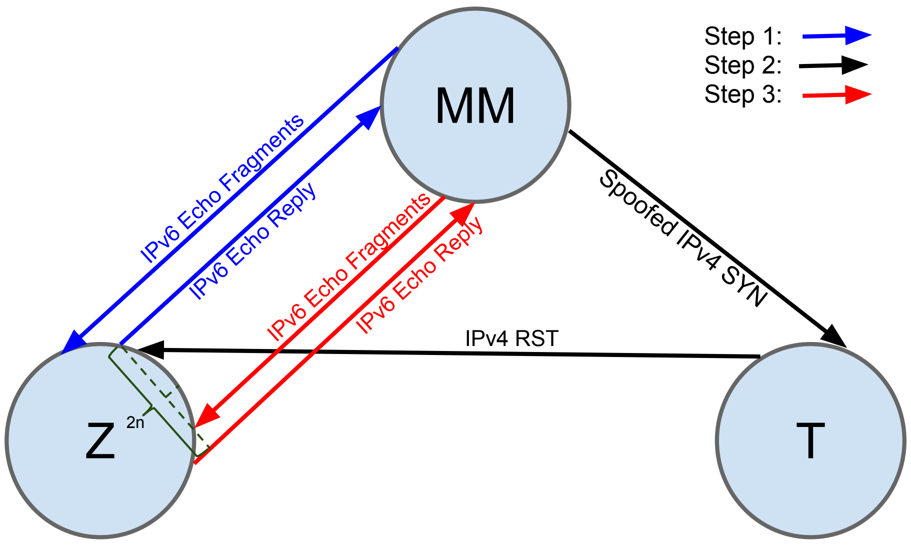

Figure 5.8: Scan of a closed port with a dual stack zombie using ONIS.

estimate the average RTT between the zombie and the measurement machine both in IPv4 and IPv6. Let the measured IPv4 RTT be r_4, the measured IPv6 RTT be r_6, and the difference between them δ such that $r_4 + \delta = r_6$. By approximating the path from the measurement machine to the zombie as half of the RTT, we divide δ by two and use the adjustment $t_2 = t_1 + 0.5 + \delta/2$, leaving t_3 unchanged. This way we can increase the chance that all three probes arrive within one system tick of each other. Note than an IPv4-only version of ONIS would not face this challenge.

Figure 5.7 shows how this was implemented. If the IPID returned from the third probe is at least two greater than that of the IPID returned by the first probe, we conclude that the tested IPv6 address shares the same IPID counter as that of the target. Otherwise, we conclude that it does not. (Here we say that $Y > X$ if Y occurs in X's upcoming half of the 32-bit sequence space.)

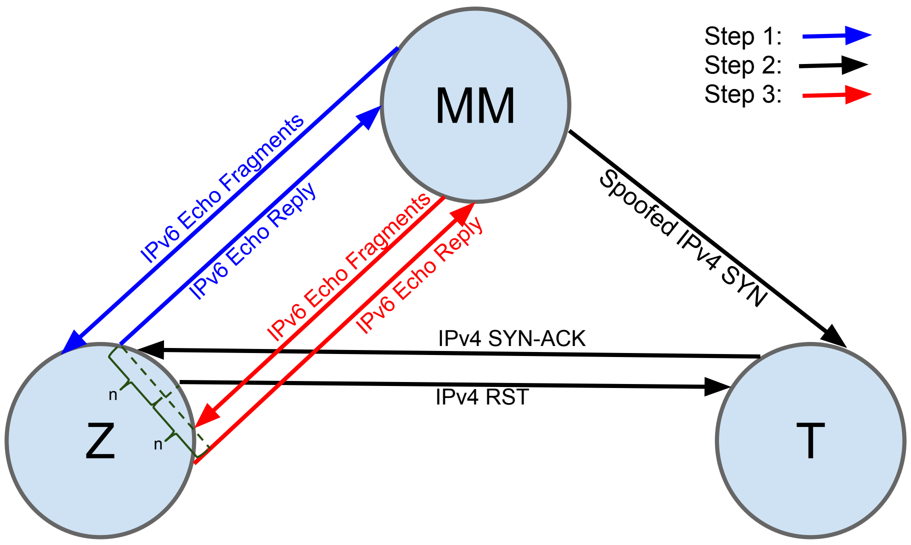

Figure 5.9: Scan of an open port with a dual stack zombie using ONIS.

