Non-Trivial Off-Path Network Measurements without Shared Side-Channel Resource Exhaustion

Geoffrey I. Alexander

Geoffrey Alexander

Candidate

Computer Science

Department

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

Approved by the Dissertation Committee:

Jedidiah R. Crandall , Chairperson

Soraya Abad-Mota

Jedidiah McClurg

Phillipa Gill

Non-Trivial Off-Path Network Measurements without Shared Side-Channel Resource Exhaustion

by

Geoffrey Alexander

B.A., Computer Science, University of New Mexico, 2011

M.S., Computer Science, University of New Mexico, 2015

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

December 2019

Acknowledgments

I would like to thank my advisor, Dr. Jedidiah R. Crandall for supporting and helping with my research over the years. I would also like to thank my dissertation committee: Dr. Soraya Abad-Mota, Dr. Phillipa Gill, and Dr. Jedidiah McClurg for serving on my dissertation committee and for their valuable feedback and suggestions on this dissertation. I would also like to thank Dr. Abdullah Mueen for his guidance when performing the clustering analysis carried out in Chapter 5.

I would like to thank my parents for showing me the value of education and helping to nourish my love for the sciences. Lastly, I would like to thank my wife, Lindsay. Without her continued love and support this dissertation would not have happened.

Non-Trivial Off-Path Network Measurements without Shared Side-Channel Resource Exhaustion

by

Geoffrey Alexander

B.A., Computer Science, University of New Mexico, 2011
M.S., Computer Science, University of New Mexico, 2015

PhD, Computer Science, University of New Mexico, 2019

Abstract

Most traditional network measurement scans and attacks are carried out through the use of direct, on-path network packet transmission. This requires that a machine be on-path (i.e., involved in the packet transmission process) and as a result have direct access to the data packets being transmitted. This limits network scans and attacks to situations where access can be gained to an on-path machine. If, for example, a researcher wanted to measure the round trip time between two machines they did not have access to, traditional scans would be of little help as they require access to an on-path machine to function. Instead the researcher would need to use an *off-path* measurement scan.

Prior work using network side-channels to perform off-path measurements or attacks relied on techniques that either exhausted the shared, finite resource being used as a side-channel or only measured basic features such as connectivity. The work presented in this dissertation takes a different approach to using network side-channels. I describe research that carries out network side-channel measurements that are more complex than connectivity, such as packet round-trip-time or detecting active TCP connections, and do not require a shared, finite resource be fully exhausted to cause information to leak via a side-channel. My work is able to accomplish this by understanding the ways in which internal network stack state changes cause observable behavior changes from the machine. The goal of this dissertation is to show that: *Information side-channels can be modulated to take advantage of dependent, network state behavior to enable non-trivial, off-path measurements without fully exhausting the shared, finite resources they use.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network researchers have a large variety of tools at their disposal to measure *on-path* network characteristics and behaviors. *On-path* machines are involved in routing packets between the two endpoints of a given network communication stream while *off-path* machines are not involved in routing. These tools include programs such as `NMAP` [2] which can scan for open ports, detect operating system type and version, or identify the service running on a given port. However, these tools all rely on information gathered from on-path scans and machines.

As an example consider `NMAP`'s port scanning measurement. This technique can determine if a given port is listening for and accepting connections by sending `SYN` packets directly to the target machine. This can be used to determine if the *measurement* machine can connect to a given port on another machine. However, it is not possible for such a technique to determine if the same port can be accessed from another arbitrary machine on the Internet. Such information is hidden from any machines that are not on-path.

In addition to determining if a given remote machine can connect to another, researchers may be interested in determining if such a connection currently exists between two machines. Again this information is readily available to on-path machines. They simply record any packets transmitted between the two machines and check to see if they correspond to an active connection. Determining if such a connection exists between two off-path machines is much more difficult. Researchers cannot capture off-path packets to check for an active connection. Using traditional on-path measurement techniques the presence of off-path connections cannot be measured.

Now, consider two machines communicating over the Internet. Data transmissions between these two machines will take a given amount of time to travel from one machine to the other and back again. This is referred to as the round trip time (RTT) between the two machines. Normally this information is only available to on-path machines involved in routing packets between the two end hosts. Off-Path machines cannot measure the RTT because they cannot see the packets and cannot measure how long one round trip takes. To work around this researchers will often attempt to use "nearby" machines as proxies when performing measurements. "Nearby" machines refer to measurement machines in the same region, country, or network as a desired target machine. These machines could be chosen from measurement networks such as PlanetLab [18], RIPE Atlas [69], or OONI [3]. However, these networks rely on volunteers to install and maintain measurement nodes. This results in cases where there are not any "nearby" machines to act as proxies, as the network may not have machines installed in a given country, region, or network.

Each of the situations described above represent an area when *on-path* measurement techniques prove unable to provide researchers with the data they might want. These limit the kinds of measurement scans researchers can perform and represent a gap in our ability to study and measure the networks. While there has been some prior work studying *off-path* techniques to address these limits via side-channel techniques, these require that researchers fully exhaust the shared, side-channel resources on remote machines. Exhausting shared resources can cause an undesirable impact on these remote machines. The impact can range from a noticeable decrease in system performance up to a complete denial of service. For researchers seeking to carry out responsible *off-path* measurements this resource exhaustion presents a major problem. In this work I will develop new techniques for performing non-exhaustive *off-path* measurements that will provide information not available via *on-path* measurements, with the need to fully exhaust shared resources on the remote machines involved.

1.1 Dissertation Overview

In Chapter 2, I review previous work related to network measurement and network side-channels. This includes work covering both *on-path* and *off-path* measurement techniques, network side-channels, and Internet scale studies of common host behaviors.

Chapter 3 discusses my technique for measuring the round trip time (RTT) between arbitrary Internet hosts from an *off-path* measurement machine. This is a novel technique that uses network side-channels present in the Linux kernel's `SYN`-backlog and is, to my knowledge, the first such work that does not require the use of an additional "proxy" machine to measure round trip time. Instead my technique only requires the use of three machines, those whose round trip time is being measured and the machine carrying out the *off-path* measurements.

Chapter 4 presents a novel technique for *off-path* detection of active TCP connections using novel side-channels present in the Linux kernel's handling of the IPv4 identification field (IPID) for outgoing TCP packets. This technique exploits the kernel's use of 2048 global IPID counters for packets sent outside established TCP connections and *per-connection* counters for packets sent inside established TCP connections. By measuring changes to the IPID field as a result of measurement probes an *off-path* machine can detect when the Linux kernel changes from using a global counter to a *per-connection* counter and thereby detect the presence of an active TCP connection.

Chapter 5 details an Internet scale study of IPv4 IPID generation algorithms and the potential side-channels they may contain. This study is the first that I am aware of to study the entire IPv4 address space, without making any assumptions about the number and specific behaviors of IPID generation algorithms used. In addition to studying the IPID generation algorithms used across the entire Internet, this chapter also details my attempts to determine if shared information flows exist for each different algorithm I identify. This study is a first step in attempting to automate part of the process of locating potential network stack side-channels, which normally requires time consuming manual analysis and reverse engineering.

In Chapter 6 I outline potential directions for future research and investigation. In Chapter 7 I summarize my findings and provide my concluding remarks.

Chapter 2

Related Work

Network side-channels for network measurement or network attacks is an area of research that has seen less focus than other types of network security or scanning. In addition, many of the side-channel techniques discussed below fully exhaust the given resource they are using as a side-channel or are only concerned with simple connectivity measurements. Techniques that do not fully exhaust shared resources, or measure features beyond simple connectivity, constitute a relatively unstudied area of research and are the focus of the work in this dissertation.

One of the first uses of network side-channels as a measurement technique was the original Idle Scan proposed by Antirez [6] to scan for open ports on a target machine. Ensafi et al. [29] proposed another side-channel using the `SYN`-backlog for the same purpose. Subsequent work by Ensafi et al. [28] used this side-channel to provide a technique for measuring intentional blocking of ports by firewalls. Morbitzer [58] proposed a technique for an IPv6 Idle Scan. Zhang et al. used side-channels to detect machines hidden behind firewalls [82] and describe ONIS [83] for port scanning. Chen et al. [16] showed how the IPID field could be used as a side-channel to measure a multitude of information about a server including amount of internal traffic generated and the number of machines used for load-balancing. Bellovin describes a technique for counting the number of hosts behind a NAT [9] and Kohno et al. [47] made use of IPIDs to fingerprint remote devices. All of these works focus on studying connectivity measures (e.g., whether packets are being blocked between two machines or whether a machine is responding to packets on a given port) or measures that are not tied to client machine state changes (e.g., counting the number of machines or the number of packets sent via IPID changes).

Knockel and Crandall [45] discuss a technique for inferring the presence of active IPv4 connections on Linux machines using the IPID field and IP fragmentation reassembly behavior. Qian and Mao [66, 67] use firewall based network side-channels to infer TCP sequence numbers and perform off-path TCP/IP connection hijacking. Cao et al. [13, 14] use the Linux kernel's initial, unpatched challenge `ACK` behavior to implement an off-path TCP reset attack and a connection hijacking attack. Quach et al. [68] later scanned the Internet to discover how quickly popular web servers patched the vulnerability discovered by Cao et al.. Chen and Qian [17] describe a side-channel present in the IEEE 802.11 protocol that allows for off-path TCP injection attacks. In comparison to my own work, many of the above side-channel techniques require the use of malicious code running on the victim machine or fully exhaust shared, global resources (e.g., global packet rate limits or shared memory buffers such as the fragmentation cache).

IPIDs and IP fragmentation have been used as side-channels in many of the techniques described by Gilad and Herzberg. They explored the use of global IPID counters, fake congestion events, and packet processing delays as side-channels to infer the presence of traffic between hosts on the Tor network [33]. They also explored the use of IP fragmentation to perform off-path interception and denial-of-service attacks on machines behind a NAT or network tunnel [35]. A part of their attack involves inferring the Linux per-destination IPID counter. This technique requires the use of an agent running on either the client machine or another machine behind the NAT or tunnel running attacker code. The attack I describe does not require the use of an agent and can be done remotely and completely off-path. Gilad and Herzberg additionally discussed attacks which combined puppets and TCP/IP side-channels to attack the Same Origin Policy and carry out TCP injections [34].

Recently non-network side-channels have become an active area of security research. The most common of these side-channel attacks take advantage of timing side-channels in both software and hardware. The Spectre [46] and Meltdown [52] attacks disclosed in 2018 take advantage of timing side-channels to enable an attacker to read private data the attacker would otherwise be unable to access. Prior work to Spectre and Meltdown showed that timing side-channels could be used to build a wide-range of attacks. Song et al. [71] showed that keypress timings could be used as a side-channel to attack SSH implementations. Later work showed that timing attacks existed in a range of CPU caches and special purpose instructions [10,11,36,53,63,80]. Many of these side-channels were used to attack specific implementations of cryptographic algorithms and leak the key or other secret values. Compared with my work they commonly target side-channels present in computer hardware and not the software implementing the network stack.

Covert channels and covert channel analysis represent an area of research related to side-channels. A covert channel is a source of information transfer between processes which are not intended to communicate with each other, according to some security policy. While side-channels leak information about system state, and do not require collusion between actors, covert channels instead leak information about system objects (e.g., privileged files or databases) and require that the actors involved collude in order to leak the chosen information. An overview of the design of systematic covert system channel analysis [48, 51] can be found in the "light pink book" [50]. The goal of covert channel analysis is to design a general, systematic approach to identifing all possible covert channels within a computer system. Kemmerer [43] developed a technique to enumerate all covert channels in a system using a transitive closure between processes in a system and their ability to read/write objects. This work provided a general technique for enumerating covert channels in a system. However, it requires that all objects within a system be known. This presents difficulties as covert channels can often use variables that may not be viewed as objects. Wray [79] described a technique for enumerating covert channels when timing and clocks are considered.

Recent work [13, 28, 29, 35, 45] on finding and using network side-channels has focused on considering shared, limited resources in isolation. The work in this dissertation shows that, like covert channels, attempts to enumerate side-channels in system networking stacks must consider sets of behaviors that go beyond exhausting individual shared, limited resources.

My techniques make use of spoofed, also referred to as forged, source IP addresses. Reverse traceroute [42] utilizes spoofed source IP addresses to perform a reverse traceroute. That is, instead of the path to an end host, the return path from an end host is traced. PoiRoot [40] uses spoofed IP addresses to investigate Interdomain Path changes, while Flach et al. [30] use spoofed IP addresses to locate Destination-Based Forwarding rule violations. Many of the side-channel techniques discussed previously also use spoofed IP addresses [28, 29, 67, 81, 82].

Previous work to measure off-path round trip times has often involved access to multiple vantage points to act as one end host. IDMaps [31] takes this approach. iPlane [54] uses multiple vantage points to measure round trip times, along with packet loss and other useful network performance metrics. De A. Rocha et al. [7] present a technique for estimating delay and variance using forged return IP addresses. Cangialosi et al. [12] describe Ting for measuring latencies between arbitrary Tor nodes. King [37] and Queen [77] estimate off-path round trip time and off-path packet loss rate by using recursive DNS queries to DNS servers topologically near each end point. Due to this King, like my technique, does not require the use of multiple vantage points or direct access to either end host. In contrast my work does not require the use of additional machines, such as King's DNS servers, or protocols beyond TCP. Zhang et al. [81] discussed a technique for off-path RTT measurement, based on my own work described in Chapter 3. Their work references my own work and uses a more active form of probing compared to my technique that relies on the presence of a mathematical model of `SYN`-backlog behavior to improve RTT estimate accuracy.

Another approach for estimating round trip times is to provide a coordinate system in which end hosts can be placed. Once a host has been given a set of coordinates, round trip time estimates can be made based on the distance between two end hosts, based on their coordinates. Vivaldi [21] and GNP [59] use such coordinate systems. Hariri et al. [38] develop a distributed latency estimation system using a coordinate based approach while Zheng et al. [84] explore how routing policies on the Internet can affect the accuracy of such coordinate systems.

Network Tomography [15, 19, 76] is a method for inferring large-scale characteristics about the internal structure of a network using probes sent from edge hosts, on the outer edge of the network. Previous work by Tsang et al. [75] has shown that tomography can be used to measure network delay.

My work described in Chapter 5 includes an Internet scale "mass-scan" to study IPID field behavior to find potential operating systems that could contain IPID-based side-channels. Internet scale scanning is a recent area of study. The most widely known Internet scale scanning tool is ZMap [26], a tool capable of scanning for open ports over the entire IPv4 address space in approximately 45 minutes. Other mass-scanning tools include scanrand [41], masscan [1], unicornscan [49], and Shodan [57]. Gasser et al. [32] extended ZMap to support IPv6 addresses and developed techniques to efficiently scan the IPv6 Internet. Much like my own work, discussed in Chapter 5, these tools use a combination of asynchronous packet transmission and stateless scanning techniques to achieve the transmission speeds necessary to scan large networks in practical timespans.

While Internet scale mass scans taking place over minutes or hours, instead of days, are a relatively recent development large scale Internet studies have been previously carried out using slower tools. One of the first such studies was carried out by Paxson [61]. Paxson studied network behaviors and features by performing bi-directional network traces (i.e., Paxson traced and recorded network connections from host A to host B *and* host B to host A). Weaver et al. [78] carried out another large scale study on the presence and behavior of injected TCP `RST` packets. Their data contained over 30 million TCP flows gathered via passive network measurement. Heninger et al. [39] studied inadequate randomness in TLS certificates and SSH keys. Dainotti et al. [22] used passive measurements to study the IPv4 address space usage. Most recently Salutari et al. [70] studied IPID behavior in the IPv4 Internet. Their work focused on studying IPID behavior based on the four most widely referenced IPID algorithms (i.e., constant values, per-host counters, global counters, and random counters). In addition their work only scanned a portion of the entire IPv4 addresses space, one machine from each /24 CIDR subnet. My own work in this area, discussed in Chapter 5 makes no assumptions as to the number of different IPID behaviors and is more focused on which behaviors may contain side-channels rather than an overall study of general behavior, and was conducted by sending packets to all routable IPv4 addresses.

The work presented in this dissertation adds to the body of previous work on network side-channels. Specifically, my work focuses on techniques for measuring network features beyond simple connectivity at the IP layer without fully exhausting the shared, finite resources used as side-channels.

Chapter 3

Off-Path Round Trip Time Measurement

Round trip time measurements represent a fundamental network metric that can be used in a wide range of applications, such as: determining the fastest route, closest server, and some forms of IP geolocation. While direct measurement of round trip time is a simple task, it can only provide information about round trip times to a single end host along a single route. This places limitations on the extent to which large scale measurements can be performed, as it requires access to a large number of well distributed end hosts to provide sufficient network coverage.

In delay based IP geolocation techniques, such as those described by Gueye et al. [37] and GeoPing [60] , the round trip time is often used as a proxy for geographic distance. By measuring the round trip time between the target machine and a series of "landmark" machines whose geographic positions are known, it is possible to provide an estimate of the target machine's geographic location. Usually these systems require access to the "landmark" servers to perform round trip time measurements, which limits their ability to choose a sufficient number of appropriate "landmarks" to allow for accurate geolocation for all possible end hosts.

In this chapter, I introduce a novel technique for off-path round trip time estimation, based on the use of information side-channels in the modern Linux TCP/IP network stack, to infer the round trip time between two arbitrary end hosts. My technique requires the use of no machines beyond the measurement machine and the chosen end hosts and does not require the use of any protocols beyond standard TCP. The only requirements are that one end host be a Linux machine, running a kernel of version 2.6 or higher, with an open port, and that the second end host respond to unsolicited `SYN-ACK` packets with `RST` packets.

When clients want to connect to a server over TCP they first use a three-way handshake to initialize the connection. Any connections that have yet to complete this three-way handshake are referred to as "half-open" connections. Information related to these "half-open" connections are stored in a shared, finite data structure known as the `SYN`-backlog. Modern network stacks only store such connections for a limited amount of time, to avoid these entries exhausting the finite resources used to store these connections. Once this time limit has passed, these "half-open" connections are removed from the `SYN`-backlog to make space for new connection attempts. By manipulating the number of "half-open" connections in a given `SYN`-backlog, it is possible to use the behavior of the `SYN`-backlog to infer useful information about connections present in the backlog, including round trip time. When the state of the `SYN`-backlog changes the behavior of a Linux kernel's network stack changes. Instead of storing all "half-open" connections, the kernel will remove older entries to avoid complete resource exhaustion. My technique can detect this behavior change, use it to infer the state of the `SYN`-backlog and from there the round trip time of off-path routes without fully exhausting the `SYN`-backlog.

The contributions of this chapter are as follows:

- I describe a method for using information side-channels in the Linux TCP/IP network stack to infer the round trip time of off-path routes.
- I design and implement a measurement technique for using these side-channels to measure off-path round trip times between arbitrary Internet end hosts.
- I provide a detailed analysis and evaluation of the technique's accuracy across a range of routes and round trip time values. I discuss possible sources of error and how these can impact the accuracy of my measurements.

The rest of the chapter is structured as follows: In Section 3.1 I provide a brief overview of the behavior of the `SYN`-backlog in modern Linux kernels and discuss how this behavior leaks information that can be used to perform useful network measurements. In Section 3.2, I describe my technique for inferring off-path round trip time in depth. I then describe my experimental setup in Section 3.3 and discuss my results in Section 3.4. Section 3.5 addresses some ethical concerns and is followed by my conclusion in Section 3.6.

3.1 Background

Standard TCP connections are established via a "three-way handshake" as defined in RFC 793 [64]. This process consists of a total of three TCP packets being sent between each end point of a connection, in order to establish all the necessary state required for a TCP connection to be established. In the first step the client sends a TCP `SYN` packet to the server. Assuming the server is accepting connections, it responds with a TCP `SYN-ACK` packet. The client responds with a TCP `ACK` packet and can begin sending data.

Connections which have not completed the "three-way handshake" are stored in a data structure called the `SYN`-backlog. Connections stored in the backlog are considered to be "half-open" connections. The `SYN`-backlog plays an important role in ensuring the establishment of TCP connections. Until the "three-way handshake" is complete the TCP connection has not been fully initiated and data can not be sent between hosts. To ensure that "half-open" connections can be completed, various information about the connection must be stored, so that the "three-way handshake" can be successfully completed. This information is stored in the `SYN`-backlog. The `SYN`-backlog also provides a protection mechanism against packet loss during a TCP connection setup. If either the `SYN-ACK` or the `ACK` in the handshake is lost or dropped, then the presence of a `SYN` packet in the `SYN`-backlog allows for the retransmission of the `SYN-ACK`, in an attempt to allow the connection to be completed.