5.3.4 An implementation of ONIS using dual-stack zombies

After discovering a source IPv6 address that shares an IPID counter with the target's IPv4 address, we can adapt ONIS to use dual stack Linux machines as zombie machines. In the first step, the measurement machine sends a fragmented IPv6 Echo Request, whose unfragmented datagram is 2000 bytes, to the zombie and records the IPID in the fragmented response. As before, a large request is used to ensure that the response is fragmented, ensuring that the IPv6 fragmentation extension header containing an IPID is included. The second step is exactly as before. In the final step, the measurement machine queries the IPID again by sending another fragmented IPv6 Echo Request. Figure 5.8 shows using ONIS when the target's port is closed. Figure 5.9 shows using ONIS when target's port is open. After collecting a group of IPIDs, we use the method in Section 5.3.5 to determine which model fits the data.

5.3.5 Model selection and noise handling

As mentioned in Section 5.3.1, in order to find out if a target's port is open or closed, we need to be able to distinguish between two distributions, $\mathcal{U}(1, t_3 - t_1)$ and $\mathcal{U}(1, t_2 - t_1) + \mathcal{U}(1, t_3 - t_2)$. Let $t_3 - t_1 = 2n$. Moreover, we will approximate $t_2 - t_1 = t_3 - t_2 = n$. Now we need only distinguish between the distributions $\mathcal{U}(1, 2n)$ and $\mathcal{U}(1, n) + \mathcal{U}(1, n)$. We showed that it is trivial to distinguish simulated cases. However, in the experiment we found noise on the network made such analysis challenging. For example, the round trip time between the measurement machine and the zombie changed in different rounds when collecting IPIDs. Thus in practice we have $\mathcal{U}(1, 2n + \delta)$, where δ is a variant.

To overcome these issues, we use a model selection method which is resistant to noise when collecting IPIDs called Akaike information criterion (AIC). Unlike the null hypothesis testing approach, AIC does not give the quality of a single model with respect to a null hypothesis, but rather estimates the relative quality of one model with respect to another. Because of this, AIC is ideal for us to handle noise in the scan.

We use AIC to select between $\mathcal{U}(1, 2n)$ and $\mathcal{U}(1, n) + \mathcal{U}(1, n)$ for a given measurement dataset of IPIDs. By definition, AIC is defined as

$$AIC = 2k - 2\ln(\hat{L}) \tag{5.1}$$

where $\hat{L}$ is the maximum value of the likelihood function, and k is the number of parameters in the model. In our case, $k = 1$. Thus, according equation 5.1, we wish to find the model with larger maximum likelihood, $\hat{L}$, in order to get a smaller AIC value. Below we will show how to calculate $\hat{L}$ in both of the cases of our scan.

Case 1: Closed or filtered port

Assuming that an observed IPID increase was generated by $\mathcal{U}(1, 2n)$, the probability density function is:

$$f(x) = \begin{cases} \frac{1}{2n} & \textit{for } 1 \leq x \leq 2n \\ 0 & \textit{otherwise.} \end{cases} \tag{5.2}$$

The likelihood function is:

$$L(n) = \prod_{i=1}^{k} f(x_i) = \begin{cases} \frac{1}{(2n)^k} & \textit{for } 1 \leq x_i \leq 2n \textit{ for all } i \\ 0 & \textit{otherwise.} \end{cases} \tag{5.3}$$

where k is our sample size.

Note that the likelihood function defined in equation 5.3 may have maximum value when $1 \leq x_i \leq 2n$ for all i. Therefore, $\max\{x_1, \ldots, x_k\} \leq 2n$, and $n \geq \lceil \frac{\max\{x_1,\ldots,x_k\}}{2} \rceil$. Note that in equation 5.3, $\frac{1}{(2n)^k}$ is monotonically decreasing when $n \geq 1$, *i.e.*, when $n = \lceil \frac{\max\{x_1,\ldots,x_k\}}{2} \rceil$, the likelihood function $L(n)$ in equation 5.3 gets its maximum value.

Case 2: Open port

Assuming that an observed IPID increase is generated by $\mathcal{U}(1, n) + \mathcal{U}(1, n)$, the probability density function is

$$f(x) = \begin{cases} \frac{n-|x-(n+1)|}{n^2} & \textit{for } 2 \leq x \leq 2n \\ 0 & \textit{otherwise.} \end{cases} \tag{5.4}$$

The likelihood function is:

$$L(n) = \frac{1}{n^{2k}} \prod_{i=1}^{k} (n - |x_i - (n+1)|) \tag{5.5}$$

for $2 \leq x_i \leq 2n$, for all i.

The monotonicity of the likelihood function in equation 5.5 cannot be determined *a priori*. However, we know that to get the maximum likelihood, $2 \leq x_i \leq 2n$ for all i, *i.e.* $\max\{x_1, \ldots, x_k\} \leq 2n$, and $n \geq \lceil \frac{\max\{x_1,\ldots,x_k\}}{2} \rceil$. Therefore, we adopt a numerical approach and enumerate multiple possible n such that $n \geq \lceil \frac{\max\{x_1,\ldots,x_k\}}{2} \rceil$ to see which n maximizes $L(n)$ in equation 5.5. We try m such values from $\lceil \frac{\max\{x_1,\ldots,x_k\}}{2} \rceil$ to $\lceil \frac{\max\{x_1,\ldots,x_k\}}{2} \rceil + m$, finding the n that gives the maximum $L(n)$. The parameter m is tunable, but we provide evidence that it is typically very low in Section 5.4.2.

5.4 Experimental Setup

In this section we describe the details of our experimental setup.

5.4.1 Zombie selection

The measurement machine we used ran Ubuntu server 16.04 with kernel version 4.4.0. The zombie machines we selected were IPv4 and IPv6 dual-stack Linux machines with kernel 3.16 or later. In the beginning, we collected a list of domains from the Alexa top one million websites and sixy.cn. For each domain name, we performed a DNS lookup to find its corresponding A record and AAAA record. Given pairs of IPv4 and IPv6 address, we performed IPv4 and IPv6 alias resolution using the method described in Section 5.3.2. The IPv4 and IPv6 resolution tests were performed three times for every IPv4 and IPv6 pair to make sure it was a desired zombie. The rate we created packets on the zombie's network was about 30 packets per second. Note that to compare IPIDs in IPv4 and IPv6, we used a 16-bit mask to mask out the leftmost 16 bits of IPIDs in IPv6. We found 78 zombies which we used in our implementation of ONIS.

5.4.2 Performing the scan

To select a target that we want to perform the ONIS on, we randomly generated an IPv4 address and started to send SYN packets to three commonly open ports (22, 80 and 443) at that address. Each packet was re-sent three times to avoid possible packet loss. We recorded the IPv4 address as a valid target address if we received a SYN-ACK response for any of the three ports.

Once the scan started, the measurement machine created multiple threads, each randomly picking up a zombie machine from the pool and finding a valid target address as discussed above to perform the scan on ports 22, 80, and 443. The scanning methodology was described in Section 5.3. For each port on a target, we collected 100 IPv6 IPID pairs, at a packet rate of three packets per second in the zombie's network. Each experiment takes about 20 minutes to finish. Then we processed the IPID samples by using AIC (see Section 5.3.5).