However, if the `SYN` packet is allowed to remain in the `SYN`-backlog indefinitely, then system resources will be exhausted storing connection information that may never be used. In order to prevent these "half-open" connections from fully exhausting available system resources, most modern network stacks only store "half-open" connections for a limited amount of time before they are removed from the `SYN`-backlog. In addition to storing these connections, network stacks will often resend `SYN-ACK` packets for which no `ACK` has been received. The number of retransmissions and the length of time for which a "half-open" connection remains in the `SYN`-backlog varies across TCP implementations.

My technique focuses on the `SYN`-backlog implementation used in Linux kernels of version 2.6 and higher. The kernel maintains a `SYN`-backlog, for each listening socket, whose length is chosen when the socket starts listening for new connection attempts. The kernel maintains a threshold for the number of retransmission attempts that will be made for each `SYN-ACK` that has not received an `ACK` packet as a response. This threshold is set to five as a default value. The Linux kernel distinguishes between "young" connections (i.e., connections for which there have been no `SYN-`

`ACK` retransmissions) and "mature" connections, for which at least one retransmission has been sent.

The kernel attempts to ensure that at least half of the `SYN`-backlog is available for "young" connections. If the `SYN`-backlog is at least half full then the kernel will attempt to remove "mature" connections, based on the number of retransmission attempts that have been sent, until the `SYN`-backlog's current size drops below half capacity. When attempting to remove "mature" connections the kernel will temporarily compute a new threshold value, depending on the number of "young" attempts present in the `SYN`-backlog. The more "young" connections that are present the larger this threshold value will be. Any "mature" connections that have sent a total number of retransmissions greater than this threshold will be removed, until at least half of the `SYN`-backlog is free for new connection attempts.

The behavior of removing "mature" connections that have not yet retransmitted the full five times provides an information side-channel that allows `SYN`-backlog information state to be leaked. If a "half-open" connection does not elicit the full five retransmission attempts, then I know the `SYN`-backlog was filled beyond half, leaking information about the state of the `SYN`-backlog. Note that causing and detecting state changes in the `SYN`-backlog does not require that I fully fill the `SYN`-backlog and exhaust this shared, finite resource. In order to cause the desired state change, the `SYN`-backlog only needs to be filled to half capacity, avoiding complete resource exhaustion.

3.2 Implementation

I designed my system based on two observations about the behavior of the `SYN`-backlog in the Linux kernel and how this behavior changes when the `SYN`-backlog state changes. The first observation is that the ratio of "young" connections to "mature" connections can be influenced by an off-path host. If a host sends a large amount of `SYN` packets, each using different sequence numbers and source ports, these will each be stored as separate `SYN`-backlog entries. By never completing the TCP three-way handshake (i.e: never sending an `ACK` in response to any `SYN-ACK`s), it is possible to ensure that after a few seconds the `SYN`-backlog will contain mostly "mature" entries. If enough entries are sent then the `SYN`-backlog will be filled beyond the halfway mark with "mature" entries, causing the kernel to remove entries which have been retransmitted too many times. This is done in order to ensure there is sufficient space in the `SYN`-backlog for newer "young" entries.

The second observation is that it is possible to influence the threshold value for which connection attempts are removed by the `SYN`-backlog when more than half the `SYN`-backlog is taken up by "mature" connections. Since the kernel determines this value based on the number of "young" connections, a `SYN`-backlog that has mostly "mature" connections and very few "young" connections will use a low threshold value when determining which connections to remove from the `SYN`-backlog. By filling the `SYN`-backlog with "half-open" connections and waiting until all these connections have received a number of retransmissions greater than the threshold, I can ensure that some of these connections will be removed if the `SYN`-backlog is then filled past the half way point.

As previously mentioned there are a few requirements that must be met in order for the measurements to be performed. First, one of the end hosts, referred to as the "Server" in the chapter, must be running a modern Linux kernel and have an open TCP port. Second, the other end host, referred to as the "Client", must respond to unsolicited `SYN-ACK` packets with a `RST` packet. Round trip time measurements can then be carried out by an off-path machine, referred to as the "Measurement Machine".

Overall my technique consists of a binary search over a range of possible round

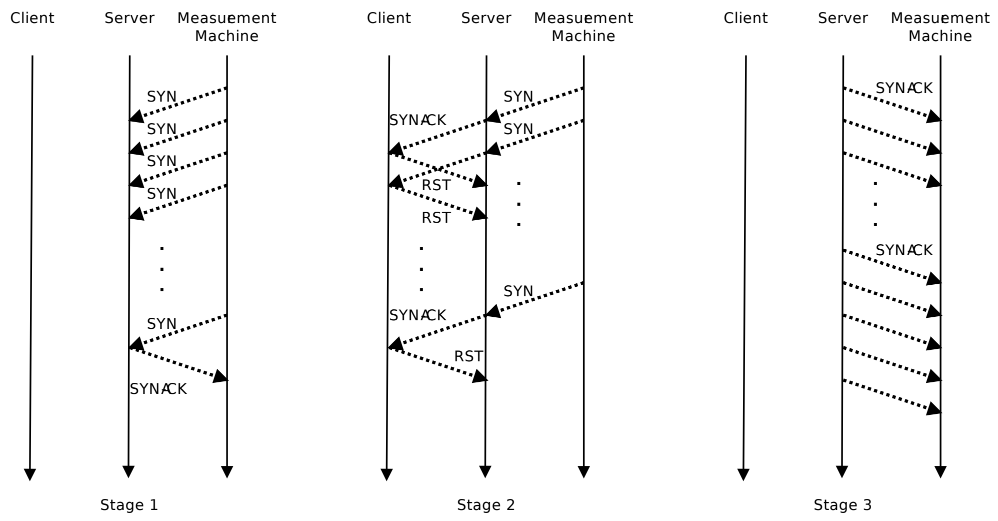


Figure 3.1: Three stages of my scan. Stage 1 partially fills the `SYN`-backlog of the Server with `SYN` packets from the Measurement Machine. Stage 2 sends `SYN` to the Server which appear to come from the Client. These trigger `SYN-ACK`s which are then reset. In Stage 3 the number of retransmissions for each `SYN` sent in Stage 1 is tallied and a determination is made about the chosen RTT estimate.

trip time values. Each round of the binary search consists of three stages, shown in Figure 3.1. These stages:

- Establish the necessary conditions for use of the information side-channel.
- Attempt to cause a change in the state of the `SYN`-backlog
- Infer whether or not the desired state change occurred.

Depending on whether or not the desired state change occurred, I can then determine how to adjust the upper and lower bounds used for the next iteration in the binary search.

3.2.1 Determining Scan Parameters

Before the round trip time between two hosts can be measured some parameters about the Server must be determined. These parameters are the size of the Server machine's `SYN`-backlog and the total amount of time taken for the Server to send all five `SYN-ACK` retransmissions to received `SYN` packets.

Determining Server `SYN`-backlog Size

To determine the Server's `SYN`-backlog size a technique similar to the full scan described below can be used. The Linux kernel places a minimum threshold on the size of the `SYN`-backlog of eight. Based on this, an attempt to determine the backlog size can proceed as follows: First, I set the estimated `SYN`-backlog size to eight and then send half the estimated size minus one unique `SYN` packets to the Server. Second, wait until two retransmissions have been seen for each `SYN` packet and then send two more unique `SYN` packets to the Server. Third, see if any of the first round of `SYN` packets do not retransmit `SYN-ACK`s. If all `SYN` packets continue to receive `SYN-ACK`s then the estimated `SYN`-backlog size is too small, otherwise it is too big. If the estimate is too small, double the estimate and repeat the previous steps. If the estimate is too large, carry out a binary search using the estimate as an upper bound and half the estimate as the lower bound.

Determining Server Retransmission Timing

Measuring the amount of time taken by the Server to send all five `SYN-ACK` retransmissions is a simple process. The Measurement Machine sends a `SYN` packet to the Server and then waits until it sees five retransmissions of the `SYN-ACK` sent in response to the initial `SYN` packet. In determining how long to wait between each retransmission attempt, the Linux kernel normally waits twice the amount of time it waited on the previous retransmission. As an example, if the kernel waits one second before sending a retransmission, it will wait two seconds after sending the first before sending then second, then four, eight and so on until five retransmission have been sent. After five retransmissions have been sent, the kernel waits for another retransmission interval before removing the initial `SYN` packet from the `SYN`-backlog. Given this behavior, once the fifth retransmission is seen the amount of time between the first `SYN-ACK`'s arrival and the fifth retransmission provides the Server retransmission timing.

3.2.2 Scanning the Target Machine

Before starting each binary search iteration, a new round trip time estimate ($eRTT$) is chosen as the midpoint between an upper and lower bound determined for the actual RTT. Each iteration will attempt to determine if the actual RTT is greater than, or less than the chosen estimate. In practice I use 0.0 as a starting lower bound and the time interval between the first `SYN-ACK` retransmission as the upper bound. For most Linux systems this time interval is one second by default. However, to measure longer round trip times a larger upper bound would be chosen.

In the first stage of my technique, the Measurement Machine sends $\frac{n}{2} - 7$ `SYN` packets to the Server to partially fill the `SYN`-backlog, where n is the size of the `SYN`-backlog for the open socket on the Server. This amount of `SYN` packets was chosen to ensure that I do not fill the `SYN`-backlog beyond halfway, which would fully exhaust this shared, finite resource, and to provide a small amount of padding space to allow other connection attempts that are not a part of the measurement to be completed without impacting my measurements. The Measurement Machine is programmed not to respond to any `SYN-ACK`s sent in response to these `SYN`s, which causes each of these SYNs to be treated as "half-open" connections by the Server. Each SYN uses a different sequence number and source port to ensure they are each treated as new, unique connection attempts. The Measurement Machine then waits until it has received two retransmission attempts from each SYN packet, before moving on to Stage 2.

In Stage 2, the Measurement Machine sends $\frac{6}{eRTT}$ SYN packets per second to the Server for ten seconds. $eRTT$ is measured in seconds. Each of these packets uses a spoofed source IP address to appear as if sent from the Client. All spoofed packets also use different sequence numbers and ports so that they are all treated as separate connection attempts. Each of the SYNs will cause the Server to send a SYN-ACK to the Client. Once the Client receives these SYN-ACKs it will determine that it sent no accompanying SYN packet and respond with a RST packet. Once these RST packets arrive at the Server they will cause the corresponding SYN packets to be removed from the SYN-backlog. After the Measurement Machine has finished sending at the determined rate for ten seconds the process moves on to the third stage.

In Stage 3, the Measurement Machine waits and records the number of SYN-ACK retransmissions it received for each of the SYN packets that were sent in Stage 1. Modern Linux kernel s will make five retransmission attempts for each "half-open" connection before removing a SYN packet from the SYN-backlog. For the default kernel configuration, these attempts will occur at 1, 2, 4, 8, and 16 second intervals after the previous SYN-ACK was sent, with the packet being removed from the SYN-backlog 32 seconds after the fifth SYN-ACK is sent. That is after the first SYN-ACK is sent the Server waits one second and if no ACK has been received the SYN-ACK is retransmitted. If after 2 seconds no ACK has been received another SYN-ACK is retransmitted and so on, until 5 retransmissions have been sent. I have also seen these 5 retransmission attempts being sent at 3, 6, 12, 24, and 48 second intervals after the previous SYN-ACK, with the kernel waiting 96 seconds after the fifth and final retransmission before the `SYN` is removed from the `SYN`-backlog. The exact timing of a given machine can be determined before any measurement attempt by sending a single `SYN` and determining the interval between subsequent retransmission attempts.

After Stage 3, it is possible to determine if the chosen round trip time estimate is greater than or less than the actual round trip time between the two end hosts. If the chosen estimate is less than the actual round trip time, then the sent packets will begin to accumulate in the `SYN`-backlog, as new packets will arrive before previous packets are removed due to the Client sending `RST` packets in response to the Server's `SYN-ACK` packets. This will cause the `SYN`-backlog's size to exceed half of its maximum capacity. This causes the kernel to remove "mature" entries until the `SYN`-backlog size is below half the maximum capacity. Any entries that are removed will stop retransmitting `SYN-ACK`s. If the estimated round trip time is greater than the actual round trip time, each packet will be reset before the next arrives and the `SYN`-backlog will never contain enough packets to cause the kernel to try and remove any "mature" connection attempts. Using this information I can continue my binary search, adjusting my round trip time estimate accordingly. My technique then continues iterating over possible ranges of round trip times until the scan has provided an accurate bound on the round trip time.

3.2.3 An Example Iteration

Consider an off-path measurement being performed between a Server, S, and a Client, C, by a Measurement Machine, M. Assume that the open socket on S has a `SYN`-backlog size of 256. The route between S and C has an actual round trip time of $60ms$ and previous iterations of the binary search have determined that the next estimate should be in between $50ms$ and $100ms$. First, a new estimate is determined to be 75ms, the midpoint between 50 and 100. In Stage 1 M sends $\frac{256}{2} - 7 = 121$ `SYN` packets to S and does not respond to any synacks sent in response. After receiving two retransmission attempts per `SYN` packet Stage 2 begins.

M then sends $\frac{6}{0.075} = 80$ `SYN` packets per second for ten seconds to S with the spoofed source IP address of C, since the midpoint time is 75ms. S will respond to each `SYN` by sending `SYN-ACK`s to C. Since C did not send any `SYN` packets it responds to these `SYN-ACK`s with `RST` packets. Once these `RST`s arrive at S the corresponding `SYN` packets are removed from the `SYN`-backlog. Based on my previous choice of packet rate, S receives six `SYN` packets for every estimated round trip time between itself and C. However, because the chosen estimate, 75ms, is greater than the actual round trip time of 60ms each `SYN` in this group of six will have already been removed before the next group of six arrive. Because of this the `SYN`-backlog will never fill beyond 127, which is less than the value required to remove "mature" connections from the `SYN`-backlog.

Finally, in Stage 3, M will receive 5 retransmissions for each `SYN` that was sent in Stage 1 since the removal behavior was never triggered. M will then conclude that the chosen estimate 75ms is too large and determine the bounds for the next binary search iteration to be a lower bound of 50ms and an upper bound of 75ms.

3.3 Experimental Setup

To test the effectiveness of my technique I compared my technique with actual round trip time measurements between end hosts on the Internet. I used a set of 15 PlanetLab [18] nodes: 3 from each of North America, South America, Europe, Asia, and Australia/New Zealand, as the Server end host. This group was chosen to provide a distributed set of Server hosts, on most continents. Each PlanetLab node runs a simple server program which creates an open port, for use by my scan, with a backlog size of 256. Each PlanetLab node was running a Linux kernel of either version 2.6.X or 2.7.X, which met my requirements for the Server end host.

These PlanetLab nodes did not provide any additional assistance to my Measurement Machine during any part of my experiment. These nodes only provided an open port, provided access to ground-truth round trip time measurements to compare my estimates to, and performed occasional traceroutes to provide information about any potential routing changes that may have influenced my results. As a result all my round trip time measurements were performed off-path, from the Measurement Machine's point of view. Actual round trip time values for each measurement were gathered from direct packet captures of the off-path scan being performed. **In other words, the exact same `SYN-ACK` packets and `RST` responses between the Server and Client were used for both direct and indirect measurements, so that any error in my indirect measurements can be attributed to my technique and not other factors.** This provided me with the actual round trip time that each off-path scan was measuring. Note, that when I say I measured the round-trip time, what I measured is the average round trip time over the entire length of the experiment. So my ground truth is based on this average.

The Measurement Machine was a machine running Ubuntu Linux 12.04 LTS with Linux kernel version 3.5.0 with a direct connection to the local Internet backbone to avoid any local filtering or firewalls.

Client end hosts were chosen by IP address, at random, from the entire publicly routable IPv4 address space. For each randomly chosen Client IP address, the Measurement Machine checked to see if the host met my requirements for a Client end host, that is that the host responded to unsolicited `SYN-ACK`s with a `RST` packet. During my experiments I noted that some machines respond with packets that had only the `RST` flag set while others responded with both the `RST` and `ACK` flags set. Both of these packets resulted in the desired Server `SYN`-backlog behavior and Client machines using either were used.

Once a Client IP address was found that met my requirements, an available PlanetLab node from my set of 15 was chosen at random to act as the Server end host for my technique. The Measurement machine then carried out my round trip time measurement technique, using the two chosen end hosts. During the scan, the chosen PlanetLab node captured all traffic being sent to the open port it had created and ran periodic TCP traceroutes, to record any route changes that occurred during the course of the scan. The Measurement Machine also captured all traffic between itself and the Server. Each measurement scan took between five and ten minutes to complete. During my measurements the Measurement Machine made no contact with any Client machines, beyond checking to ensure that they met my requirements.

3.4 Analysis

To evaluate the accuracy of my technique I collected data over a two week period, from July 14, 2014 through July 28, 2014. During this time span I collected 616 round trip time estimates. Each scan measured both the actual round trip time and the round trip time determined by my off-path technique, as measured by the Measurement Machine. Figure 3.2 shows a CDF plot of the ratio between the actual round trip time, R, and the off-path estimate, E, E/R.

If my measurements were perfect then the ratio E/R would be 1.0 for each measurement estimate. Any ratio that is less than 1.0 indicates an underestimated round trip time, a ratio greater than 1.0 indicates an overestimated round trip time. Overall, I found that approximately 60% of the round trip times measured in my dataset resulted in ratios of 1.0 or lower, providing a lower bound on the round trip time in these cases. Over 60% of scans in my dataset were within 10% of the actual round trip time measured, with over 80% of my off-path measurements falling within

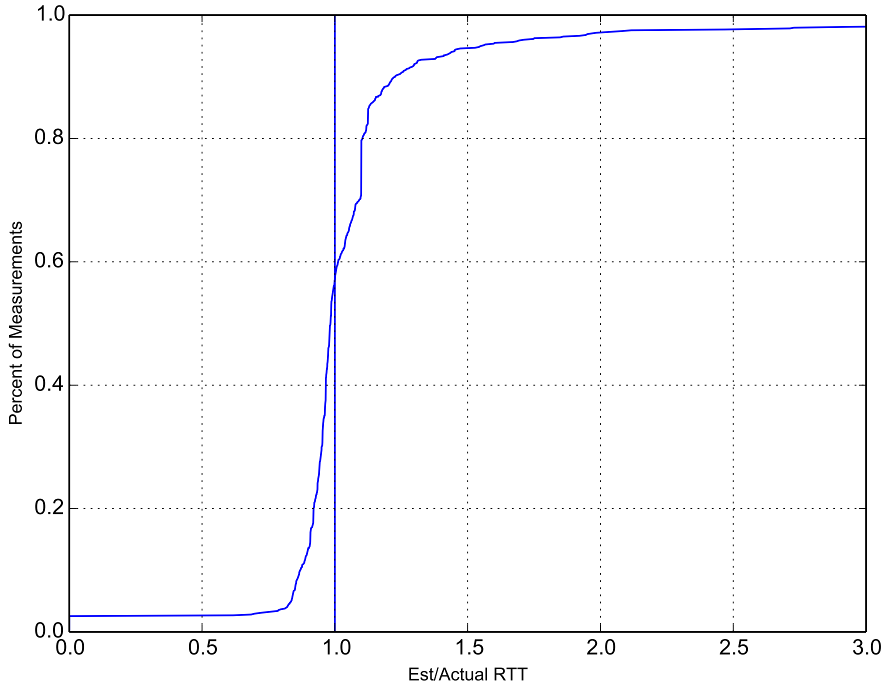


Figure 3.2: Accuracy of round trip time estimates.

20% of their actual round trip times. Table 3.1 summarizes my results.

3.4.1 Effect of Packet Loss

One prominent source of error that affects the accuracy of my technique is packet loss between the Server and Client. In cases where there is noticeable packet loss, which I define as greater than 5%, between the Server and the Client, my measurements often differ from the actual round trip time by 25% or more. These large error values are the result of packet loss which impacts the accuracy of my off-path measurements.