In Section 5.3.5, we mentioned that in order to find the maximum likelihood for the open port model, we had to enumerate the parameter n in m times. Each time we added one to n and calculated the corresponding likelihood L. During our experiment, we set $m = 1000$, as in practice we found out from our result that most of the time the first few i, where $0 \leq i \leq m$ lead to the maximum likelihood. Figure 5.10 shows the distribution of i which gives the maximum likelihood for model $\mathcal{U}(1, n) + \mathcal{U}(1, n)$. n is virtually always less than 1000.

To compare the accuracy of our results, we ran nmap's built-in idle scan (*nmap -Pn*) on the same targets that we performed ONIS on. Nmap implements the idle scan technique by using zombies with globally incrementing IPIDs. To find machines with globally incrementing IPIDs, we generated random IP addresses and tested to see if the IPIDs were globally incrementing. After we got a list of zombies, we culled the list for several rounds until all zombies appeared to be idle. We were able to

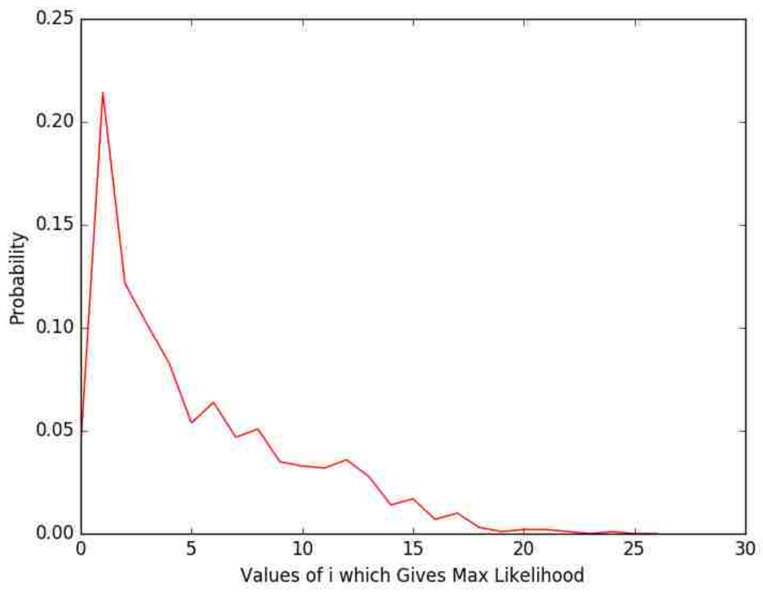

Figure 5.10: After trying $m = 1,000$ different n, the distribution of iterations i it took to find the n that maximizes the likelihood function while performing ONIS.

identify 175 machines with globally incrementing IPIDs. Both the nmap idle scan results and ONIS results were compared with direct scan (SYN scan) results in order to calculate the accuracy.

5.5 Results

We collected 1309 results using 78 zombies starting May 1, 2017 and ending May 12, 2017. Each result showed whether specific ports (22, 80, 443) on a target machine were open or closed. We compared ONIS results with direct scan results and found that 1141 out of 1309 are correct, with an accuracy of 87.2%. There were 145 false negatives (failed to find an open port) and 23 false positives (reported a port as open that was not) out of 168 incorrect results. One possible reason for more false positives might be due to the fact that we found an incorrect collision with target IPv4 address using our IPv6 address before the scan. Thus once the scan started, the IPIDs in the TCP flow were not hashed into the expected bucket.

For comparison, we also collected 175 zombies with global incrementing IPIDs. To form a direct comparison between nmap results and ONIS results, we performed nmap idle scan on the same 1309 results. The nmap idle scan had 86.4% accuracy, with 1131 correct results and 178 incorrect results. 64 of the results were false positive, 57 of the results were false negative. 57 of the results showed that the zombie was too noisy to be used to perform the idle scan.