Consider the case where a `SYN-ACK` sent from the Server to the Client as a result of a spoofed `SYN` is lost. In this case the Client will never send a `RST` packet and

RTT Range	Within 10%		Within 20%	
	Packet Loss	No Packet Loss	Packet Loss	No Packet Loss
$[25\text{ms}, \infty)$	63.6%	72.1%	83.7%	92.7%
$[100\text{ms}, \infty)$	67.1%	75.9%	87.2%	96.0%
$[0\text{ms}, 25\text{ms}]$	18.0%	16.0%	46.1%	60.0%
$[0\text{ms}, 100\text{ms}]$	35.5%	39.3%	58.06%	69.05%
$[25\text{ms}, 100\text{ms}]$	43.5%	49.3%	63.5%	72.9%
Overall	60.7%	68.87%	81.33%	90.84%

Table 3.1: Percent of measurements within a given percent of actual round trip time.

the spoofed `SYN` will remain the Server's `SYN`-backlog longer than was expected. If enough `SYN-ACK`s are lost in transit the `SYN`-backlog will fill beyond halfway, causing the kernel to evict "half-open" connections as previously described. My technique measures this eviction as an indication that the current round trip time estimate is too high resulting in an incorrect measurement. A similar situation occurs if the `RST` sent by the Client in response to a `SYN-ACK` sent by the Server is lost.

Figure 3.3 compares my overall estimates with those that had no packet loss. For measurements with a packet loss rate of 5% or less, a lower bound is provided for 64% of paths. Over 65% of measurements are within 10% of the actual round trip time, while over 85% are within 20%. When considering only those measurements with no packet loss, a lower bound is found in over 65% of measurements. Over 68% of estimates are within 10% of the actual round trip time and over 90% are within 20%. Compared to King [37] which measures round trip times to within 10% error for roughly two-thirds of chosen routes and upwards of three-quarters of routes within 20% error, my technique provides similar accuracy with packet loss and improved accuracy without packet loss.

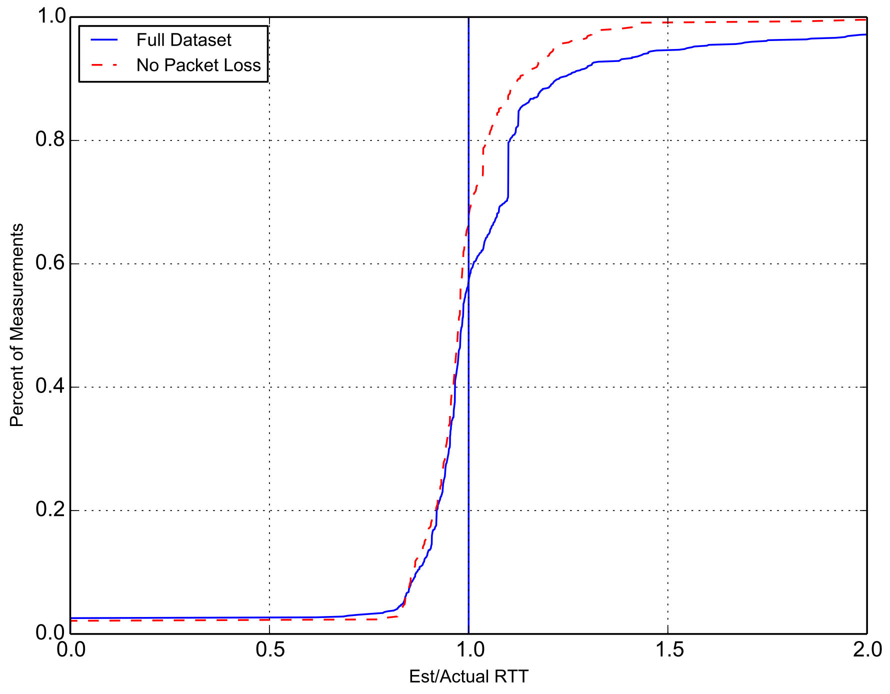


Figure 3.3: Effect of packet loss on measurement accuracy.

3.4.2 Accuracy when Actual Round Trip Time is Small

To further explore the accuracy of my measurements, I considered situations in which specific network characteristics are present. In cases where the round trip time I attempt to measure is small I found that my technique performed quite poorly. I considered actual round trip times of 25ms or less as small. When I considered routes with actual round trip times less than 25ms, my estimate represents a lower bound in 43.6% of cases. Only 18% are within 10% of the actual round trip time and 46.1% are within 20%. As I previously discussed, high packet loss rates can have a significant impact on the accuracy of my measurements. When considering small round trip time routes with no packet loss, a lower bound is found in 52% of measurements, with 16% and 60% falling within 10 and 20 percent of the actual round trip time as

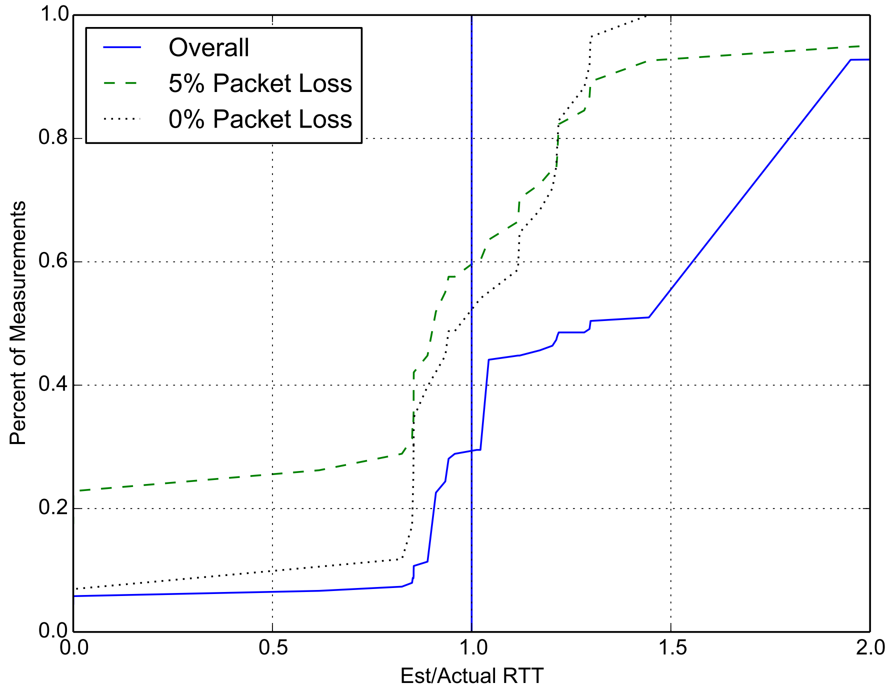


Figure 3.4: Accuracy of round trip time estimates with actual RTT less than 25ms.

shown in Figure 3.4.

I next considered my accuracy when excluding low round trip time routes. For all routes with round trip times greater than 25ms a lower bound is found for 60.83% of routes, with 63.6% being measured within 10% of the actual round trip time, and 83.7% measured within 20%. Figure 3.5 shows these results. Excluding routes that experienced any packet loss, 70.6% of measurement provided a lower bound, 72.1% were within 10% of the actual round trip time, and 92.7% were within 20%.

One possible explanation for this drop in accuracy is that, as the round trip time I measured gets smaller, the impact that small variations in the round trip time have on my measurements are magnified. When dealing with round trip times below 25ms, a difference of a few milliseconds can affect the rate at which my technique fills

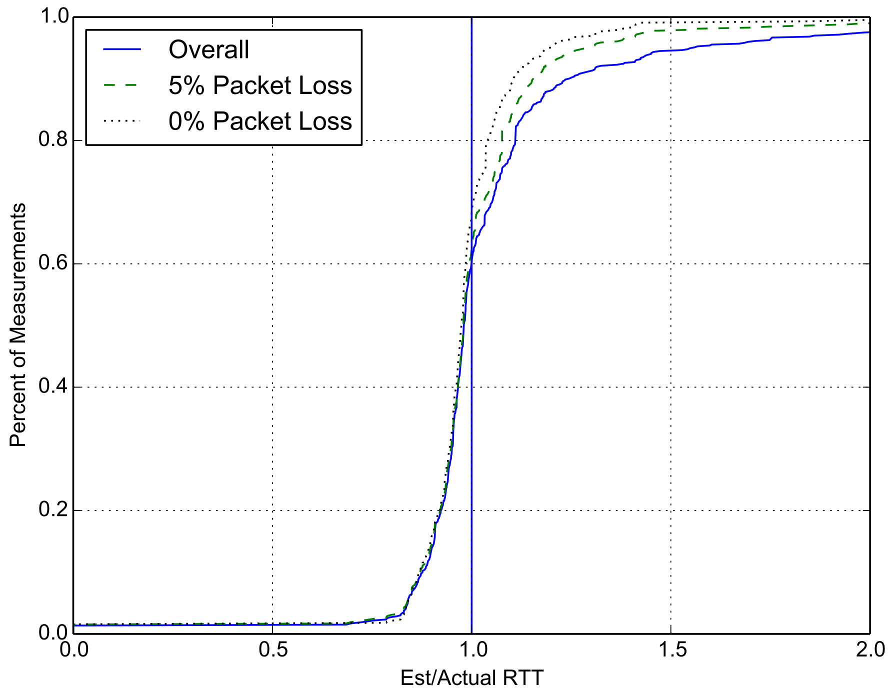


Figure 3.5: Accuracy of round trip time estimates with actual RTT greater than 25ms.

the `SYN`-backlog. If the backlog fills faster than expected as a result of these variances this could cause packets to be removed when they should not be. This unexpected change in `SYN`-backlog behavior would cause incorrect binary search decisions, which results in the inaccuracy of my measurements for small round trip times.

3.4.3 Accuracy on Routes with Large Round Trip Times

Next, I considered the accuracy of my measurements on routes that have a large round trip time. I define a large round trip time to be greater than 100ms. For such routes 63.2% are lower bounded, 67.1% are within 10% of the actual round trip time, and 87.2% are within 20% of the actual round trip time. On routes that experience

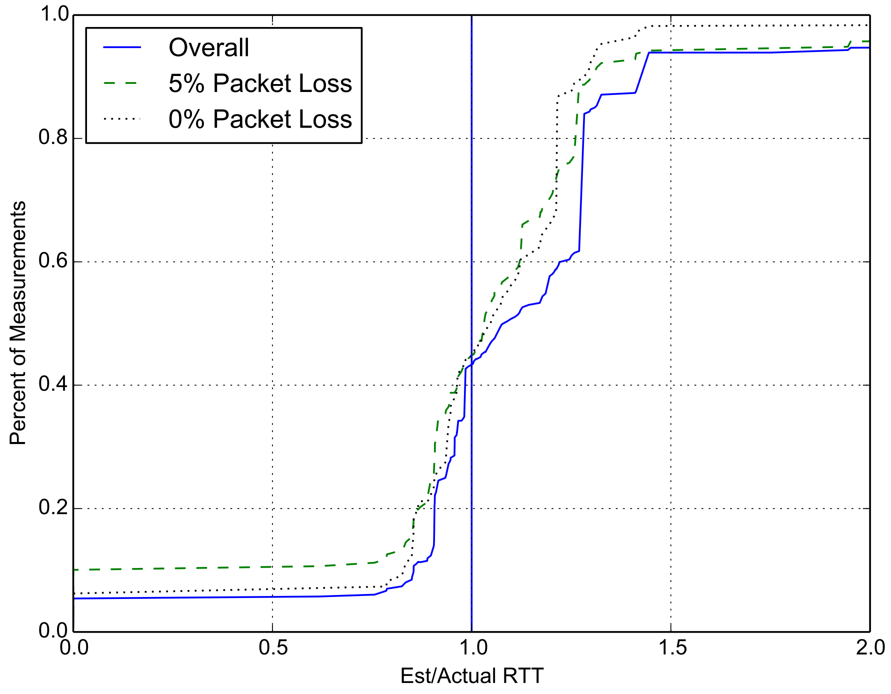


Figure 3.6: Accuracy of round trip time estimates with actual RTT less than 100ms.

no packet loss, my measurement technique determined an accurate lower bound for 73.6% of measurements and estimate the round trip time within 10% for 75.9% of routes and within 20% on 96% of routes. Figure 3.6 compares measurements of routes with round trip times less than 100ms, with and without packet loss, and Figure 3.7 compares routes with round trip times greater than 100ms. Figures 3.8 and 3.9 compare my overall results across various round trip times, with and without packet loss.

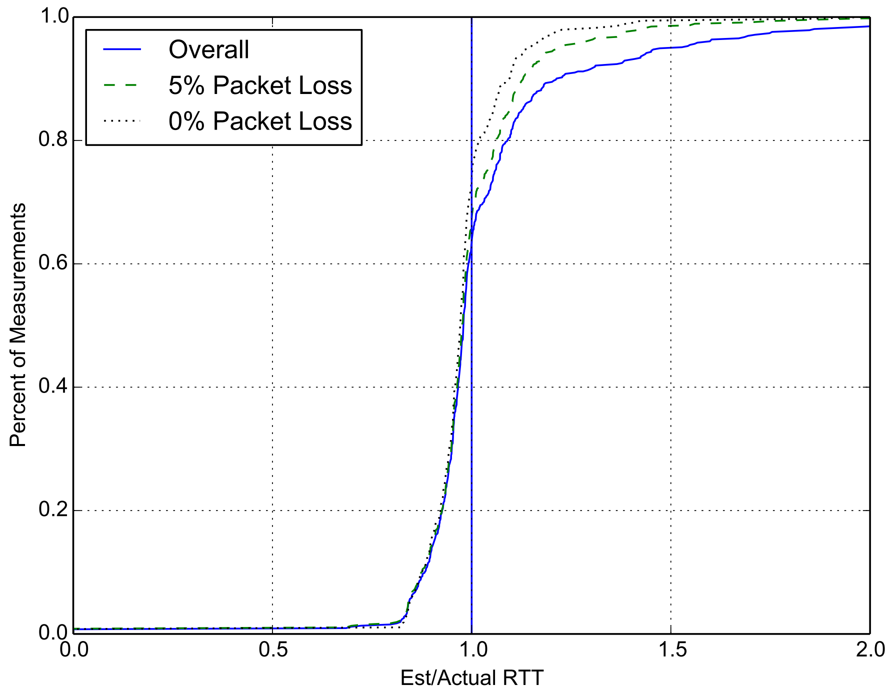


Figure 3.7: Accuracy of round trip time estimates with actual RTT greater than 100ms.

3.4.4 Effect of Large RTT Variations

Another source of error is the impact that variations in the round trip time between the chosen hosts have on the accuracy of my measurements. Consider a route between two hosts, where the round trip time varies between a high of 400ms and a low of 200ms. In one iteration of the binary search the round trip time is measured as 200ms and I adjust my bounds accordingly. However, during the next iteration the round trip time jumps to 400ms. This new value is then measured and the bounds are adjusted. If the round trip time then drops near 200ms for the next iteration I will have incorrectly adjusted my bounds closer to 400ms, when they should be closer to 200ms. In some cases, depending on the chosen bounds, such an error could

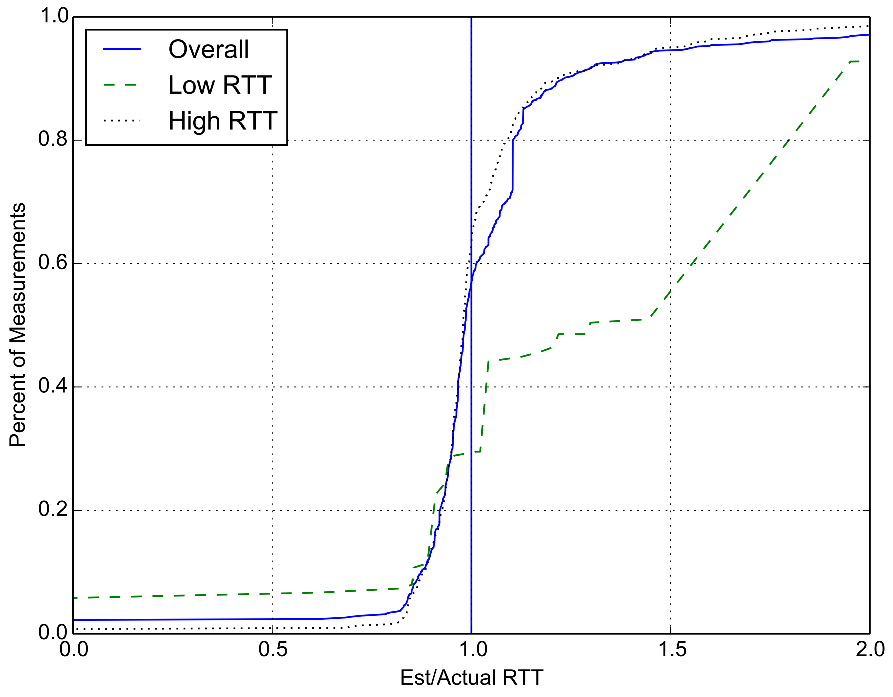


Figure 3.8: Comparison between overall dataset, lower RTT only, and high RTT only.

result in the actual round trip time no longer being within these new bounds, making accurate measurement impossible.

3.4.5 Effect of Route Changes

The final source of error I noted was changes to the route between the chosen end hosts. While minor changes to the route often had little impact on my final accuracy, in a few cases new routes resulted in different overall round trip times. If these differences in round trip time are large enough then some packets may arrive sooner or later than anticipated. If packets arrive too soon the `SYN`-backlog will not fill up and evict "mature" `SYN`s in cases where it should. However, if packets arrive too

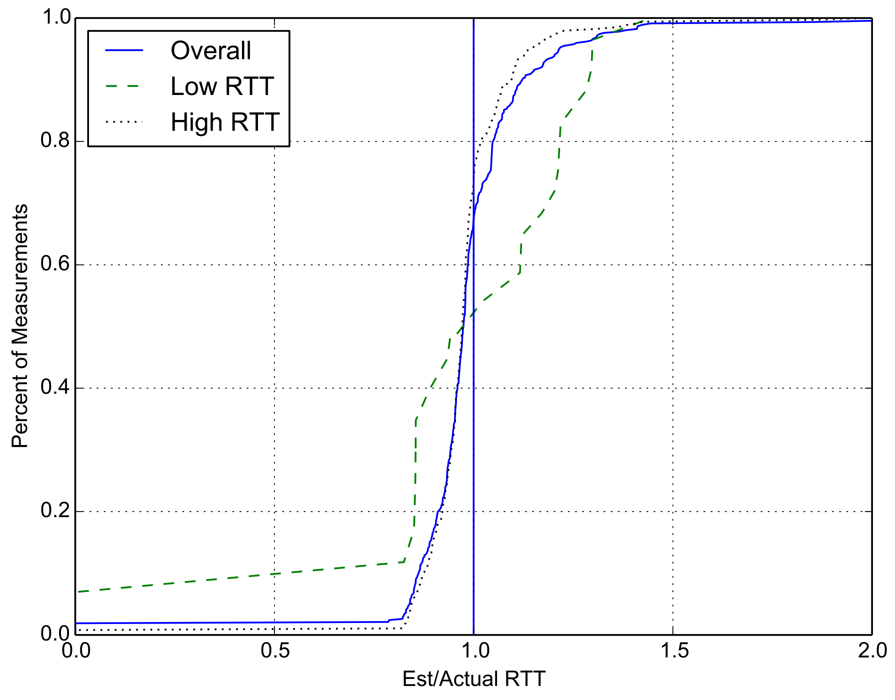


Figure 3.9: Comparison between overall dataset, lower RTT only, and high RTT only with no packet loss.

late then the backlog will fill in cases where it should not and cause packets to be evicted. Both cases can result in inaccurate measurements.

3.5 Ethical Considerations

When performing network measurement scans there are potential ethical issues that each scan must address and my scan is no exception. Every packet sent by a scan consumes resources for each scanned machine. These resources include network bandwidth, CPU cycles dealing with incoming packets, and time spent by system administrators determining if packets are a threat to the network. For my scan there are a few possible ethical issues that should be discussed and addressed.

The first is the use of the `SYN`-backlog as a side-channel. As part of my measurements I need to fill the `SYN`-backlog of the Server machine at least half full with `SYN` packets, whose connections will never be completed. This creates a drain of resources on the Server by causing the creation of “half-open” TCP connections for the duration of the scan. If my scan were to fill the backlog completely this might cause a denial-of-service if preventative measures are not taken by the Server. However, my scan should never fill the backlog beyond half full for more than one-fifth of a second, due to the behavior of the Linux kernel. When the backlog is half full the kernel will remove “mature” packets, including those that I sent to fill the backlog initially five times every second. Because of this behavior, which I exploit to perform my scan, my scan will never cause a denial-of-service due to filling the `SYN`-backlog with unfinished connection attempts. In this way my technique differs from previously discussed work, such as that of Knockel and Crandall [45] and Cao et al. [13], in that my technique does not exhaust the shared, finite resource used as a side-channel.