The resulting comparison of the two methods is shown in Table 5.1. We can see that the overall accuracy of the two methods is comparable. Both scans have a certain amount of false negatives. One reason for that might be the possible ingress filtering in the target's network which prevents the spoofed SYN packets from the zombie. As a result, there is no subsequent traffic created. The zombie will not generate an IPID in response to the client. Both scans falsely assume that the port is closed in this case.

We also noticed that ONIS has more false negatives than the idle scan. We believe that is due to the fact that the previous step of collision finding is very sensitive to round trip time variations between IPv4 and IPv6. For example, route changes of IPv6 can cause the round-trip time for IPv6 between the zombie and the measurement machine to be larger. As a result, the second IPv6 echo fragments arrive at the zombie later than 1 millisecond. In this case, it is possible that the $\mathcal{U}(1, 1+\alpha)$ generates a number larger than 1, where α is a delay of the second IPv6 packet fragments.

Table 5.1 also shows that nmap has more false positives. Although we ensure that zombies with global IPID on our list are all idle before the scan, it is possible

	Corret	False Positive	False Negative	Failed
ONIS	1141	23	145	0
Idle Scan	1131	64	57	57

Table 5.1: Result Comparision of ONIS and the Idle Scan

that during the nmap idle scan the zombie is connecting with other hosts, causing it to no longer be idle thus breaking the scan. As a result, nmap falsely thinks that the target has an open port. We also noticed that during the nmap idle scan, every zombie may become active at certain time. In order to get ideal results with nmap's current implementation, we need to cull the list such that the zombies were noiseless right before we start the idle scan. Otherwise, the accuracy of results drops significantly. While for ONIS, there is only a 1/2048 chance of interference with any other host because of Linux's 2048-bucket implementation.

5.6 Discussion

5.6.1 Scan applicability

ONIS allows a broader choice of zombies and it is more reliable compared to the idle scan, which uses zombies with globally incrementing IPIDs.

The idle scan requires the zombie to be completely idle, while ONIS does not because the chance of every other connection on the zombie with the same global incrementing counter is just 1/2048. Machines on the Internet are seldom idle, so in this sense, ONIS greatly improves on the reliability of the idle scan.

We set up experiments by using dual-stack zombies with Linux kernel 3.16 or later. This is just one implementation of ONIS to show that it works. We also provided a new technique to perform IPv4 and IPv6 alias resolution for Linux systems. However, ONIS is not limited to use on IPv4 and IPv6 dual stack systems.

In applications where the zombie can be, *e.g.*, from a specific large network (such as a particular country) and need not be a specific machine, we can scan and get a list of Linux machines with kernel 3.16 or later. Then for a randomly selected target, we can try every zombie in our list until a collision is found. Each time there's a 1/2048 chance that the measurement machine will share the same IPID counter on the zombie as the target. For 10,000 zombies, there is more than a 99% probability of finding a collision.

ONIS allows broader choices of zombies when inferring TCP/IP-based trusting relationship off-path. We performed a IPv4 SYN-ACK scan on the Alexa top web 1 million machines and found 170,630 of them to have per-flow IPIDs (about 17%), which are potential zombies that can be used in ONIS.

5.6.2 Ethical concerns

Compared to the idle scan, we perform an extra step to perform collision finding. However, the packets we send in this step are only IPv6 echo fragments and IPv4 SYN-ACKs, which should not consume too much computation resources on zombies' systems. Our IPv6 echo request is 2000 bytes long, and an IPv4 SYN-ACK is just 60 bytes. At a packet rate of 30 packets per second, we can create 40.6 kilobytes per second per zombie. During the scan on the target, the packet rate is only 3 packets per second. Since the spoofed SYNs to the target will end up reset by the zombie, it will not cause any denial of service on the target.

5.6.3 Defending against ONIS

One way that the Linux kernel could protect against being used as a zombie is to switch to a different kind of IPID counter. However, it is unclear which kind could protect widely against this sort of scan. Linux previously used per-destination counters, but this exposed them to a side channel attack that could leak the number of packets a machine sends another [48]. However, as the authors of that attack note, RFC 791 [62] mandates that IPIDs must be unique for every in-flight path, and so there will always be non-zero information flow in any shared sequence of numbers that has restrictions on repetition.