Another potential issue is that the act of partially filling the `SYN`-backlog over a short period of time will be incorrectly identified as the start of an attempted denial-of-service attack by a system administrator, firewall, or intrusion detection system. While I cannot guarantee that this will never happen as an initial reaction to my scan, further inspection of my network traffic will reveal that the transmission and creation of “half-open” TCP connections lasts a short time, typically only a few seconds, and is not continuous or large enough to cause sufficient resource allocation to result in a denial-of-service, as discussed above.

The third issue to be addressed is the amount of network traffic generated by my scan. When attempting to measure paths with small round trip time values, usually below 20ms, my scan can send upwards of 800 packets per second when attempting to trigger the `SYN`-backlog behavior I use to gather my data. Each of these packets is an unsolicited `SYN-ACK` packet which will be immediately reset by the Client on arrival resulting in very little system resources being used to process the packet since it is immediately reset without any further processing. Each packet is 60 bytes in size and at a rate of 800 packets per second would result in 48 kilobytes per second of network traffic, a level that modern systems and networks are more than capable of handling without noticeably affecting the quality of communication.

3.6 Conclusion

I have presented a novel technique for inferring the round trip time between two Internet end hosts, who meet a few simple behavioral requirements, using side-channels in the TCP/IP stack of modern Linux kernels. Specifically my technique uses a side-channel present in the kernel's `SYN`-backlog to infer off-path round trip times. Compared to similar techniques, my system does not require the use of any additional machines, beyond the end hosts and the scanning machine, nor the use of any protocols beyond standard TCP/IP. I only require that one end host have an open port and be running a Linux kernel of version 2.6 or higher and that the other end host respond to unsolicited `SYN-ACK`s with `RST` packets.

Unlike previous side-channel techniques for off-path measurement, my technique does require the use of a third party "proxy" machine to measure round trip time, nor does it fully exhaust the shared, finite resource that it exploits as a side-channel. In comparison to other techniques mine also measure network characteristics beyond simple measures, such as connectivity by measuring round trip time of communication between off-path machines. I accomplish this by causing state changes in the Linux kernel's `SYN`-backlog which result in measurable behavior differences that can be tied to characteristics of off-path Internet routes.

I have provided an evaluation of my technique "in the wild" and compared my estimates with actual round trip time measurements. I evaluated my technique across a range of round trip times and routes and discussed its strengths and weaknesses for specific route characteristics. I have shown that my technique is accurate and discussed possible causes of error that impact my measurement accuracy.

Chapter 4

Off-Path TCP Connection 4-Tuple Inference Attack

In this chapter, I describe a novel attack for detecting the presence of an active TCP connection between an arbitrary client and a remote Linux server using information side-channels present in the Linux kernel's implementation of global and per-connection IPv4 IPID values. The attack's requirements are as follows:

- A server that is a Linux machine running kernel version 4.0 or newer.
- Access to multiple IPv4 addresses to use as attacker addresses.

The proposed attack makes use of the Linux kernel's behavior of responding to "unsolicited" `SYN-ACK`s with a `RST`. This is the default Linux kernel behavior and is also described as the proper behavior for handling "unsolicited" `SYN-ACK`s in RFC 793 [64]. An "unsolicited" `SYN-ACK` is a `SYN-ACK` for which no `SYN` was sent and so does not represent a potential connection. The number of IPv4 addresses required for the attack to be reliable is at least multiple hundreds, with thousands increasing the probability that the attack can be carried out. While requiring this many may prove a hindrance to attackers with a small amount of resources it is well within the realm of possibility for large botnets or nation-state attackers.

By detecting when the Linux kernel changes from using one of its 2048 global IPID counters to using a per-connection TCP IPID counter, the attack I describe is able to infer the IP-port 4-tuple that corresponds to an active TCP connection without being an on-path observer. The IP-port 4-tuple representing an active TCP connection is the source address, source port, destination address, and destination port used for TCP communication.

My contributions are as follows:

- I describe a method for using a side-channel present in the Linux kernel's implementation of global and per-connection IPv4 packet IPIDs to infer an active connection's IP-port 4-tuple. Mine is the first such attack to infer the existence of a connection completely off-path without exhausting any global resource.
- I design and implement a proof-of-concept attack for using these side-channels to detect the presence of active connections between arbitrary Internet end hosts.
- I provide a detailed analysis and evaluation of the attack, analyze possible sources of error, and discuss possible mitigations.

A key novelty of the side-channel attack described in this chapter, compared to past work, is that it does not exhaust any global resource in detecting active connections. To the best of my knowledge, there are only two existing side-channel attacks in the literature where the existence of a TCP/IP connection could be inferred: Knockel and Crandall [45] , where the global fragment cache was filled and Cao et al. [13], where a global challenge `ACK` rate limit was reached. Note that I exclude attacks that require malicious code on the victim machine or an attacker machine behind the same NAT as the victim [17, 33–35, 67]. My attack uses a per-destination (i.e., not global) duplicate `ACK` limit for one non-default corner case that I encountered, but is otherwise based on inferring *which* resource is being used rather than exhausting a specific resource. This is a major conceptual difference that challenges the notion that there is a direct one-to-one connection between shared, limited resources and non-trivial network side-channels. While past side-channels have had the property of not exhausting global shared, limited resources, such as Antirez's idle scan [6] for detecting open ports, to date such side-channels have been relatively trivial and could not reveal information about active connections. What the attack presented in this chapter demonstrates is that simply enumerating globally shared resources (rate limits, buffers, caches, etc.) and then considering each in isolation is not sufficient for enumerating all possible side-channels that can be used to infer a connection.

The rest of the chapter is structured as follows: Section 4.1 discusses scenarios that motivated this chapter. Section 4.2 reviews what an IPID value is, how the Linux kernel generates IPIDs, what a challenge `ACK` is, and how the Linux kernel handles challenge `ACK`s. Section 4.3 discusses the methods for using IPIDs and challenge `ACK`s as side-channels to detect the presence of an active TCP connection. Section 4.4 describes my experimental methodology. Then, I discuss my results in testing the attack in Section 4.5. In Section 4.6 I discuss the applicability of the attack, the challenges it faces "in the wild", common sources of error, and possible mitigations. I finish with my conclusions in Section 4.7.

4.1 Motivation

One common assumption made by many privacy tools using the TCP protocol is that information about the state of an existing connection does not leak outside of the connection itself. This includes information about whether or not a connection exists. Many privacy and censorship circumvention tools rely on this to ensure that this information could only be discovered by an *on-path* attacker. If an attacker were able to detect the existence of a connection between a client and a circumvention tool off-path it, could allow the attacker the possibility of deanonymizing a client, detecting a hidden service, or other attack vectors.

One scenario where the ability to detect off-path connections is useful is the case of a user accessing a sensitive website via a Tor [23] bridge, which is a type of relay that is supposed to be unknown to the censor. The attacker may suspect that the user is connecting to a bridge and could try to confirm this suspicion. While there has been evidence of nation-states using active probing to identify such hidden machines [27], obfuscation protocols such as obfs4 [5] can be used to impede such probing. Using an attack that could detect an off-path TCP connection an attacker could attempt to detect a TCP connection between a suspected Tor bridge and a Tor directory server after a user opens a connection to the Tor bridge. Since this would detect the connection it would not require active probing that could be impeded by obfs4 or similar mitigations. Note that once the connection is open the distinction between client and server is interchangeable for the attack I present. Also note that six out of 10 Tor directory servers are dual stack [72] allowing an attacker to use both IPv4 and IPv6 address when attempting to find IPID hash collisions. As multiple IPv6 addresses are often assigned to a single machine or network, compared to IPv4 addresses, these additional IPv6 addresses provide attackers with a much large pool of addresses that could be used to find IPID hash collision. While the attack I present focuses on a simple IPv4 only implementation, there are many different variations on the attack to make it practical for any given application.

Generally, the attack I describe in this chapter provides the attacker with a primitive for inferring the existence of connections off-path, which violates assumptions often made by privacy tools. I focus my experimental methodology on understanding the base accuracy and speed of the attack on one client/server pair in isolation, whereas a real attacker may have additional flexibility in carrying out the attack and can use an improved implementation and/or different tradeoffs. Thus, while the attack as presented in this chapter assumes that the connection persists for roughly two minutes, and sometimes fails, I establish that the basic attack primitive exists. A real attacker may implement it differently. For example, for the aforementioned application of detecting Tor bridges the attacker may look for collisions with any of a large set of Tor directory servers and guard nodes, all of which Tor bridges are likely to make persistent connections to, if they are heavily used. The attacker may repeat attacks for identified connections to avoid false positives, and may be able to tolerate false negatives because even if a Tor user's connection is interrupted only half the time or only after limit use, their quality of service is diminished and they are likely to use other services that are more reliable (such as government-sponsored VPNs). And, even if the attack is mitigated (e.g., by filtering out the attack traffic) the damage may already be done in terms of user trust in a given tool's availability.

4.2 Background

The attack relies on side-channels in the Linux kernel's handling of IPv4 IPID values as a mix of global and per-connection counters.

4.2.1 IPIDs

IPv4 packet headers contain a 16-bit identification field known as the *IP Identifier* (IPID). During the course of transmission it is possible that a given IPv4 packet may be too large to transmit over a given link. In such cases the packet can be broken in smaller packets known as *fragments*, which all retain the original packet's IPID. Once these fragments reach their final destination, the receiving machine uses the IPID value of each fragment to determine how to correctly reassemble the fragments to rebuild the initial IPv4 packet.

IPv6 packet headers do not contain an IPID field. Instead, when they are fragmented an IPv6 extension header is added containing a fragment ID value which functions similarly to an IPv4 IPID value. Fragmenting IPv6 packets is never performed by routers, with hosts relying on *Path MTU*(PMTU) Discovery to determine the largest packet size and only send packets of that size or smaller. IPv4 can also use PMTU Discovery but this is not always enabled. In this chapter I will focus on IPv4 as it is always guaranteed to have an IPID field present.

4.2.2 Linux Kernel IPv4 IPID Values

Early network stack implementations often used a global IPID counter that was incremented for each packet sent. However, work on *idle scans* and similar techniques [6,28,29] exploited information side-channels in the global IPID field to make measurements of off-path machines. This led to the Linux kernelmoving to the adoption of per-destination IPID counters [24]. However, this technique has since been removed in favor of a mixed approach. This new approach consists of a set of 2048 separate IPID counters. Each connection is assigned a counter to use based on a hash of the source and destination IP addresses, the protocol number of the IPv4 packet (e.g., TCP, UDP, ICMP), and a random value generated on system boot.

Research by Knockel and Crandall [45] showed that simple, incrementing, per-destination IPID counters made it possible to use the IPID field to count the number of packets sent between two machines for UDP and ICMP, and infer the existence of a TCP connection, completely off-path. Per-destination IPID counters were already being phased out in an experimental version of the kernel because they were stored in a global resource called the inet peer table that could be exhausted [24], leading to performance and security problems (because when peers were evicted they reverted back to a predictable IPID). The global resource that Knockel and Crandall exhausted to infer IPIDs off-path was the IP fragment cache. In response to Knockel and Crandall the 2048 separate IPID counters strategy that had been under testing was released early, with the addition of random noise [25] and hashing of source address and protocol number (in addition to destination address and network secret). These changes were made after some discussion about if the new IPID strategy were more resistant to off-path attacks than the old. For Linux's current IPID implementation, every time a packet is sent the chosen counter (among the 2048) is then incremented by a random value, chosen from a uniform distribution between one and the number of system ticks (typically milliseconds) since the last packet transmission that used the same counter.

4.2.3 IPv4 Do Not Fragment Behavior

While the above IPID behavior is used in most cases, there exists a special case in the kernel's handling of TCP connections. As discussed previously in Section 4.2.1, IPIDs are used to assist in reassembling *fragmented* IPv4 packets. However, if a machine is set to use Path MTU Discovery, it will attempt to find the largest packet size a given route can handle and attempt to only send packets of this size or smaller. This is done to avoid fragmenting packets during transmission. PMTU Discovery changes how the Linux kernel chooses IPIDs when sending TCP packets.

```
393 void ip_select_ident_segs(...) {
        ...
398     if ((iph->frag_off & htons(IP_DF)) && !skb->ignore_df) {
            ...
404         if (sk && inet_sk(sk)->inet_daddr) {
405             iph->id = htons(inet_sk(sk)->inet_id);
406             inet_sk(sk)->net_id += segs;
407         } else {
408             iph->id = 0;
409         }
410     } else {
411         __ip_select_ident(net, iph, segs);
412     }
413 }
```

Figure 4.1: Linux kernel IPID selection.

When PMTU Discovery is active the Linux kernel does not pick an IPID from one of the 2048 counters, discussed in Section 4.2.2, when using TCP and the *Do Not Fragment* flag is set. Instead, the kernel picks from a *per-socket* IPID counter unique to each TCP socket. Further analysis of this code shows that this code path is followed by all TCP packets sent by the kernel except `SYN-ACK` and `RST` packets that are not part of an active connection. These packets are assigned an IPID value from one of the 2048 counters as described previously. Figure 4.1 shows the IPID selection behavior for Linux kernel version 4.16 [73]. The `ip_select_ident_segs` function is eventually called to assign an IPv4 packet an IPID value. On line 398 the kernel checks to see if the *Do Not Fragment* flag should be set. If not, the IPID is chosen from one of the 2048 counters based on its hash value in line 411. Otherwise, the kernel will ensure that the socket exists and has a known destination address in line 404 before using the IPID counter from the current socket (line 405). For a full source listing see the Linux kernel source code [73]. The attack I describe handles both cases when the Do Not Fragment flag is and is not set. However, I will focus on the default Linux kernel configuration which sets the Do Not Fragment flag.

4.2.4 RFC 5961

RFC 5961 was introduced in August 2010 as a method for improving the TCP protocol's resistance to blind TCP `RST` attacks. It does so by adding the following behavior to the standard TCP resetting algorithm:

1. Incoming `RST` packets are checked to see if they match a valid TCP connection by verifying that the source address and source port match an established TCP connection in the machine.

2. The sequence number is compared to the next expected TCP sequence number and the next TCP window.

    - If the TCP sequence number *exactly* matches the next expected TCP sequence number the connection is reset as before.
    - If the TCP sequence number does not exactly match the next expected sequence number and *is not* in the expected TCP window the `RST` is ignored.
    - If the sequence number does not exactly match and **is** in the TCP expected window a challenge `ACK` packet is sent to the other end host.

3. Then, the machine receiving the challenge `ACK` responds with a `RST` packet with a sequence number *exactly* matching the acknowledgment number of the challenge `ACK` per normal TCP behavior.

4. Finally, the connection is reset since a `RST` packet was received using the *exact* expected sequence number.

The additions proposed by RFC 5961 appear to address the vulnerability of a blind TCP `RST` attack. An off-path attacker who could previously brute force a valid sequence number and cause a reset must now correctly respond to a challenge `ACK` that they can not see. While such an attacker could theoretically guess the correct sequence number to use as a challenge `ACK` response they would need to guess from all 2^{32} possible sequence numbers, an unlikely event.

In my tests and code review of the Linux kernel's implementation of RFC 5961, I discovered that Linux sends challenge `ACK`s in response to unsolicited `SYN-ACK`s. This is relevant to my attack because it means I do not need to account for sequence numbers in the spoofed `SYN-ACK`s I send (although doing so would simply be a matter of sending four sets of `SYN-ACK`s to ensure one set has sequence numbers in the expected window).

4.3 Implementation

The attack for detecting active TCP connections relies on using the Linux kernel's IPID counter behavior and the difference between per-connection TCP counters and other non TCP counters to detect a connection. As discussed in Section 4.2.2 the Linux kernel uses 2048 IPID counters that are assigned based on the hash of the IP addresses and protocol for a given connection.

In order to detect off-path TCP connections the attack I describe requires the following primitives:

- A reliable method an off-path attacker can use to trigger off-path traffic that increments different IPID counters when there is a TCP connection present on a given port and when there is not a TCP connection.

- A method for counting IPID changes caused by off-path traffic.
- A method for determining which counter off-path traffic used based on the counted IPID changes.

I will discuss my implementations for all three primitives below and describe how they can be combined in the attack I describe.

4.3.1 Triggering Off-Path Traffic

To trigger the desired off-path traffic an attacker would need to send packets to the targeted machine that cause an off-path response, sent over TCP, which will use different counters when a TCP connection is present on a given IP-Port 4-tuple and when a TCP connection is not present. As described in Section 4.2.4, if a TCP connection is present `RST` packets within the current sequence number window will trigger a challenge `ACK` from the server. Out of sequence `SYN-ACK` packets also trigger this behavior in the Linux kernel's network stack. My implementation of the attack presented uses these out of sequence, or "unsolicited", `SYN-ACK` packets, using the spoofed source IP address of the off-path client machine, to cause the targeted server to generate off-path challenge `ACK` packets to be sent to the client machine. These packets will use the TCP connection's per-connection IPID counter to populate the challenge `ACK` packet's IPID field. If there is no TCP connection present for the IP-Port 4-tuple used by the "unsolicited" `SYN-ACK`, the packet will trigger a `RST` packet in response. This `RST` packet will populate the packet's IPID field from one of the 2048 IPID counters, since there is no per-connection counter to use because there is no TCP connection. By using "unsolicited" `SYN-ACK`s in this way my implementation has a method for triggering off-path TCP traffic that will use different counters depending on the state of a given IP-Port 4-tuple (i.e., whether or not a TCP connection exists for a given 4-tuple).

4.3.2 Counting IPIDs

As discussed in Section 4.2.2 the Linux kernel uses a mix of per-connection counters for established TCP connections and 2048 IPID counters for TCP packets sent outside established TCP connections. In order to count packets my implementation of the attack relies on counting packets sent *outside* an established TCP connection. This can be accomplished via the use of hash collisions in the hashing algorithm used to determine which of the 2048 IPID counters an outgoing TCP packet uses when sent outside an established TCP connection. Using this technique an off-path attacker can ensure that the source IP address they are using pulls from the same IPID counter that is used when sending off-path traffic. This allows the attacker to count the number of off-path packets sent by computing the change in IPID between two probe packets, with the difference being the number of packets sent.

Finding IPID Collisions

In order to find IPID collisions the attack uses a technique similar to that described by Zhang et al. [83], though only applied to IPv4 addresses. As described in Section 4.2.2 the Linux kernel uses a hash of a connection's source and destination addresses, network protocol, and a random value generated on startup. Given this I define an IPID collision as: For a server with a secret value S and IPv4 address T I say that an attacker IPv4 address A *collides* with another IPv4 address C if:

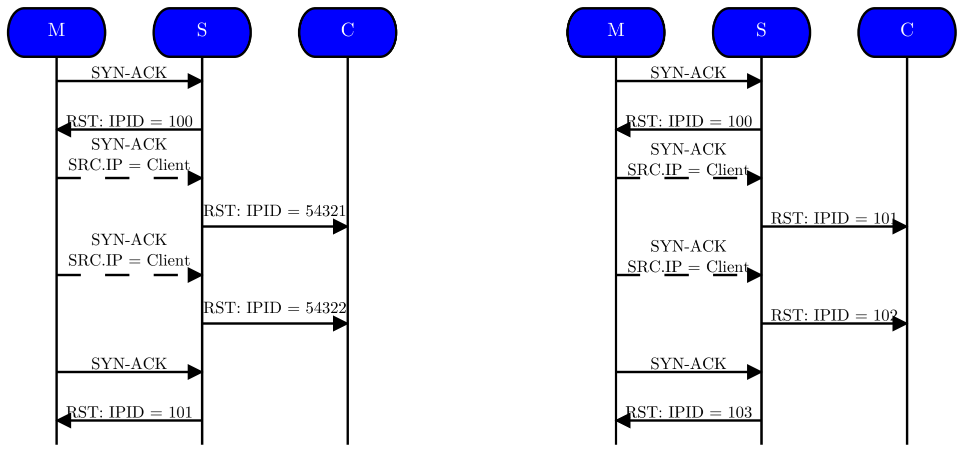
$$hash(A, T, PROTO\_TCP, S)$$

points to the same IPID counter as

$$hash(C, T, PROTO\_TCP, S)$$

Since the attack focuses on IPv4 TCP connections and my implementation uses only IPv4 and TCP packets I can treat the destination address and network protocol as constant. I assume that the server I am using as the destination of the attack does not reboot during my scan which allows us to treat the random value used in the hash as constant. This means that in order to find an IPID collision between an attacker IPv4 address and the client of an active TCP connection I only need to find a IPv4 address that would result in the same hash as my client machine's IPv4 address. Note that while my implementation only uses TCP packets to try to find IPID collisions, for simplicity, it is possible to use other protocols to increase the probability of finding a collision, though I did not explore this possibility.