A strategy that may help defend against the technique used in this chapter is to use a Poisson distribution instead of a uniform distribution for generating IPID noise. Our technique takes advantage of the fact that $\mathcal{U}(0, n)$ has a different distribution than $\mathcal{U}(0, n/2) + \mathcal{U}(0, n/2)$. However, the Poisson distribution does not have this limitation. If $\mathcal{P}(\lambda)$ is a Poisson random variable parameterized by rate λ, then $\mathcal{P}(\lambda) + \mathcal{P}(\mu) = \mathcal{P}(\lambda + \mu)$, and so $\mathcal{P}(\lambda) = \mathcal{P}(\lambda/2) + \mathcal{P}(\lambda/2)$. However, Poisson random numbers are computationally expensive to generate, and so they may not be suitable as a means to add noise to an IPID counter that may be accessed frequently.

5.7 Summary of this Chapter

We presented ONIS, a novel scanning technique which provides much broader choices of zombies when performing off-path TCP/IP trust relationship measurement. The accuracy of ONIS is comparable to nmap's implementation. One caveat is that ONIS requires access to many IP addresses for the measurement machine, but the scan is flexible enough to enable different trade-offs in this sense, and IP addresses are easily obtainable in various ways. We expect that ONIS will become an essential part of a network researcher's toolbox and fulfill the practical potential for various network measurement tasks.

Chapter 6

Conclusion and Future Work

In summary, this dissertation has provided penetration testers, digital forensics experts, and general network researchers a set of new techniques to fulfill modern network measurement tasks. We first presented a technique to find machines hidden behind firewalls, which is a critical first step for penetration testers to understand modern network structures. Then we presented an improved off-path round-trip time measurement technique with high fidelity, which is important for network researchers to learn IP geolocation information and understand global Internet routing, especially when the availability of measurement infrastructure is limited. At last, we presented a novel technique to detect the trust relationships between two arbitrary machines off-path, which is essential to map out firewall rules, to measure international packet drops, or to detect censorship.

Previous TCP/IP side channel research showed that information flow is unavoidable because modern operating systems have shared, limited resources. We showed that there are better ways to utlize side channels in terms of reducing the costs on the target sytems in order to be non-intrusive, as well as improving the fidelity of the results by: (1) using mathematical models to reduce the influence of noise, and (2) deeply understanding the modern systems' network stack behavior. We also showed that new side channels can be discovered and utlized to overcome the limitations of previous side channels as systems are changing their implementation, for example, using per flow IPID instead of global IPID to reduce the noise by using side channels in the newest Linux systems (kernel version 3.16 and later).

Side channel techniques were applied to perform some network measurement tasks, which cannot be done by using traditional measurement tools. However, the limitations of previous side channel techniques prevent them from becoming reliable and widely applied network measurement tools for modern network measurement tasks. The side channel techniques we proposed demonstrate better ways of using old side channels, as well as possibilities of finding new side channels on modern system kernels, to accomplish network measurement tasks ethically and efficiently. We show a promising future for side channel techniques to help penetration testers, digital forensics experts, and general network researchers with various tasks of modern network measurements.

For future work, there are three main directions:

1. One is to apply our set of tools to various network measurement tasks: *e.g.* using the off-path round trip time technique to perform global IP geolocation, which would be very helpful to improve the precision of current geolocation databases; using ONIS to detect packet drops between a host and a Tor bridge. This is important to the study of censorship because sometimes suitable vantage points or access to certain services are not available to network researchers.

2. Second is to use a machine learning approach to automatically find side channels. Currently TCP/IP side channels are found by manually checking the kernels' source code. A tool which can automatically create and send layer 3 probes to target systems and then find the corresponding side channels would greatly

help system developers to find potential side channels much more quickly.

3. At last, there are also some improvements we can apply to the individual techniques as mentioned in Sections 3.8, 4.7, and 5.7. For example, implementing a back off scheme when detecting other scans performed on the same machine, adopting the side channels to a broader set of server OSes, *etc.*

References

[1] FTP-bounce. https://nmap.org/nsedoc/scripts/ftp-bounce.html.

[2] MASSCAN: Mass IP port scanner. https://github.com/robertdavidgraham/masscan.

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

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

[5] RIPE Atlas. https://atlas.ripe.net/about/.

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

[7] SYN Cookies. http://en.wikipedia.org/wiki/SYN_cookies.

[8] Unicorn Scan. http://www.unicornscan.org/.

[9] Zmap. https://zmap.io/.

[10] E. S. Al-Shaer and H. H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2605–2616. IEEE, 2004.

[11] G. Alexander and J. R. Crandall. Off-path round trip time measurement via TCP/IP side channels. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1589–1597. IEEE, 2015.

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

[13] F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. *Computer Networks*, 42(6):717–735, 2003.

[14] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 17–31. IEEE, 1999.

[15] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 123–134. ACM, 1999.

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

[17] D. J. Bernstien. SYN Cookies. `http://cr.yp.to/syncookies.html`.

[18] J. Cannady. Artificial neural networks for misuse detection. In *National information systems security conference*, pages 368–81, 1998.

[19] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, and L. M. Marvel. Off-path tcp exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225. USENIX Association.

[20] W. Chen, Y. Huang, B. F. Ribeiro, K. Suh, H. Zhang, E. d. S. e Silva, J. Kurose, and D. Towsley. Exploiting the IPID field to infer network path and end-system characteristics. In *Passive and Active Network Measurement*, pages 108–120. Springer, 2005.

[21] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: An overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003.

[22] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 15–26. ACM, 2004.

[23] M. De Vivo, E. Carrasco, G. Isern, and G. O. de Vivo. A review of port scanning techniques. *ACM SIGCOMM Computer Communication Review*, 29(2):41–48, 1999.

[24] E. Dumazet. inetpeer: get rid of ip_id_count, 2014.

[25] E. Dumazet. ip: make ip identifiers less predictable, 2014.

[26] R. Ensafi, J. Knockel, G. Alexander, and J. R. Crandall. Detecting intentional packet drops on the Internet via TCP/IP side channels: Extended version. *CoRR*, abs/1312.5739, 2013. Available at `http://arxiv.org/abs/1312.5739`.

[27] R. Ensafi, J. Knockel, G. Alexander, and J. R. Crandall. Detecting intentional packet drops on the Internet via TCP/IP side channels. In *Passive and Active Measurement*, pages 109–118. Springer, 2014.

[28] 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 *USENIX Security Symposium*, pages 257–272, 2010.

[29] D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 827–835. Society for Industrial and Applied Mathematics, 2001.

[30] T. Flach, E. Katz-Bassett, and R. Govindan. Quantifying violations of destination-based forwarding on the Internet. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 265–272, New York, NY, USA, 2012. ACM.

[31] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *Networking, IEEE/ACM Transactions on*, 9(5):525–540, 2001.

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

[33] C. Gates. Coordinated Scan Detection. In *NDSS*, 2009.

[34] 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*, pages 402–408. IEEE, 2006.

[35] Y. Gilad and A. Herzberg. Fragmentation considered vulnerable: blindly intercepting and discarding fragments. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 2–2. USENIX Association, 2011.

[36] Y. Gilad and A. Herzberg. Spying in the dark: Tcp and tor traffic analysis. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 100–119. Springer, 2012.

[37] Y. Gilad and A. Herzberg. Off-path tcp injection attacks. *ACM Transactions on Information and System Security (TISSEC)*, 16(4):13, 2014.

[38] M. G. Gouda and X.-Y. Liu. Firewall design: Consistency, completeness, and compactness. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 320–327. IEEE, 2004.