To detect such a collision I can probe the server from a range of IPv4 addresses checking to see if responses to my TCP probes show the presence of off-path traffic using the same global IPID counter. First, I send a `SYN-ACK` packet to the server from an attacker address. This `SYN-ACK` is not part of any active connection so the server will respond with a `RST` packet. Next, I send a `SYN-ACK` packet that spoofs the source address of the client machine. These spoofed packets are sent to find IPID hash collisions, not active connections, and due to this are sent using a source port unlikely to be used in an active connection, such as a port outside common ephemeral port ranges. Once again the server will respond to this `SYN-ACK` with a `RST`, since it is not part of an active connection. Finally, I probe from the attacker address using a `SYN-ACK` and receiving a `RST`. If the attacker's IPv4 address and the client's IPv4 address generate the hashes that point to the same counter then their IPIDs will come from the same IPID counter. I can check if this is the case by computing the difference between the IPID of the responses to the two probe packets. Assuming that all 3 packets were sent in under 10ms the random value added to each IPID



(a) No IPID Collision (b) IPID Collision

Figure 4.2: Example of behavior with and without an IPID hash collision.

packet as described in Section 4.2.2 will never be larger than one. If the difference modulo 2^{16} is one then a `RST` was not sent to the client machine in between my probes using the counter the attacker's IPv4 address uses. If the difference is two then the `RST` sent to the client used the same IPID counter as the probe packets and I know that my IPv4 address collides with the client's IPv4 address. The process of sending a probe `SYN-ACK`, spoofed `SYN-ACK`, and a probe `SYN-ACK` and checking the difference between response packet IPIDs can be repeated to validate the accuracy of a potential collision. Once I have found such a collision I can then proceed with the attack. For my implementation I chose the simplest method of sequentially scanning for collisions between each possible client and attacker IPv4 address pair.

As an example, consider Figure 4.2a and Figure 4.2b. In Figure 4.2a the attacker (M) and the client machine (C) do not have hashes which collide and therefore they do not use the same IPID counter. When the attacker probes the server (S) the response has an IPID of 100. Probes sent by the attacker, spoofing the IP address of the client, use a different IPID counter and each probe gives a different value than if

Number of Addresses (k)	Probability
1	0.00048828
2	0.00097632
4	0.00195169
8	0.00389958
16	0.00778395
32	0.01550732
64	0.03077416
128	0.06060128
256	0.11753004
512	0.22124677
1024	0.39354340
2048	0.63221039
4096	0.86473080
8192	0.98170224
16384	0.99966519
32768	0.99999989

Table 4.1: Probability of a collision amongst k addresses.

a collision had occurred. In Figure 4.2b the attacker and client machines are using two IP addresses that will cause a collision. In this case I can see the IPIDs use the same counter and increase by one for each packet sent to either the attacker or the client machine. The attacker can tell the two cases apart by checking the difference between its two probes. A difference of one indicates that the attacker's IP address does not collide with the client's and a difference of one plus the number of spoofed `SYN-ACK`s sent indicates a collision. In both cases the server is only responding with `RST` packets which will always choose an IPID counter from one of the 2048 counters as discussed in Section 4.2.2, avoiding the per-socket counter described in Section 4.2.3.

Probability of Finding a Collision

To find a collision an IPv4 address is needed that collides with the client's IPv4 address on the server. Given that the Linux kernel uses 2048 different IPID counters based on a hash of connection parameters, any single IPv4 address has a probability of $\frac{1}{2048}$ of its hash colliding with the client's. If the attacker instead uses a pool of available IPv4 addresses then the probability of finding a collision within this pool of addresses increases as the size of pool does. The probability of finding an IPID hash collision between a given client IPv4 address and k attacker IPv4 addresses can be calculated as one minus the probability that **no** collision is found:

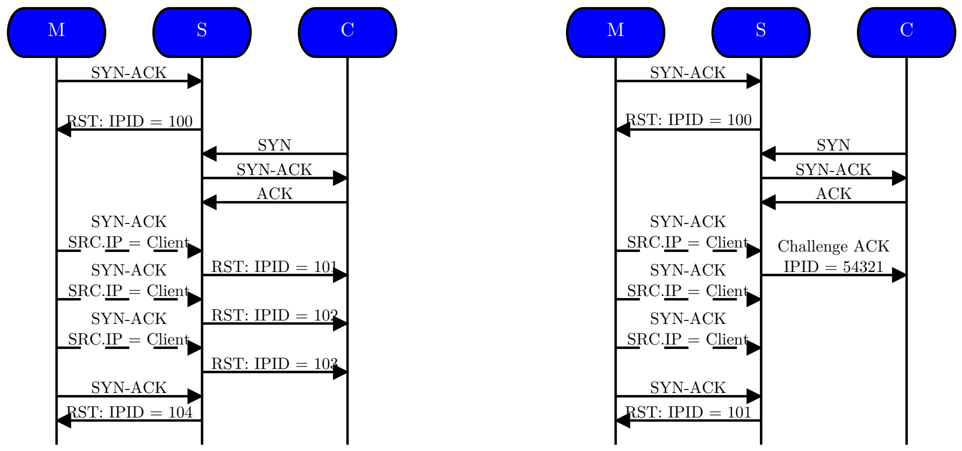
$$Pr(X = k) = 1 - (1 - p)^k \tag{4.1}$$

where k is the number of IPv4 addresses used and p is the probability of any single IPv4 address colliding, $\frac{1}{2048}$.

Table 4.1 shows the probability of finding a collision when using common CIDR network sizes.

Example

Consider what happens when I check a 4-tuple that represents an active TCP connection. First, the attacker queries the current IPID value of IPID counter used by the client machine, via an IPID hash collision discovered previously. Then, when the attacker sends "unsolicited", spoofed `SYN-ACK` packets to the server it will respond with a challenge `ACK`, sent to the client. The challenge `ACK` sent will use the IPID value from its per-socket counter as shown in Figure 4.3b. Since this is a separate counter from the IPID counter used for the `RST` packets sent in response to the attacker's probe `SYN-ACK`s it does not increase the IPID counter used and the difference



(a) No Connection (b) Active Connection

Figure 4.3: Checking for a connection.

in IPID between the two `RST` packets is one. As discussed previously if there is no active connection on a given 4-tuple each spoofed `SYN-ACK` packet will cause a `RST` to be sent to the client. Since no connection exists each of these `RST` packets will use the hash based IPID counter and each `RST` will increment this counter by one, as seen in Figure 4.3a. This will cause the difference in IPID between the `RST` packets sent in response to the attacker's probe packets to be the number of spoofed packets sent plus one.

4.3.3 The Attack

The attack for detecting active TCP connections uses the side-channel for using IPIDs to count the number of packets sent to differentiate between active and inactive TCP connection 4-tuples.

First, the attacker finds an IPv4 address that collides with the client IPv4 address in my potential TCP connection 4-tuple as described previously. Next, assuming a colliding IPv4 address was found, the attacker scans a chosen range of potential ports to try to detect a TCP connection. While the simplest method would be to scan each potential port one at a time this runs the risk of taking too long to find a connection before it closes, in the case of short lived connections. This is because the port field is a 16-bit value giving 2^{16} possible ports. Scanning each of these 65,536 possible ports that a machine could be using sequentially would likely take more time than the time a short lived TCP connection exists. To address this problem an attacker could modify the above attack to check multiple potential ports at once. However this would cause any parallel scans to interfere with each other since each one would be probing the same IPID counter.

I address these problems by dividing the range of standard ephemeral ports into a series of buckets where each bucket contains N connection 4-tuples where the client's source port is chosen from the ephemeral ports assigned to the bucket. For each bucket I send M `SYN-ACK` probes to each 4-tuple in the bucket. I only query the IPID value before and after I send all the probes from a given bucket. Based on the challenge `ACK` behavior described in Section 4.2 a `SYN-ACK` probe using a 4-tuple that does not represent a connection will trigger a `RST` packet response which increases the IPID counter by one. A 4-tuple that represents a valid connection will trigger a challenge `ACK` using the per-connection counter. Using this information I know that if a bucket does not contain a 4-tuple that represents a connection the difference in IPID value of the two queries should be equal to $M \cdot N$ since I send M probes for each of the N 4-tuples in a bucket. If the bucket does contain a 4-tuple that is an active connection the IPID difference will be $N \cdot (M - 1)$ as the packets sent via the active connection use the per-connection IPID counter. By only querying the IPID value before and after scanning each bucket instead of after each 4-tuple I can simply send all my $M \cdot N$ 4-tuple probes as fast as possible since I do not have to check for changes after each one. If I detect a bucket that contains a valid 4-tuple I then scan each of the N 4-tuples in the bucket until I find a 4-tuple that represents an active connection. Using this technique to speed up the attack, my implementation is as follows:

- Divide the ephemeral port range into buckets containing N ports.
- Scan each bucket as described in above.
- If a bucket containing a valid 4-tuple is found, scan each port in the bucket individually as described below:
  - Send a `SYN-ACK` from the attacker's IP address to the server.
  - Send 2 or more `SYN-ACK`s using spoofed packets using the client's IP address and the target port.
  - Send another `SYN-ACK` from the attacker's IP address to the server.

For my specific implementation I chose to send eight `SYN-ACK` packets for each individual port. I chose eight packets as it provides a large enough number to account for a small amount of noise caused by other packets using the same IPID counter while still being small enough to send quickly to avoid the kernel adding its own noise to the counter. The attack will work for a larger number of packets, though the more packets that are sent the longer the attack will take.

For each individual port scanned, I measure the IPID difference before and after sending all eight `SYN-ACK` packets. If the difference is less than the number of `SYN-ACK` probes sent to the potential 4-tuple minus one then I consider the port to be a potential TCP connection. I chose this threshold to allow for a small amount of noise from outside traffic to impact the IPID without greatly impacting my implementation. If the difference is greater than or equal to the number of `SYN-ACK` probes sent to the potential 4-tuple, minus one, I conclude that the 4-tuple does not represent a valid TCP connection and move to scan the next possible 4-tuple.

4.4 Experimental Setup

In testing the effectiveness of the attack I wanted to answer three main questions:

1. How effective was the attack “in the wild”?
2. Was the attack robust to noise that might be present “in the wild”?
3. How fast was the attack and is this fast enough to be practically usable?

In order to answer these three questions I opened TCP connections with popular web servers, that met the attack’s required criteria, and attempted to use the attack to detect these connections from a third “attacker” machine. The attack requires:

- A target server that is a Linux machine running kernel version 4.0 or newer.
- Access to multiple IPv4 addresses to use as attacker addresses.

My experiment proceeded as follows:

1. Using the top 250 sites from the Alexa Top Global Sites [4] I made a DNS `A` lookup for each and recorded all IPv4 addresses returned by the lookup.
2. Each IPv4 address was then scanned to check if it met the criteria for the attack.
3. For each IPv4 address that met the criteria for the attack I generated all unique pairs of IPv4 addresses from a set of 242 IPv4 addresses on my research network. These pairs were then used as client and attacker IPv4 addresses along with a valid server address to use when trying to find IPID collisions in the next step.

4. Once I had generated all client and attacker IPv4 address pairs I scanned each server IPv4 address that met my criteria to try and find IPID collisions on the server between a given client and attacker pair. Each pair that collides was then cached as a collision pair to be used in the full attack.

5. I then carry out the attack both before and after opening a TCP connection with each server IPv4 address. No data is sent to the server during the attack except TCP keep-alive packets to ensure the connection remains active during the course of the attack.

6. Once the attack has finished I close the TCP connection and log the results.

I chose to use the sites from the Alexa Top 250 as these represent some of the most popular and highly trafficked websites. Due to this, these machines can be viewed as a plausible worst case scenario for the attack. This is because such large amounts of traffic increases the likelihood that another connection's IPID hash collides with the client and the attacker's IPID hash. This traffic can be TCP traffic, such as `RST` packets, that use one of the 2048 global counters to populate the IPID field or non-TCP traffic (e.g., ICMP or UDP). In these cases any such collisions will add additional noise to the IPID counter, making precise changes more difficult to detect. Such noise makes these sites an excellent set to test the attack's robustness when dealing with noisy machines.

When choosing ephemeral ports to use as client source ports I wanted to choose a range that covered common operating system default values. TCP ports are represented as 16-bit integers, with the first 1024 usually reserved for common services. This leaves 64,512 potential ephemeral ports that could possibly be used as a client's source port. However, many operating systems do not use this full range of ports by default. Linux uses the range $[32,768,\ 61,000]$ as available ephemeral TCP ports. The Internet Assigned Numbers Authority (IANA) recommends that ephemeral ports be chosen from the range $[49,152,\ 65,535]$ [20]. Since Windows Vista, Microsoft operating systems have used the IANA range of ephemeral ports. An attacker would likely pick one of these two ranges to maximize the number of client machine's whose connections could be attacked while also avoiding attempting to scan all possible ephemeral ports. For my experiments I chose to use a hybrid range of $[32,768,\ 65,535]$ as possible ephemeral ports to cover both the IANA range and the Linux range.

All client and measurement IPv4 addresses used were unbound addresses chosen from a set of 242 IPv4 addresses on my measurement network. The measurement machine I used responded to all packets sent to these addresses. The measurement machine was an Ubuntu Linux machine running Ubuntu 16.04 LTS with Linux kernel version 4.4. In a real attack, the packet delays, packet loss, and other characteristics of the Internet between the measurement machine and the server would have an effect on the attack, but for the client the Internet characteristics between the client machine and the server are not relevant to the attack, since the client IP is simply kept as state on the server for an open TCP connection from the attacker's perspective. Therefore my experimental setup is identical to a real attack in terms of Internet traffic considerations.

4.5 Analysis

In this section I describe the results of my experiment and discuss potential factors that could impact the attack's effectiveness. As described in Section 4.4 I chose web sites from the Alexa Top 250 as servers. After performing DNS `A` lookups for each host I obtained 464 unique IPv4 addresses to use as server machines. Of these 464 IPv4 addresses I found 182 (39.22%) that met the required behavior criteria: a machine running a Linux kernel version 4.0 or newer, that responded to "unsolicited"

Scans	Connection						No Connection			
	True Positive		False Negative		False Positive		True Negative		False Positive	
	#	%	#	%	#	%	#	%	#	%
2593	2194	84.61%	282	10.88%	117	4.51%	2452	94.56%	141	5.44%

Table 4.2: Overall accuracy of the attack.

`SYN-ACK` packets with `RST` packets. Of these 182 IPv4 addresses I was able to find valid IPID collisions between a pair of my 242 measurement IPv4 addresses and 136 (74.73%) server addresses. I carried out 2,593 total mock attacks, over the course of 7 days, on active TCP connections between an IPv4 address from the Alexa Top 250 and a client IP address on my own network. Each scan attempted to detect the presence of a TCP connection before and during an active TCP connection.

4.5.1 Analysis of the Attack's Accuracy

When describing my results I separate attacks where no TCP connection existed from those where a TCP connection does exist. Overall there are four possible outcomes, depending on whether or not a connection existed:

- True Positive: A connection exists and the attack found the 4-tuple corresponding to the connection correctly.
- True Negative: No connection exists and the attack found no connection.
- False Positive: No connection exists and the attack found a connection **or** a connection exists and the attack found an incorrect 4-tuple for the connection.
- False Negative: A connection exists and the attack failed to find the connection.

In cases where a connection did exist the attack was able to detect the TCP connection 84.61% of the time with a false positive rate of 4.51% and a false negative rate of 10.88%. When no connection was present the attack was able to correctly detect this 94.56% of the time with a false positive rate of 5.44%. Table 4.2 shows my overall results.

4.5.2 Analysis of the Attack's Runtime Performance

In addition to analyzing the accuracy of the attack I also analyzed the performance of the attack when scanning the standard Linux kernel's ephemeral port range for active TCP connections. If this scanning takes too long to complete then it is possible that an attacker would miss the existence of a short lived connection.

Using the same set of server IPv4 addresses and IPID collisions used in the accuracy analysis detailed in Section 4.5.1 I measured the time the attack took to either find an active TCP connection or scan the chosen ephemeral port range. On average the attack takes 75.81 seconds to find an active TCP connection. When there is no connection the attack takes 135.59 seconds on average to scan the chosen ephemeral port range and confirm there is no connection. How fast the attack needs to be depends on a lot of context that will vary across different potential types of attacks (e.g., Is the connection repeated so that multiple attempts can be made? What are the `keepalive` settings of the server and browser? Is it an application that has persistent connections, such as Tor?).

4.6 Discussion

In this section I discuss the potential applicability of the attack, factors that I noticed during my experiments that affect the applicability of the attack, possible mitigations that could be taken against the attack, and ethical considerations I made in designing my experiments.

4.6.1 Applicability

As discussed previously this attack requires the use of multiple attacker IPv4 addresses to increase the probability of an attacker finding a valid IPID hash collision so that IPID values of attacker packets and client packets use the same IPID counter. In order to have a 50% probability of finding a collision an attacker would need to use approximately 1400 IPv4 addresses. To have a 90% or greater probability an attacker would need approximately 4700 IPv4 addresses. While access to this many available IPv4 addresses is likely beyond the capabilities of a simple attacker it is within the realm of availability for a large botnet or nation-state attacker. Previous work studying the Great Cannon [56] and the Great Firewall of China's attempts to actively probe Tor bridges [27] shows that some nation-states likely already possess the ability to spoof thousands of IPv4 addresses, 13,183 and 16,083 respectively, most of which came from the same /16 subnet. If similar numbers of IPv4 addresses were used to implement the attack I describe in this chapter, an attacker would have a greater than 99% probability of finding a valid collision that could be used to detect an active TCP connection with a chosen, vulnerable Linux machine.

I assume that the attacker has a reason to suspect that the victim machine is making a connection to a given server, and that this is likely the case. The base rate fallacy [8] would apply if the likelihood that the connection actually exists is relatively low. This depends on the overall goals of the attacker and context of the attack, which may also mitigate the effects of the base rate fallacy. For example, if an attacker's goal is to know if a given client is connected to the guard node associated with a Tor circuit, a higher false positive rate may be tolerable because this is simply another lead to follow in a broader and more thorough investigation.

4.6.2 Sources of Error

During the course of my experiments I noticed three main causes of errors that impacted the effectiveness of the attack. The first cause was noise causing larger than expected IPID increases. The most likely cause of this behavior is additional machines connecting to the server whose IPv4 addresses hash to use the same IPID counter as an attacker IPv4 address. Each packet these additional connections send increases the IPID counter in addition to the increases caused by the attack. This additional increase can cause the IPID counter an attacker is using to increase beyond the number of probe packets expected making it appear that no challenge `ACK` was sent when scanning a given 4-tuple as described in Section 4.3. One method for accounting for this noise would be to try modeling the noise. Ensafi et al. [27, 28] used ARMA modeling to account for noise affecting the global IPID counters. Pearce et al. [62] accounted for noisy IPID counters by using Sequential Hypothesis Testing, while ONIS [83] modeled noise and accounted for it by using the Akaike Information Criterion. The attack I have described does not attempt to handle detected cases of noise; instead I leave this possibility for future work.

The second cause of errors I noticed is that certain IPv4 addresses I used as servers would change the IPID counter being used after successfully finding an IPID hash collision. Since I cached all located IPID collisions as described in Section 4.3 this caused any subsequent attempts using the cached collision to fail. An attacker using a similar system of caching known collisions would be forced to try to find another valid collision before continuing their attack. I noticed these changes occurring on the order of hours and not minutes meaning that single attacks or those not using a similar system of caching collisions are unlikely to be affected. Performing an additional test to ensure the chosen collision is still valid before launching the full attack mitigates this source of error.

There are two likely explanations for this changing collision behavior. The first is that the target machine was restarted. As discussed previously in Section 4.2 a random value is included as part of the IPID hash to prevent the possibility of precalculated collisions. This value is chosen when a machine boots and remains unchanged while the machine is still running. Any precomputed collisions that use a server that restarts will become invalid whenever that server restarts. The second possibility is that these IPv4 addresses represent multiple machines assigned the same IPv4 address via load balancing or an address translation protocol. If this is the case when an attacker first finds a collision it is valid for one of the set of machines assigned a given IPv4 address. However, if the attacker later attempts to reuse the same collision the load balancing or address translation system could send the attacker's traffic and the client's traffic to different machines. This would result in the collision no longer being valid and all subsequent attacks would fail. Popular web sites are likely to use load balancing techniques to improve overall performance and an attacker would need to detect whenever this case has occurred and appropriately handle when it has. I leave the development of such a technique to future work, and note that prior TCP/IP side-channel attacks (e.g., Cao et al. [13] and Knockel and Crandall [45]) are also affected by layer 4 load balancing, making it an interesting open problem in general.

The third source of error I noticed was the largest cause of false positives in my data. These errors were caused by packets arriving out of order and the server responding to the attacker's second probe packet before all the spoofed packets. Recall the attack as described in Section 4.3. At two points during the attack I send N spoofed packets using a given spoofed 4-tuple while sending probe packets from the attacker IPv4 address before and after sending the spoofed packets. My implementation assumes that responses to all N packets are sent before the response to the second attacker probe. In the event that these packets arrive out of order the second attacker probe will have an IPID value that is in the middle of the IPID values given to packets sent in response to the N probes. This results in a given probe appearing to cause less packets to be sent making it seem like the probe triggered the packets that used the per-connection IPID counter. This causes the attack to either detect a connection where none existed or to detect a connection on an incorrect port. Given my low false positive rate of less than 5% this does not occur frequently and could be mitigated by rerunning the attack multiple times. I leave the development of techniques to detect and better handle packet reordering to future work.

4.6.3 Mitigations

There are a number of possible mitigations the Linux kernel could use to try and prevent this attack. The immediate place to consider mitigations is the Linux kernel's IPID behavior.

The kernel could change the IPID counter behavior it uses to try to remove any information flows that could be used as a side-channel. However there is no obvious choice for the behavior that should be chosen. If the kernel were to use per-destination IPID counters, as it did previously, then the kernel would again be vulnerable to side-channel attacks such as those discussed by Knockel and Crandall [45]. Moving to any type of global IPID counter is also a poor choice as it would remove the need for an attacker to have hundreds of IPv4 addresses available to find IPID hash collisions, though it would increase the amount of noise present for active machines.

One possibility is that the Linux kernel could use a constant IPID for `RST` packets that are not part of an active TCP connection. `RST` packets should not contain any data and as a result should never be fragmented. Using a constant IPID for such packets would make it impossible for the attack I have described to determine if off-path packets have been transmitted, removing the attack's ability to tell whether the traffic used a per-connection counter or one of the 2048 global counters. This solution would mitigate the attack I have described, though it does still allow an off-path attacker to count non-TCP packets via IPID hash counter collisions.

Another possibility is to use random IPID values for each outgoing packet. This is the approach taken by some versions of BSD-based operating systems, including Mac OSX. Using this approach, for each packet that is sent a random value is generated and used as the IPID field. This removes any ability for an off-path attacker to count IPID differences, assuming the random number generation scheme is sufficiently difficult to predict. However, this approach adds additional overhead computing random values to each outgoing packet, which may have undesirable performance overhead.

4.6.4 Ethical Considerations

In order to avoid potentially exhausting resources on the server machines I used in my experiment I strove to follow best practices. I only initiated one TCP connection at a time with a given server and all TCP connections were completed to avoid taking up resources with "half-open" TCP connections. Once I had finished a given scan I closed and reset each connection immediately, to avoid using up server resources. All of my probe packets are `SYN-ACK` packets, which are immediately reset by the server. At no point do I test the attack described on any TCP connection other than those I have initiated. Via reverse DNS my client and measurement machine IPs pointed to a website that explained the nature of my study and gave contact information for network administrators to opt out of my probes, and I have arranged for all abuse complaints for my research network to be forwarded directly to me. I received no opt-out requests or abuse complaints at any time during this study.

Disclosure

I disclosed the side-channels used in the attack to the Linux kernel developers on August 22, 2018. After discussing the attack and possible mitigations, the Linux developers released a patch that mitigated the attack on September 11, 2018. The chosen mitigation strategy was to have the Linux kernel use an IPID of zero for all TCP `RST` packets sent outside an established TCP connection. As I discussed previously this removes the side-channel which allows an off-path attacker to determine whether off-path TCP traffic was using one of the 2048 global IPID counters or a per-connection counter and detect the presence of a TCP connection. As result, Linux kernel versions 4.18 and newer are no longer vulnerable to the attack I have described. However, the underlying side channel still remains for other protocols and could be used to attack user privacy in applications that use these protocols (e.g., UDP-based protocols such as DNS).

4.7 Conclusion

I have presented a novel off-path attack that can detect the presence of an active TCP connection between a remote Linux server and an arbitrary client using side-channels present in the Linux kernel. The attack leverages side-channels in the kernel's implementation of shared and per-connection counters. This attack is a purely off-path attack that does not require access to any packets sent between the client and the server. All that is required to reliably use the attack is access to multiple IPv4 addresses for the measurement, that the server machine be running a Linux kernel of version 4.0 or higher, and that the server responds to unsolicited `SYN-ACK` packets with `RST` packets.

Compared to previous attacks which could detect TCP connections off-path [13, 45] my attack does not fully exhaust the shared, finite resource it uses as a side-channel. Instead my attack uses a side-channel not to see if a shared, finite resource has been exhausted but rather *which* resource is being used. I am able to do this by associating changes in the Linux kernel's networking state with dependent changes in the usage of the shared, finite resources used to populate IPv4 packet IPID fields. This indicates that networking side-channels can potentially exist in cases where *any* shared state or information is used to determine network state behavior. This means that enumeration and hardening of shared, finite resources such as global rate limits or shared buffers is likely to be insufficient when attempting to harden networking stacks against side-channel attacks.

I have provided an evaluation of the attack "in the wild" and discussed its effectiveness and performance. I have shown that the attack is accurate and can be run quickly enough to detect active connections that persist for an average of 76 seconds or longer. In addition I have discussed the potential applicability of the attack, sources of error that would affect its applicability, and possible mitigations against the attack. Finally, I have discussed what this attack can tell us about yet undiscovered TCP/IP side channels that past attacks could not.

Chapter 5

Internet-Wide Measurement of Likely IPID Side-Channels and Behavior

In this chapter, I describe my work studying IPv4 IPID value generation and uniqueness across the entire IPv4 address space. Previous work has only studied IPID generation across subsets of the IPv4 address space and made assumptions as to the number of different IPID generation schemes.

The purpose of my study is two-fold: To develop a method for trying to detect likely IPID side-channels via large-scale scanning and black-box testing. My previous work, as well as related works, often require researchers to manually reverse engineer the network stack of various systems to find side-channels that can be leveraged to create new scanning techniques or which represent new security vulnerabilities. This is a time consuming process and often requires direct access to source code or compiled binaries, making it difficult to apply to specialized network hardware (e.g., Internet backbone routers, embedded systems, Internet-of-Things devices).

Techniques for carrying out blackbox testing could help automate this part of this process or provide a list of systems likely to contain side-channels, to help narrow down areas where more focused research would be fruitful. To study IPID generation methods across the entire IPv4 address space and provide a richer picture than those presented by previous studies. This is helpful in providing a more complete picture of the number of systems that could be used in side-channel based attacks or scanning techniques. Since previous studies only scanned a subset of available IPv4 addresses, often chosen from publicly provided lists of web servers, DNS servers, or popular services the overall trends they report on may not be accurate or provide a complete picture of IPID behavior on the Internet as a whole.

In addition to providing a more complete picture of IPID behaviors on the Internet that could contain side-channels, my study also provides analysis of Internet behaviors that make side-channels more difficult to use. As discussed in Section 4.6.2, load-balancing systems can present problems for side-channel based techniques by sending packets sent to a single IP address to different machines, making accurate usage of side-channels difficult, if not impossible. The work I present in this Chapter will provide a current view of the prevalence of load-balancing and other systems that can impact side-channel techniques.

My contributions are as follows:

- An IPv4 Internet wide scan and analysis of the different IPID generation behaviors observed.
- Analysis of those IPID generation behaviors that appear to contain shared information flows that could be leveraged as information side-channels.

The rest of this chapter is structured as follows: Section 5.1 discusses common IPID generation schemes and the possible ways each can be used as a side-channel.

Section 5.2 describes the scan I performed to gather data about the kinds of IPID generation schemes present across the entire IPv4 address space, while Section 5.4 details the results of my scan. In Section 5.5 I discuss the implications of my results and finish with my conclusions in Section 5.7.

5.1 Background

As discussed in Chapter 4, every IPv4 packet header contains a 16-bit identification field referred to as the IPID, as defined in RFC 791 [65]. RFC 791 states that each IPID must be unique for the 3-tuple containing a packet's source address, destination address, and protocol number for as long as the packet, or any of its fragments, could be alive on the Internet. While this could be handled via the use of one IPID counter per destination host, this would be impractical given the IPID field is only 16 bits in length and is unable to provide a unique value for each of the 2^{32} possible IPv4 addresses. Instead, it may simply be easier to use unique identifiers for each packet, regardless of destination, a practice that most operating systems choose to follow.

The specification for the usage and generation of the IPID field was updated in RFC 6864 [74]. This RFC updated the IPID field to better match its current use case on the Internet and to more clearly define the purposes for which the IPID should be used. Specifically RFC 6864 updates the IPID field specification as follows:

- The IPID field MUST NOT be used for any purpose other than packet fragmentation and reassembly.
- Originating sources MAY set the IPID field of atomic packets to any value.
- All devices that examine IPv4 headers MUST ignore the IPID field of atomic packets.

These changes focus on the use of IPID values in atomic packets. Atomic packets are IPv4 packets which are small enough to fit within the minimum packet size for IPv4 packets, as specified in RFC 791, without fragmentation. This minimum packet size is 68 bytes. Since atomic fragments can be transmitted without ever being fragmented, and the IPID field should only be used for packet fragmentation, the value of the IPID field is irrelevant for atomic packets.

5.1.1 Common IPID Generation Methods

Previous work studying IPID behavior [70] discussed four common IPID generation methods. The first of these is one using a shared, global, incrementing counter. In this generation method a given machine has a single, shared counter which is incremented by a constant value every time a packet is transmitted. Another common IPID generation method, designed to mitigate security issues of a shared, global, incrementing counter, is to use *per-host* or *per-connection* counters. In this method each unique IP address or established, stateful connection uses its own unique counter for each packet sent to a given host or via a given connection. Two less common IPID generation schemes described are a constant value and random values. In a constant IPID generation scheme every outgoing packet uses the same, constant value. This method prevents the effective use of IPv4 fragmentation of such packets and does not conform to the RFC describing IPv4 and IPID generation. The random generation scheme assigns a random IPID value to each outgoing packet. In contrast to the other commonly used methods, the random method includes a higher performance cost as each outgoing packet requires the generation of a random number instead of simply reading and incrementing a counter.

5.1.2 Shared Global Counter as a Side-Channel

The most common early IPID generation method was the usage of a shared, global, incrementing counter. Operating systems using this method provided an easy to access side-channel that allowed for the trivial counting of the number of off-path packets sent by a given machine. All that an attacker needs to do to count the number of packets is to send their own probes and calculate the IPID difference between each set of probe responses. Consider a machine, using a shared, global, incrementing counter being scanned by an attacker. First, the attacker sends a probe packet that will cause the remote machine to respond to the attacker's machine. This response will be assigned the current value of the shared IPID counter, which will then be incremented. This increment is most often a simple increase of one but some operating systems use another constant value (e.g., 8, 256, 512). The attacker then attempts to trigger off-path traffic between then chosen remote machine and another machine. Assuming the attacker is able to trigger such off-path traffic, all outgoing off-path traffic will be assigned the current IPID value which will then be incremented. Then, the attacker sends a second probe packet to the remote machine. This response will use the current IPID counter. The difference between the first and second probe packet responses the attacker receives will then indicate the total number of packets sent as a result of off-path traffic. This provides a simple, yet flexible, side-channel for an attacker to exploit to gain information about the off-path behavior of the remote machine.

5.1.3 Per-Host Counters as a Side-Channel

Per-host IPID counters present a more difficult challenge for attackers attempting to use the IPID field as a side-channel. Work by Knockel and Crandall [45] showed that per-host counters can be used with additional side-channels in other shared, finite resources (e.g., global fragmentation caches or other shared memory buffers) to infer information about off-path connections.

In Chapter 4 I discussed my work developing an attack that takes advantage of the Linux kernel using a mix of shared and per-host IPID counters. My attack takes advantage of the Linux kernel using different counters depending on the state of TCP connections to detect *which* counter is used as a side-channel to detect active TCP connections.

As my own work and prior work has shown, per-host IPID counters can still be used as side-channels. However, when compared to shared, global IPID counters, per-host counters often require an additional side-channel or the presence of associated behavior changes to allow attackers to infer off-path information via these side-channels. This is due to the lack of direct information flow between separate counters.

5.1.4 Random Values as a Side-Channel

Random IPID values have been previously described as an uncommon occurrence on the Internet, but is a method of IPID generation that has seen some recent, growing interest. Unlike other generation schemes a randomly assigned IPID, when generated using sufficient randomness and with replacement, should be practically impossible to predict and eliminate any information flows through the IPID field in outgoing packets. However, such as system is unlikely to be fully RFC compliant, as the possibility exists that two outgoing packets could have the same randomly generated IPID value. In addition, it is computationally expensive to generate truly random numbers most systems instead choose to use a pseudo-random number generator (PRNG). Due to their pseudo-random nature PRNGs are deterministic and poorly designed or implemented PRNGs generate a sequence of values that it is possible to predict. An attacker who gains sufficient information about the PRNG a given operating system uses can predict the sequence of IPID values the system will use. Once this is accomplished the attacker can count packets in much the same way they could for a shared, global IPID counter. Instead of calculating the difference between the IPIDs of the attacker's probe packets the attacker would compute the number of random values that a given PRNG must have generated between the IPIDs of the responses to their probe packets. Klein [44] describes how a predictable PRNG used in older version of OpenBSD, NetBSD, FreeBSD, and Mac OS X could allow an attacker to predict upcoming IPIDs, opening the possibility of these systems being used as part of an off-path idle scan or other IPID-based side-channels.

5.2 Experimental Setup

To study IPID generation algorithms used across the entire IPv4 address space I wanted to focus on how IPIDs were generated for different kinds of packets, protocols, and source IP addresses. To this end my scans were as follows:

- Single source address mixed TCP and ICMP scan.
- Multiple source address TCP scan.

For all scans packets were sent at 10ms intervals (i.e., The first packet was sent, after 10ms the second packet was sent and so on.) and a 100ms "cooldown" was implemented between each scan. This "cooldown" period served two purposes: First, it was designed to ensure that a scan target was not subject to a persistent stream of packets that could be interpreted as a Denial-of-Service attack or cause connection issues. Second, this "cooldown" ensured that subsequent scans did not occur one immediately after the other, potentially causing them to interfere with each other.

While round-trip-times greater than 100ms are not uncommon on the Internet, this "cooldown" needs to be greater than common round-trip-time variances, not the round-trip-time values, to avoid packet reordering causing interference between scans. The value of 100ms was chosen empirically, as it seemed to give the best trade-off between avoiding subsequent scans interfering with each other and allowing each scan to complete quickly enough that the overall scan did not take too much time.

5.2.1 Single Source Address Mixed TCP/ICMP Scan

The single source address mixed TCP/ICMP scan consists of one type of TCP packet, TCP `SYN-ACK`s, and one type of ICMP packet, ICMP Echo Requests, sent in alternating order (i.e., TCP packet, ICMP packet, TCP packet, ICMP packet, etc.). The goal of this scan is to determine how a given host generates IPIDs for different protocols. The Linux kernel uses a generation scheme where different protocols use different IPID counters, as I discussed in Section 4.2. The goal of this scan is determine how widespread this type of generation behavior is and if there exist operating systems other than Linux that have similar behavior.

5.2.2 Multiple Source Address TCP Scan

In the multiple source address TCP scan, I scan from two different IPv4 source addresses, alternating which address is used for each outgoing packet. The goal of this scan is to study how many different systems use some form of *per-host* or *per-destination* IPID counters. As with the single source TCP scan, four different types of TCP packets are sent in groups of thirty-two. The types of TCP packets are the same as those described in the single source TCP scan.

5.3 Analysis

In this section I describe the data analysis techniques I used to study IPv4 Internet IPID generation and uniqueness behaviors. In order to get a better understanding of the different types of IPID generation and uniqueness behaviors and their prevalence on the IPv4 Internet, I chose to analyze the data collected during my Internet-scale scan using hierarchical clustering: without making assumptions about predefined IPID behaviors. I chose to use hierarchical clustering because the goal of my scan was to study and better understand the different types of IPID behavior my scan recorded. Hierarchical clustering, as a descriptive data analysis tool, fits this goal better than a predictive classification method.

5.3.1 Dataset Size

Overall, my scan received responses from 57,706,077 unique IPv4 addresses. Of these, 55,166,980 IPv4 addresses responded to the Multiple Source Address TCP Scan and 21,336,820 IPv4 addresses responded to the Single Source Address Mixed TCP/ICMP Scan. The set of unique IPv4 addresses which responded to both scans contained 18,797,723 IPv4 addresses. Of these IPv4 addresses I determined that 8,483,921 provided a sufficient number of responses to build a reliable set of features and use the hierarchical clustering described below. It is these 8,483,921 IPv4 addresses that make up the dataset I analyzed.

5.3.2 Data Preprocessing and Feature Extraction

Hierarchical clustering is an unsupervised learning technique that has proven to be useful in a variety of data analysis contexts. Clustering is often performed either directly on raw data or on a set of features derived from the raw data. For my work I chose to derive a set of features from each of my data points and perform hierarchical clustering using these features, instead of the raw data. Given the potential size of my dataset and the number of potential dimensions that would exist for each individual IPID value sent by a scanned IPv4 address, the usage of features over raw data allows for a more tractable amount of data analysis. This comes from the reduced dimensionality of the feature set compared to the raw data. In addition, prior work by Salutari et al. [70] made use of features, when studying IPID behaviors with respect to only IPv4 address changes, indicating that such a technique is a viable approach to studying IPID values.

As discussed previously in Section 5.2, I used data collected from two tests, Single Source Multiple Protocol and Multiple Source Single Protocol test, as the data to use for my analysis. These tests provided data that would allow me to not only test behavioral changes related to the use of different source IPv4 addresses, similar to the work done by Salutari et al. [70], but to also test behavioral changes related to the use of different protocols. Given that the Linux kernel will choose different IPID values when either of these fields changes, as discussed in Section 4.2, this is important as only studying one type of change could provide an incomplete picture when compared with known IPID behavior.

Responses to each of these tests are then split into sequences of packets based on the field which is changed during the test. For packets received in response to the Single Source Multiple Protocol test received packets are divided into two sequences: x_{tcp} which contains responses to the TCP probe packets, and x_{icmp} which contains responses to the ICMP probe packets. The Multiple Source Single Protocol responses are also divided into two sequences: y_a which contains responses to probes sent using IPv4 address a, and y_b which contains responses to probes sent using IPv4 address b. It is these four sequences x_{tcp}, x_{icmp}, y_a, and y_b from which all features are derived. Table 5.1 contains the full list of features derived from my data and used during

Feature	Description
$H(x_{tcp})$	Entropy of sequence x_{tcp}
$H(x_{icmp})$	Entropy of sequence x_{icmp}
$H(y_a)$	Entropy of sequence y_a
$H(y_b)$	Entropy of sequence y_b
$S(x_{tcp})$	Circular standard deviation of sequence x_{tcp}
$S(x_{icmp})$	Circular standard deviation of sequence x_{icmp}
$S(y_a)$	Circular standard deviation of sequence y_a
$S(y_b)$	Circular standard deviation of sequence y_b
$ks(x_{tcp}, x_{icmp})$	2-Sample Kolmogorov-Smirnov score between x_{tcp} and x_{icmp}
$ks(x_{tcp}, y_a)$	2-Sample Kolmogorov-Smirnov score between x_{tcp} and y_a
$ks(x_{tcp}, y_b)$	2-Sample Kolmogorov-Smirnov score between x_{tcp} and y_b
$ks(y_a, y_b)$	2-Sample Kolmogorov-Smirnov score between y_a and y_b
$corr(x_{tcp}, x_{icmp})$	Circular correlation between x_{tcp} and x_{icmp}
$corr(x_{tcp}, y_a)$	Circular correlation between x_{tcp} and y_a
$corr(x_{tcp}, y_b)$	Circular correlation between x_{tcp} and y_b
$corr(y_a, y_b)$	Circular correlation between y_a and y_b

Table 5.1: Features derived from raw data.

hierarchical clustering. Each feature is described in more detail below.