[39] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18. ACM, 2002.

[40] J. D. Guttman. Filtering postures: Local enforcement for global policies. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 120–129. IEEE, 1997.

[41] J. D. Guttman and A. L. Herzog. Rigorous automated network security management. *International Journal of Information Security*, 4(1-2):29–48, 2005.

[42] A. Hari, S. Suri, and G. Parulkar. Detecting and resolving packet filter conflicts. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1203–1212. IEEE, 2000.

[43] N. Hariri, B. Hariri, and S. Shirmohammadi. A distributed measurement scheme for Internet latency estimation. *Instrumentation and Measurement, IEEE Transactions on*, 60(5):1594–1603, 2011.

[44] U. Javed, I. Cunha, D. Choffnes, E. Katz-Bassett, T. Anderson, and A. Krishnamurthy. PoiRoot: Investigating the root cause of interdomain path changes. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 183–194, New York, NY, USA, 2013. ACM.

[45] 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*, pages 211–225. IEEE, 2004.

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

[47] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. Anderson, and A. Krishnamurthy. Reverse traceroute. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 15–15, Berkeley, CA, USA, 2010. USENIX Association.

[48] J. Knockel and J. R. Crandall. Counting packets sent between arbitrary internet hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*, 2014.

[49] J. Knockel and J. R. Crandall. Counting packets sent between arbitrary internet hosts (slides), 2014.

[50] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.

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

[52] 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*, volume 5, pages 3178–3182. IEEE, 2004.

[53] A. X. Liu and M. G. Gouda. Firewall policy queries. *Parallel and Distributed Systems, IEEE Transactions on*, 20(6):766–777, 2009.

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

[55] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 367–380. USENIX Association, 2006.

[56] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, pages 177–187. IEEE, 2000.

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

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

[59] T. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 170–179. IEEE, 2002.

[60] P. Pearce, R. Ensafi, F. Li, N. Feamster, and V. Paxson. Augur: Internet-wide detection of connectivity disruptions. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 427–443. IEEE, 2017.

[61] J. Postel. Transmission Control Protocol. RFC 793, RFC Editor, September 1981.

[62] J. Postel et al. Rfc 791: Internet protocol. 1981.

[63] Z. Qian and Z. M. Mao. Off-path TCP sequence number inference attack-how firewall middleboxes reduce security. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 347–361. IEEE, 2012.

[64] 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*, CCS '12, pages 593–604, New York, NY, USA, 2012. ACM.

[65] A. Quach, Z. Wang, and Z. Qian. Investigation of the 2016 linux tcp stack vulnerability at scale. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):4, 2017.

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

[67] 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*, pages 646–649. IEEE, 2008.

[68] S. Staniford, J. A. Hoagland, and J. M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1):105–136, 2002.

[69] J. Treurniet. A network activity classification schema and its application to scan detection. *Networking, IEEE/ACM Transactions on*, 19(5):1396–1404, 2011.

[70] Y. A. Wang, C. Huang, J. Li, and K. W. Ross. Queen: Estimating packet loss rate between arbitrary Internet hosts. In *Passive and Active Network Measurement*, pages 57–66. Springer, 2009.

[71] A. Wool. Architecting the Lumeta Firewall Analyzer. In *USENIX Security Symposium*, pages 85–97, 2001.

[72] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[73] X. Zhang, J. Knockel, and J. R. Crandall. Original SYN: Finding machines hidden behind firewalls. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 720–728. IEEE, 2015.

[74] X. Zhang, J. Knockel, and J. R. Crandall. High fidelity off-path round-trip time measurement via tcp/ip side channels with duplicate syns. In *Global Communications Conference (GLOBECOM), 2016 IEEE*, pages 1–6. IEEE, 2016.

[75] H. Zheng, E. K. Lua, M. Pias, and T. G. Griffin. Internet routing policies and round-trip-times. In *Passive and Active Network Measurement*, pages 236–250. Springer, 2005.