Entropy

One of the features I derive from each of the above sequences (i.e., x_{tcp}, x_{icmp}, y_a, and y_b) is entropy. For this feature I calculate the entropy of each sequence, in bits. Used in this way entropy can be seen as a measure of how many repeated or constant values each sequence contains. This is useful in separating between IPID behaviors that return constant values and those that generate unique values for each packet. Note that, I use the term entropy informally to refer to the entropy of the process that generated the sequence. Sequences have algorithmic entropy, but entropy is a property of a process, not a sequence.

Kolmogorov-Smirnov Statistic

Another feature I derived was based on the 2-sample Kolmogorov-Smirnov statistic. The Kolmogorov-Smirnov test (K-S) represents a two-sided hypothesis test used to test whether two independent samples are drawn from the same underlying distribution. The K-S test commonly provides two values, a K-S statistic (D) and a p-value (p). Based on the size of the two samples and the desired p-value to meet or exceed, a *critical-value* (c) can be calculated that can be used as a threshold value for the given K-S statistic. If the K-S statistic is less than this *critical-value* then we cannot reject the null hypothesis that the two samples come from the same distribution. In order to use the K-S statistic as a feature I apply to following function to the results of a 2 sample K-S test, $ks(x, y)$ applied to a subset of the possible pairs of sequences derived from each test:

$$ks(x,y) = \begin{cases} 0 & D < c \\ D & otherwise \end{cases}$$

This feature provides a way to determine if two sequence of IPIDs are likely to have been generated using the same underlying method or pulled from the same counter. As an example of where this could prove useful, consider the following two sequences of IPIDs:

$$x = [1, 3, 5, 7, \ldots, 25, 27, 29, 31]$$
$$y = [2, 4, 6, 8, \ldots, 26, 28, 30, 32]$$

these two sequences are likely to have been generated from the value of a shared, global IPID counter. Each packet sent causes the counter to increment resulting in a difference of 1 between subsequent packets, assuming trigger packets were sent in a way such that responses would alternate which sequence they belong to. These two sequences will have a very high K-S value, indicating that it is very likely they are from the same distribution or counter.

Next, consider the following two sequences:

$$a = [2, 4, 6, 8, \ldots, 26, 28, 30, 32]$$
$$b = [1000, 1002, 1004, 1006, \ldots, 1024, 1026, 1028, 1030]$$

these sequences have a difference of 2 between each measurement within each sequence, just as the sequences x and y described above. However, it is clear that these two sequences do not come from the same underlying counter or distribution. These sequences are likely to have a very small K-S value, indicating that they are likely not from the same distribution or counter. The K-S statistic provides a way to differentiate between these two possible scenarios and help determine when a machine may be using a predictable counter or distribution to populate IPID values.

Circular Standard Deviation and Correlation

To the best of my knowledge, prior work studying IPID behavior that used common statistics, such as mean, standard deviation, and correlation, treated the sequence of IPID used as a linear sequence. While this performs well with compact IPID sequences that have relatively small difference between subsequent values it does not provide an accurate picture of the statistical behavior of IPID sequences when subsequent IPIDs have large differences between them or cross the "wrap-around" point. This "wrap-around" point occurs when an IPID value, which is represented as a 16-bit field, is incremented to a value greater than can be represented in 16 bits. The standard behavior when this occurs is for the 16-bit field to "wrap-around", beginning again from 0. These "wrap-arounds" cause IPID sequences to behave like circular data instead of linear data. As an example, consider the following sequence of IPIDs:

$$[65521, 65522, 65523, \ldots, 65535, 0, 1, \ldots, 13, 14, 15]$$

Using the common, linear method for calculating mean and standard deviation gives a mean of approximately 31,710.96 and a standard deviation of approximately 32,743.20. Based on these values it would appear that this sequence of IPIDs is centered close to 32,743 and has a large degree of variance between IPID values. However, this is not the case. When the "wrap-around" is accounted for it can be clearly seen that this sequence of IPIDs is centered around the "wrap-around" point and that the values are spaced very close together, since the difference between each value and the next is one.

A more accurate representation of IPIDs is that of a sequence of data values contained in a circular space, like directions on a compass face or the minutes of an analog clock, rather than a linear one. By treating an IPID value not as a point on a line but as the angle from 0 to a point on a unit circle, I can more accurately calculate statistics about IPID sequences while accounting for "wrap-arounds" and more sparse IPID sequences. Treating the above sequence of IPIDs as this type of circular data and computing the circular mean and standard deviation gives values of approximately 0.0 and 8.95, which more accurately describes the behavior of the sequence.

Representing IPIDs as circular data I derived two sets of features to use in my hierarchical clustering: circular standard deviation and circular correlation. Circular standard deviation provides a measure of how sparsely clustered IPIDs are. IPID behavior which uses constant values will have no spread, global and per-host behaviors will likely have some spread but not much, and random IPIDs will exhibit a large amount of spread. Circular correlation will allow me to better determine how correlated behavior is both between and within each individual test. This will allow me to better determine differences between random behavior, predictable IPID behavior such as global IPID counters, and IPID behavior that uses different per-host or per-protocol counters.

5.3.3 Hierarchical Clustering

The features described above and in Table 5.1 give me a set of 16 features for each IPv4 address that responded to my tests. To better study the variety of behavioral classes present in my data I used agglomerative hierarchical clustering using the UPGMA linkage method to study the data. Agglomerative hierarchical clustering is a method of clustering data based on the distance between clusters. Starting from a set of all individual data points as singleton clusters, agglomerative hierarchical clustering then iteratively merges the closest two clusters at each step to build a hierarchy of clusters built from nearest to furthest apart. To determine the distance metric between my data points I chose to use the cosine similarity distance as my distance metric. I chose this metric because it provided a way for me to compare my data points based on how similar their behaviors are, based on their features.

Due to resource and computational limitations I was not able to cluster my complete data set at once. As a result, I chose to cluster a random sample of 100,000 unique IPv4 addresses to determine an overall set of unique clusters within my data. Once I built this set of clusters I used a similarity search process between these clustered data points and the remaining, unclustered data points. This process calculates a similarity value between each unclustered data point and all of the clustered data points. Unclustered data points are then assigned to the cluster containing the most similar clustered data point.

5.4 Results

In this section I discuss the results of my analysis of the collected IPID behavior data. This includes discussion of the chosen “cut points” used to derive clusters during hierarchical clustering, an overview of the results of my hierarchical clustering on a

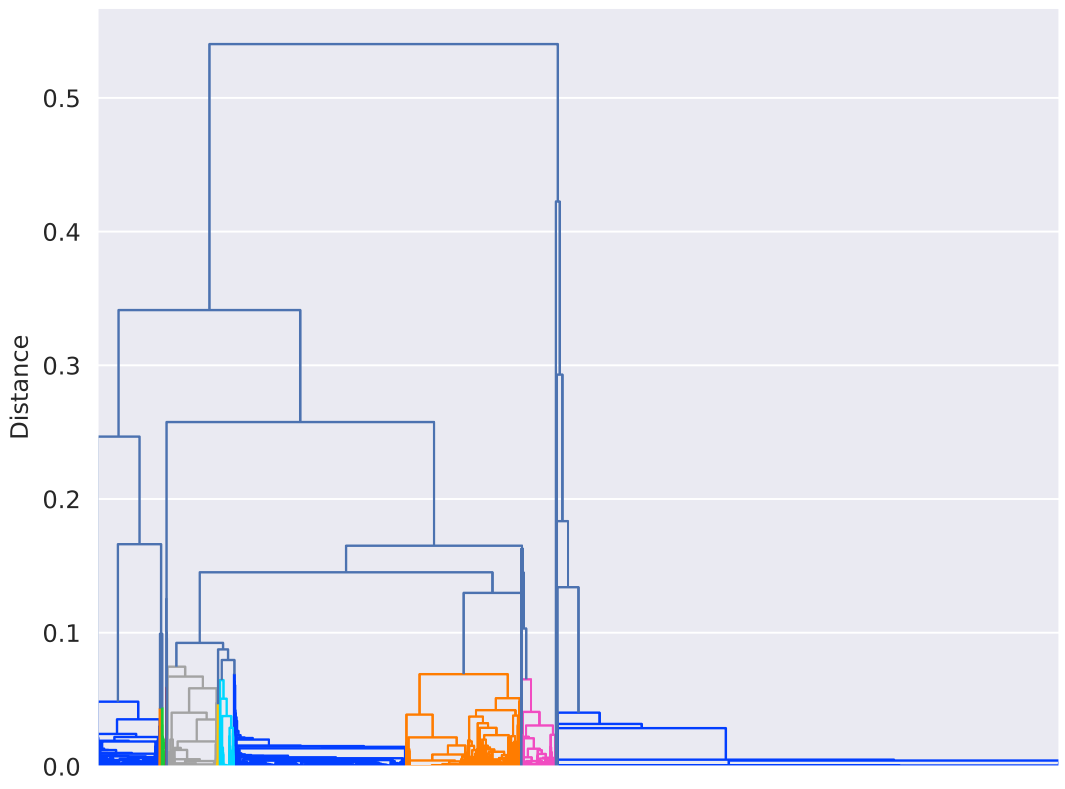


Figure 5.1: Dendrogram of clusters from a random sample of 100,000 datapoints.

random sample of 100,000 data points, and an overview of the analysis of the full data set using a similarity search process to assign data points to known clusters.

5.4.1 Choosing a Cut-Point

Hierarchical clustering builds a hierarchy of clusters, beginning with one single cluster at the highest-level and branching until each individual datapoint forms a singleton cluster at the lowest-level. A common practice when analyzing the clusters formed via hierarchical clustering is to choose a "cut-point" at a chosen distance threshold or number of clusters. Once this cut-point is chosen, any clusters formed by branching

Behavior	Description
Random	IPIDs appear to be generated at random
Random with Incremental	Random TCP and incremental ICMP IPIDs
Random with Firewall	Random TCP and constant ICMP IPIDs
Per-Host	Unique counters for each unique IPv4 address
Per-Protocol	Unique counters for each protocol
Unpatched Linux	Linux kernels before v5.19
Global IPID	A single, global IPID counter
Global IPID with Firewall	Global TCP counter and constant ICMP IPID
Constant	A single, constant IPID value
Patched Linux	Linux kernels after v5.19
Multiple Counter with Firewall	Mixed TCP behavior and constant ICMP IPID

Table 5.2: Unique behaviors found via clustering.

below this point are considered part of a single cluster. Based on empirical and manual analysis of a variety of cut-points I ultimately chose to use a cut-point of 26 clusters. Based on my manual analysis of the clusters created, I felt this cut-point provided the best set of unique IPID generation clusters without unnecessarily dividing clusters that represented larger, more common IPID generation methods.

Within my data, IPID behaviors were found that contain erratic behavior that is neither random nor consistent with previously documented behaviors. Often these behaviors form small clusters containing only a handful of datapoints. Due to the small size of clusters and their lack of similarity with any other clusters I chose to treat these small clusters as outliers, rather than unique behaviors. Based on empirical analysis of the 26 clusters created using the above discussed cut-point I chose a minimum threshold of 100 items for my clusters. Any clusters containing less than 100 datapoints were considered to contain outliers. Ultimately my analysis found 11 clusters of unique, non-outlier, IPID behavior. These behaviors are listed in Table 5.2.

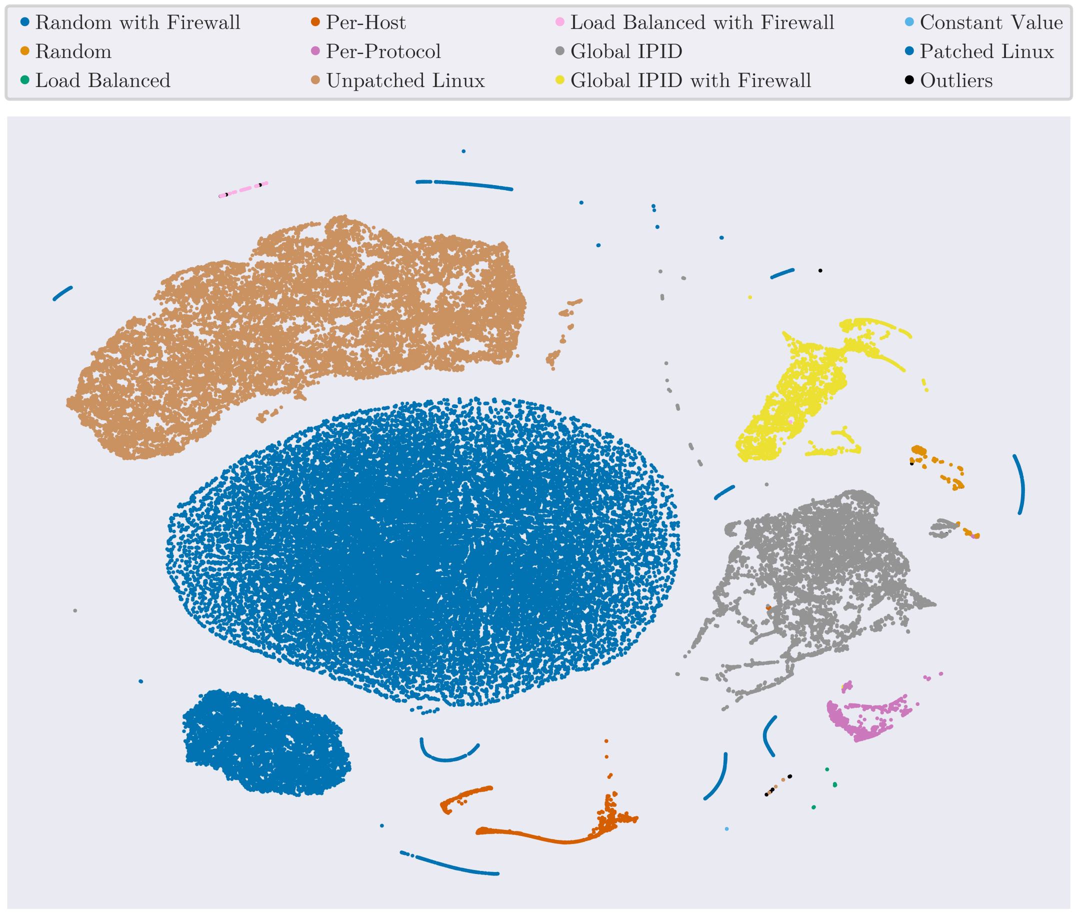


Figure 5.2: Scatter plot of clusters found on random sample of 100,000 datapoints.

5.4.2 Clustering Results for Random Sample

Table 5.3 shows the results of my clustering on a random, sample of 100,000 data points that responded to my scan. Overall, Linux machines make up the majority of these 100,000 datapoints. Determining the identity of the operating systems in use by non-Linux machines is more difficult due to the lack of additional information provided by an IPID scan alone. Linux is unique, in the sense that it uses an IPID generation scheme, discussed in Chapter 4, that has a unique set of IPID behaviors. Operating systems which use Global IPIDs or Random IPIDs are difficult to uniquely

Behavior	Number	Percent of Total
Random	386	0.39%
Random with Incremental	299	0.30%
Random with Firewall	6,604	6.60%
Per-Host	5,181	5.18%
Per-Protocol	1,366	1.37%
Unpatched Linux	17,749	17.75 %
Global IPID	11,923	11.92%
Global IPID with Firewall	3,474	3.48%
Constant	108	0.10%
Patched Linux	52,344	52.34%
Multiple Counter with Firewall	338	0.34%
Outliers	228	0.23%
Total	100,000	100.0%

Table 5.3: Clusters derived from a random sample of 100,000 datapoints.

identify based on IPID values alone. Figure 5.2 shows a projection of these 100,000 datapoints onto a two dimensional space, to provide a visual representation of the clusters I found during my analysis.

5.4.3 Cluster Analysis of Full Dataset

Table 5.4 shows the clustering results of the similarity search process applied to my full dataset. Compared with the results of my analysis of a random sample of 100,000 datapoints, shown in Table 5.3 the results for the full dataset are very similar. The overall distribution of datapoints between clusters, measured as the percent of my dataset contained within each cluster, remains relatively unchanged.

Overall Linux machines make up a majority of the IPv4 addresses in my dataset, 5,938,448 (70%) addresses in total. My scan was run approximately six months after the side-channel vulnerability I discussed in Chapter 4 was disclosed and patched,

Behavior	Number	Percent of Total
Random	34,521	0.41%
Random with Incremental	25,348	0.30%
Random with Firewall	548,033	6.46%
Per-Host	447,351	5.27%
Per-Protocol	118,266	1.39%
Unpatched Linux	1,509,941	17.80%
Global IPID	1,012,838	11.94%
Global IPID with Firewall	304,713	3.59%
Constant	8,848	0.10%
Patched Linux	4,428,457	52.20%
Multiple Counter with Firewall	27,889	0.33%
Outliers	17,716	0.21%
Total	8,483,921	100.0%

Table 5.4: Clusters derived from similarity search process.

however my analysis shows that close to 25% of all Linux machines in my dataset had not been patched over that timespan, a not insignificant number.

Focusing on non-Linux machines, I find that machines using some form of Global IPID counter to generate IPID values make up close to 16% of my dataset. Machines using random IPID values make up approximately 7% of my dataset, an increase in the amount found by related work [70].

One interesting behavior that I found in my analysis was the number of machines using constant values for ICMP packets but not for TCP packets. These are the machines that I have classified as behaviors “with Firewall”. The generation of TCP packet IPID values for these machines covers a range of behaviors (e.g., Random IPIDs or Global IPIDs) but all of these machines use a constant value for their ICMP packet IPIDs. The most common value I saw was 1 but other values existed within my dataset as well. The machines “with Firewall” make up 10.38% of the addresses in my dataset, a not insignificant portion.

5.5 Discussion

In this section I present interesting behaviors I uncovered by the hierarchical clustering I carried out, compare the relative prevalence between common behaviors, and discuss interesting network and routing behaviors that impact IPID-based side-channels.

5.5.1 Machines Vulnerable to known IPID Side-Channels

One of the main goals of the work discussed in the Chapter, was to scan the entire IPv4 address space to gain a better understanding of the different IPID behaviors present on the Internet as a whole and to better understand how many machines may be using behaviors that are vulnerable to known IPID-based side-channels. One such side-channel is the vulnerability discussed in Chapter 4. The side-channel I described was present in the Linux kernel but was patched following my disclosure of the vulnerability. However, as discussed previously over 25% of the Linux machines in my dataset were running versions of the Linux kernel that had not been patched to mitigate the vulnerability I disclosed.

Prior work [6, 9, 16, 28, 47, 58] discussed IPID-based side-channels that relied on Global IPID values to function. As operating systems have transitioned away from the use of Global IPID values the number of machines these side-channels exist in has decreased but there still exists a sizeable portion of machines in my dataset that use Global IPID values. As discussed in Chapter 5.4.3 close to 16% of my dataset, 1,317,818 IPv4 addresses in total, appeared to be using a global IPID counter to generate IPID values. These machines are likely to contain IPID-based side-channels that would allow them to be used in the techniques mentioned above.

Based on the set of IPID-based side-channels I am aware of, and those I have discussed in this dissertation, I can make an estimate of the number of machines in my dataset which are containing exploitable side-channels. Combining the set of machines which appear to be using Global IPIDs and those Linux machines running kernel versions that were not patched to mitigate the vulnerability discussed in Chapter 4 I find that 2,827,759 (33.33%) addresses in my dataset are likely to contain an known, IPID-based side-channel. While this number is likely to decrease, as the Linux machines running kernels that had not been patched to mitigate the side-channel vulnerability discussed in Chapter 4 are patched, this still constitutes a large number of active machines on the Internet.

5.5.2 Comparison with Prior IPID Studies

Comparing my study with prior studies shows a continuing evolution of IPID generation methods on the Internet. The most recent studies of IPID generation methods are by Salutari et al. [70] and Pearce et al. [62]. Salutari et al. found that 18% of addresses used Global IPIDs, while Pearce et al. found that 16% of addresses used Global IPIDs. These results are inline with my own finding that approximately 16% of addresses in my dataset use Global IPIDs. This fits with an evolution of IPIDs away from Global IPID counters, likely due to changes in operating systems such as Linux. Earlier studies showed a much larger prevalence of Global IPIDs. A 2003 study by Mahajan et al. [55] found that 70% of addresses used Global IPIDs. In 2005, Chen et al. [16] placed the percentage of addresses using Global IPIDs at 38%. Gilad and Herzberg [35] found 57% of Top Level Domain TLDs used Global IPIDs, in 2011.

The largest area where my results differ from prior work is in the prevalence, or lack of, addresses using a constant IPID value. Salutari et al. [70] found that 34% of addresses they scanned, responded with a constant IPID value. Gilad and

Herzberg [35] found that 9% of addresses they scanned used a constant IPID value. My own results find that less than 1% of addresses, in my dataset, respond with a constant IPID value. The two most likely causes of this difference are continued operating system change and the presence of ICMP filter or traffic manipulation devices. As operating systems continue to change and improve, systems using constant counters are likely to either be phased out of usage or change to another system of IPID generation. This continued change could be the cause of the difference between the results found by Gilad and Herzberg (i.e., that 9% of addresses used a constant IPID value) and my own findings (i.e., that less than 1% of addresses use a constant IPID value). A likely reason for the large difference between my own findings and those of Salutari et al. is the presence of large numbers of machines in my dataset that appear to filter or change ICMP traffic to use a constant IPID value but use another IPID generation method for TCP traffic. As I discussed in Section 5.4.3, these addresses make up 10% of the addresses in my full dataset. While this does not fully account for the 34% of machines seen using a constant IPID value by Salutari et al. it does account for a large portion of the difference. Other differences in choice of addresses to scan, scanning methodology, or unforeseen network changes may help explain the remaining difference.

Another area where my results show continued evolution of IPID generation methods is the increase in the prevalence of random IPID values in my dataset, compared to prior work. Random IPID generation methods are unlikely to contain side-channels, assuming the generation method is sufficiently random, as a series of random values provides an off-path machine with no source of usable information. Gilad and Herzberg [35] found only 1% of address they scanned were using random IPID values. The most recent study, by Salutari et al. [70] found that 2% of address they scanned used random IPIDs. My own results show that approximately 7% of the addresses in my dataset use random IPID values, a noticeable increase. One possible explanation for the difference between my results and those of Salutari et al.

is that Salutari et al. limited their scan to ICMP packets only. As discussed above one large-scale behavior I noticed was the prevalence of addresses which used some kind of traffic filtering or manipulation mechanism for ICMP traffic. This resulted in a large number of addresses, 880,903 in total, using a constant IPID value for ICMP traffic but another IPID generation method for TCP. Salutari et al. would have classified any of these addresses as using a constant IPID value, due to their focus on ICMP traffic only. My study shows that addresses using random IPID values to be a much larger portion of the IPv4 addresses space.

5.6 Ethical Considerations

In order to avoid potentially exhausting resources on the machines I scanned I strove to follow established best practices. All TCP packets that I sent do not attempt to create a TCP connection and should all be immediately `RST`. This helps to ensure that my scan did not cause a denial-of-service (DoS) attack. ICMP packets were never sent in amounts greater than thirty-two per test to ensure that any resources used to respond to them or store ICMP fragments were kept to a minimum. In addition I maintained a blacklist of IP addresses and IP networks that were to never be scanned, based on opt-out requests from previous studies. Via reverse DNS all my measurement IP addresses pointed to a website which explained what my scans were and provided contact information to allow network operators to opt-out of any further scans of their IP addresses or IP networks. Emails pertaining to my scans sent to the university's network security and abuse team were also forwarded to me so I could remove those IPs who wished to not be scanned. I received five complaints asking that IP addresses or networks be added to my blacklist and each was added as soon as I received the email.

5.7 Conclusion

In this chapter I have presented the results of an IPv4 address space scan of IPID value generation and the prevalence of known IPID side-channels. I analyzed my scan data using hierarchical clustering to study the different types of IPID generation methods used to respond to my scan and the relative prevalence of each behavior in my dataset. Overall, I find 11 unique IPID behaviors in my dataset.

My scan differs from previous studies of IPID behavior in that it studies not only IPID differences *per-host* but also differences in IPID generation *per-protocol*. This is important because some operating systems (i.e., Linux) are known to consider both the source address and protocol when generating IPID values for outgoing packets. Whereas previous work showed a higher than expected rate of machines using constant IPID values, 34% of addresses, my results show that the number of machines using constant IPID values is much lower, less than 1% of addresses. This can be explained by the prevalence of firewalls or other networking devices which filter traffic differently, based on the protocol being used. In the case of my data, this takes the form of IP addresses responding with constant IPID values for ICMP traffic but using non-constant values for other protocols, such as TCP. By considering both per-host variations and per-protocol variations researchers studying IPID side-channels can gain a much clearer picture of which machines are likely to contain side-channels on the Internet.

Based on my analysis, I find that close to 35% of the IP addresses in my dataset use IPID generation methods that are likely to contain IPID side-channels. These include side-channels which allow for known attacks, such as the attack discussed in Chapter 4, or known network measurement techniques that rely on IPID side-channels, such as IPID-based Idle Scans. While this number is likely to decrease, as machines which had not been patched to mitigate the attack discussed in Chapter 4, this would still leave close to 17% of the addresses in my dataset using IPID generation methods that are likely to contain known IPID side-channels. This is similar to previous best estimates of 16-18% IPv4 machines using IPID generation methods containing side-channels but shows that there still exist a significant number of machines across the IPv4 address space likely to contain IPID-based side-channels.

In Chapter 3 and Chapter 4 I discussed techniques for using network side-channels that do not fully exhaust the shared, finite resources used to access an information side-channel. Both of these techniques targeted the Linux kernel's TCP networking stack. Due to the unique method the Linux kernel uses to generate IPID values I can provide an estimate of the number of machines that each technique could target. For the attack discussed in Chapter 4 this is approximately 18% of the IP addresses contained in my dataset. The round-trip-time measurement technique I discussed in Chapter 3 uses a side-channel in the Linux kernel's `SYN`-backlog. Based on my analysis, 70% of the IP addresses in my dataset are likely candidates for use with this technique. This represents a lower bound on the total number of addresses, as it is possible some of the addresses in my dataset using Global IPID values could be older versions of Linux which have the required `SYN`-backlog behavior but are not recent enough to use the IPID behavior discussed in Chapter 4. This shows that non-exhaustive side-channel techniques can be used against a large number of IP addresses and provide the first IPv4-scale analysis of the extent to which the technique discussed in Chapter 4 might be usable.

I have discussed an IPv4 address space "mass-scan" of IPID generation across differences in both changes to the source IP address and IP packet protocol. I analyzed the results of this scan using hierarchical clustering to study both the total number of common generation methods and their relative prevalence. I have shown that scans which use only a single IP protocol or IP address can provide an incomplete picture of IPID generation methods "in the wild". Finally, I have discussed how my results provide a lower bound on the prevalence of known, non-exhaustive IPID side-channels and provided an estimate of the number of IP addresses which could be used for the non-exhaustive, side-channel technique discussed in Chapter 4.

Chapter 6

Future Work

This dissertation presented research that builds the premise that: Network side-channels can be used to measure machine and connection state beyond connectivity without fully exhausting shared resources. There exist a number of different directions in which future research into this area could be taken. One possible area is to measure lower level connection features beyond round trip time. Features such as number of routing hops between off-path machines, detecting the presence of load-balancers, and counting load balancers would all be of interest to researchers trying to get a better understanding of network topology. As an example consider a hypothetical side-channel that allows researchers to count changes to the time-to-live (TTL) field in IPv4 packet headers. Using this side-channel researchers could easily determine how many machines were involved in routing a given packet. Whether such side-channels exist and can be used to carry out off-path measurement without fully exhausting shared resources is an area that has seen limited research and could provide a wealth of possible techniques.

The research in this dissertation studied side-channels in IPv4 and TCP. Both of these protocols exist in the lower levels of the OSI model of computer networks [85], Layers 3 and 4. Are there side-channels present in higher layer protocols (e.g., HTTP, SMTP, DNS, or DHCP)? And could such side-channels be leveraged to carry out off-path measurements or attacks? These are both open questions that have seen comparatively less research than side-channels in lower level protocols like IPv4 and TCP. There has been limited prior work that targets these higher level protocols. Instead, the side-channels used are often in the lower level protocols the higher level protocols exists on top of.

Many network side-channels are likely to be unique to specific operating systems or even specific operating system versions. As discussed in Chapter 4, the Linux kernel's network stack has used three different versions of IPID generation code during its lifetime. Each of these different versions contained documented side-channels. One avenue for future work is to use the side-channels discussed in this dissertation and in other research as a method of performing off-path operating system fingerprinting. As an example, the Linux kernel and the Microsoft Windows operating system likely both contain network side-channels however each will contain different side-channels. The differences in which side-channels can be used could help to identify the operating system of a remote host using only off-path packets. In addition such tests could be added into existing tools, such as nmap's OS fingerprinting feature, to help augment its accuracy or better distinguish between hosts with similar behaviors. The development of such techniques could allow researchers to perform OS fingerprinting of machines that are behind firewalls or only respond to a limited set of packets, making traditional direct OS fingerprinting difficult or impossible.

Another area for future work that follows from the mass-scan carried out in Chapter 5 is to study other packet fields across the entire Internet. Fields such as TCP sequence and acknowledgement number generation, TCP ephemeral ports, and ICMP ping sequence and id numbers could all contain side-channels if not generated in a sufficiently robust manner. In addition, to my knowledge there has not been an

Internet wide study of `SYN`-backlog behavior or IPv4 fragmentation cache behavior. As discussed in Chapter 3 and by Knockel and Crandall [45] both of these structures are often implemented as a shared buffer that can be used as side-channels. An Internet scale study of the different behaviors within these structures could point to other side-channels within previously unstudied network stacks. One difficulty in carrying out such a study is that side-channels within these shared buffers often require the buffer be completely filled or exhausted. Finding a technique that can study their behavior without fully exhausting these shared resources would be needed to ensure any such study did not cause a denial-of-service attack on any scanned machines.

One area that has received less research is the presence of side-channels in the IPv6 networking stack of operating systems. All of the work in this dissertation focused on the IPv4 networking stack. However, as IPv6 adoption continues to increase more machines will be communicating using IPv6. Since IPv6 has not been studied as much as IPv4 are there may side-channels within various IPv6 implementations that are not present in IPv4? Is there any overlap between side-channels present in IPv4 and IPv6? ONIS [83] and Morbitzer's work [58] showed that the Linux kernel's IPv6 IPID implementation shared similar side-channels as its IPv4 implementation. Are there other such "dual-stack" side-channels that can be used to measure complex state, such as detecting TCP connections? Further study into both IPv6 side-channels and the overlap between side-channels in IPv4 and IPv6 could help to answer these questions.

Chapter 7

Conclusion

Network side-channels are information flows within shared, finite resources in the networking stack of computer operating systems. These resources often take the form of shared, global counters, global packet rate-limits, or memory buffers used to populate packet header fields or queue incoming packets for later processing. Prior work studying network side-channels often relied on completely exhausting these resources in order to gain information on the state of remote or off-path machines. In this dissertation I have discussed techniques which depart from this method of exhausting side-channel resources and instead extract information about the state of remote, off-path machines without fully exhausting shared, finite resources.

At the start of this dissertation I introduced the thesis statement that underlies my dissertation research: *Network side-channels can be used to measure machine and network state beyond connectivity without fully exhausting the shared, finite resources that create them.* In Chapter 3 I detailed my research into the use of a side-channel in the Linux kernel's `SYN`-backlog to measure off-path round trip time (RTT). This work supports my thesis statement by providing a method to measure round trip time, a network property independent of connectivity, without fully filling, and thereby exhausting, the shared memory buffer used to implement the kernel's `SYN`-backlog. This technique is completely off-path (i.e., it can measure round trip time between two remote hosts without having access to either host or the traffic sent between them), has accuracy comparable to other techniques that require the use of additional machines beyond those whose RTT is being measured, and does so without fully exhausting all available space in the `SYN`-backlog.

Chapter 4 discussed my work using a side-channel present in the Linux kernel's handling of IPID counters to detect the presence of active TCP connections between two remote off-path machines. This work takes advantage of the Linux kernel's IPID implementation which uses one of two different IPID counters for TCP traffic. One shared counter used for TCP packets that are not part of established TCP connections and another, *per-connection*, counter used when a connection has been established. Using IPID changes to determine which counter is being used, an off-path attacker can check possible TCP/IP 4-tuples (i.e., source and destination IPv4 addresses and source and destination TCP ports) and detect the presence of an active TCP connection. This work supports my thesis statement by demonstrating a technique to measure machine state beyond connectivity, namely the presence of an established TCP connection, and does so without exhausting a shared memory buffer or global rate-limit as done by related work. In addition this work is the first that I am aware of to detail the presence and usage of a side-channel in a network stack that makes use of "hybrid" counters or shared resources. That is a system that uses a mix of global, shared resources and individual, per-connection or per-destination resources depending on system networking state.

I discussed an Internet scale "mass scan" which studied IPv4 Identification field value generation and uniqueness across different destination hosts and protocols in Chapter 5. This work helps to support my thesis statement by providing an Internet scale study of IPID value generation and uniqueness, with a focus on detecting the presence of likely side-channels that could be used without fully exhausting shared resources on remote machines. This work focuses on attempting to measure the dependence between the IPID values sequentially sent to separate IPv4 address and IPv4 next level protocols. When such a dependence exists it points to the possible presence of a side-channel because the dependence allows an off-path machine to determine whether or not off-path traffic caused a change in a given IPID. This is due to the dependence between the two IPID values causing one to change based on the value of the other. This dependence allows an off-path machine to extract information about whether or not off-path traffic occurred without fully exhausting network resources by using the IPID dependence to predict what IPIDs would be in cases where traffic is and is not sent. Based on which of these cases the measured IPIDs best match an off-path machine can determine which case occurred and use the IPID field as a side-channel.

In this dissertation, I have discussed three different directions of research which studied the use of network side-channels to carry out off-path network measurements. My work includes techniques for measuring non-trivial network state and attacks without requiring the complete exhaustion of the shared, finite resources that give rise to these side-channels. This illuminates new directions that network side-channel research can be taken and highlights that using network side-channels does not require fully exhausting a shared, finite resource.

References

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

[2] Nmap: the Network Mapper. `https://nmap.org`.

[3] OONI: Open Observatory of Network Interference. `https://ooni.toproject.org`.

[4] Alexa. Alexa Top 500 Global Sites. `https://www.alexa.com/topsites`.

[5] Y. Angel and P. Winter. obfs4 (the obfourscator). `https://gitweb.torproject.org/pluggable-transports/obfs4.git/tree/doc/obfs4-spec.txt`, 2014.

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

[7] A. d. A. Antonio, R. M. Leao, and E. d. S. e Silva. A Non-cooperative Active Measurement Technique for Estimating the Average and Variance of the One-Way Delay. In *NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*, pages 1084–1095. Springer, 2007.

[8] S. Axelsson. The Base-rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, CCS '99, pages 1–7, New York, NY, USA, 1999. ACM.

[9] 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.

[10] D. J. Bernstein. Cache-timing attacks on AES. 2005.

[11] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks Against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[12] F. Cangialosi, D. Levin, and N. Spring. Ting: Measuring and exploiting latencies between all Tor nodes. In *Proceedings of the 2015 Internet Measurement Conference*, pages 289–302. ACM, 2015.

[13] 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.

[14] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, and L. M. Marvel. Off-Path TCP Exploits of the Challenge ACK Global Rate Limit. *IEEE/ACM Transactions on Networking*, 26(2):765–778, 2018.

[15] R. Castro, M. Coates, G. Liang, R. Nowak, and B. Yu. Network Tomography: Recent Developments. *Statistical science*, pages 499–517, 2004.

[16] 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.

[17] W. Chen and Z. Qian. Off-path TCP exploit: How wireless routers can jeopardize your secrets. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[18] 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.

[19] M. Coates, A. Hero, R. Nowak, and B. Yu. Internet tomography. *Signal Processing Magazine, IEEE*, 19(3):47–65, 2002.

[20] M. Cotton, L. Eggbert, J. Touch, M. Westerlund, and S. Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335 (Draft Standard), Aug. 2011.

[21] 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.

[22] A. Dainotti, K. Benson, A. King, M. Kallitsis, E. Glatz, X. Dimitropoulos, et al. Estimating Internet Address Space Usage through Passive Measurements. *ACM SIGCOMM Computer Communication Review*, 44(1):42–49, 2013.

[23] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[24] E. Dumazet. inetpeer: get rid of ip_id_count. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/commit/?id=73f156a6e8c1074ac6327e0abd1169e95eb66463`, 2014.

[25] E. Dumazet. ip: make ip identifiers less predictable. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/commit/?id=04ca6973f7c1a0d8537f2d9906a0cf8e69886d75`, 2014.

[26] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *USENIX Security Symposium*, volume 8, pages 47–53, 2013.

[27] R. Ensafi, D. Fifield, P. Winter, N. Feamster, N. Weaver, and V. Paxson. Examining how the Great Firewall discovers hidden circumvention servers. In *Proceedings of the 2015 Internet Measurement Conference*, pages 445–458. ACM, 2015.

[28] 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.

[29] 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.

[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] O. Gasser, Q. Scheitle, S. Gebhard, and G. Carle. Scanning the IPv6 Internet: Towards a Comprehensive Hitlist. *arXiv preprint arXiv:1607.05179*, 2016.

[33] 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.

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

[35] Gilad, Yossi and Herzberg, Amir. 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] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[37] 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.

[38] 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.

[39] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium*, volume 8, page 1, 2012.

[40] 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.

[41] D. Kaminsky. Doxpara: Paketto keiretsu (scanrand). `https://dankaminsky.com/2002/11/18/77/`, 2002.

[42] 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.

[43] R. A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Trans. Comput. Syst.*, 1(3):256–277, 1983.

[44] A. Klein. OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability, 2007.

[45] 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.

[46] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

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

[48] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[49] R. Lee and J. Louis. Unicornscan. `http://www.unicornscan.org/`, 2008.

[50] Light Pink Book. A guide to understanding covert channel analysis of trusted systems, version 1. NCSC-TG-030, Library No. S-240,572, November 1993. TCSEC Rainbow Series Library.

[51] S. B. Lipner. A comment on the confinement problem. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles*, pages 192–196, New York, NY, USA, 1975. ACM Press.

[52] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[53] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[54] 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.

[55] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet Path Diagnosis. *ACM SIGOPS Operating Systems Review*, 37(5):106–119, 2003.

[56] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson. China's great cannon. *Citizen Lab*, 10, 2015.

[57] J. C. Matherly. Shodan the computer search engine. *Available at [Online]: http://www. shodanhq. com/help*, 2009.

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

[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] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for internet hosts. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 173–185. ACM, 2001.

[61] V. Paxson. End-to-end internet packet dynamics. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 139–152. ACM, 1997.

[62] 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.

[63] C. Percival. Cache missing for fun and profit, 2005.

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

[65] J. Postel et al. RFC 791: Internet protocol. 1981.

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

[67] Qian, Zhiyun and Mao, Zhuoqing Morley. 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.

[68] 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.

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

[70] F. Salutari, D. Cicalese, and D. J. Rossi. A closer look at IP-ID behavior in the Wild. In *Proceedings of the 19th International Conference on Passive and Active Measurement*, PAM'18, pages 243–254. Springer-Cham, 2018.

[71] D. X. Song, D. Wagner, and X. Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium*, volume 2001, 2001.

[72] The Tor Project. Tor Metrics.

[73] L. Torvalds. Linux Kernel V4.16. `https://github.com/torvalds/linux/blob/v4.16/include/net/ip.h\#393`, March 2018.

[74] J. Touch. RFC 6864: Updated Specification of the IPv4 ID Field. 2013.

[75] Y. Tsang, M. Coates, and R. D. Nowak. Network delay tomography. *Signal Processing, IEEE Transactions on*, 51(8):2125–2136, 2003.

[76] Y. Vardi. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association*, 91(433):365–377, 1996.

[77] 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.

[78] N. Weaver, R. Sommer, and V. Paxson. Detecting forged TCP reset packets. In *NDSS*, 2009.

[79] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, pages 2–7, 1991.

[80] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, volume 1, pages 22–25, 2014.

[81] 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.

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

[83] Zhang, Xu and Knockel, Jeffrey and Crandall, Jedidiah R. ONIS: Inferring TCP/IP-based Trust Relationships Completely Off-Path. In *IEEE INFOCOM 2018*, 2018.

[84] 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.

[85] H. Zimmermann. OSI reference model-the ISO model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.