Counting Packets Sent Between Arbitrary Internet Hosts

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

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

Abstract

In this paper we demonstrate a side-channel technique to infer whether two machines are exchanging packets on the Internet provided that one of them is a Linux machine. For ICMP and UDP exchanges, we require that at least one machine is a Linux machine, and for TCP connections, we require that at least the server is a Linux machine. Unlike many side-channel measurement techniques, our method does not require that either machine be *idle*. That is, we make no assumptions about either machines' traffic patterns with respect to other hosts on the Internet. We have implemented our technique, and we present the results of a proof-of-concept experiment showing that it can effectively measure whether hosts are communicating.

1 Introduction

One of the most basic assumptions that privacy and anti-censorship technologies commonly make is that it is not possible for an adversary to count the packets sent from one machine to another on the Internet without being in the routing path between the two machines. In this paper we challenge this basic assumption.

Every IPv4[1] packet has a 16-bit field called an *IP identifier* (IPID). If an IP packet is too large to travel over a link, it may be broken up into smaller *fragments* that are then sent over that link. The final destination is able to reassemble the original datagram by collecting all incoming fragments into a *fragment cache* and using the IPID to determine which fragments belong to which original datagram.

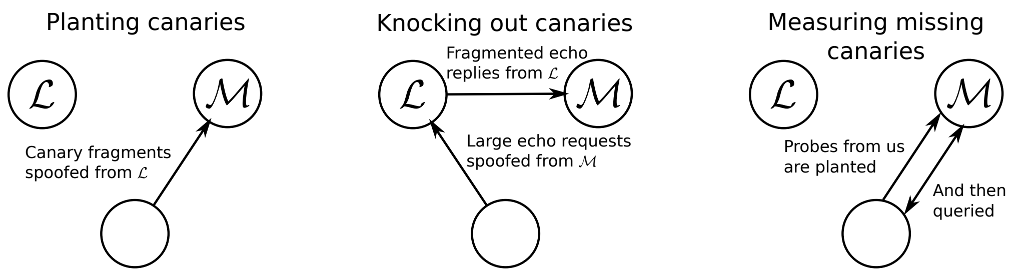
Different machines use varying algorithms for assigning an IPID to outgoing packets. Some machines such as Windows assign IPID's by using a *global IPID counter*. For every outgoing packet, the value of this counter is assigned to the outgoing packet's IPID, and the counter is then incremented so that the next outgoing packet's IPID will be plus one $\pmod{2^{16}}$.

Antirez [1] proposed a technique called an "idle scan" that uses the global IPID as a side channel to scan ports. In this method, an "attacker" attempts to discover whether a port on a "victim" is open or closed. By utilizing information flow between the status of the victim's port and the global IP identifier counter of a "zombie," the attacker infers whether the victim's port is open without having to send any packets to the victim containing the attacker's address. The attacker instead sends TCP SYN's spoofed from the zombie to the victim, faking a connection request from the zombie to the victim. If the victim's port is open, the victim will send a TCP SYN-ACK to the zombie, attempting to complete the connection. However, the zombie, never actually initiating the connection, will send a TCP RST to the victim. Sending the RST increments the zombie's global IPID counter, which the attacker can measure.

In addition to assuming that the zombie has a global IPID counter, this scan also assumes that the zombie is *idle*, *i.e.*, that the zombie would not otherwise be sending any packets. This is so that increases to the zombie's global IPID counter can be unambiguously attributed to RST's sent by the zombie to the victim.

Before this scan's discovery, the Linux kernel had a global IPID counter, but in response to its discovery, the kernel's developers replaced the global IPID counter with two new types of counters which we refer to as *per-connection* and *per-destination* counters. Per-connection counters are counters that the kernel maintains for each TCP connection, whereas per-destination counters are counters used for all other packets outside of connections. The kernel maintains a per-destination counter for each host it recently sent packets to in a cache. It was believed that this implementation defeated Antirez's scan, since any reset packets sent from the zombie to the victim will draw from a different counter than the attacker's counter.

[1]Hereafter referred to simply as IP, as we make no claims about IPv6.



Planting canaries
$\mathcal{L}$
$\mathcal{M}$
Canary fragments spoofed from $\mathcal{L}$
Knocking out canaries
Fragmented echo replies from $\mathcal{L}$
$\mathcal{L}$
$\mathcal{M}$
Large echo requests spoofed from $\mathcal{M}$
Measuring missing canaries
$\mathcal{L}$
$\mathcal{M}$
Probes from us are planted
And then queried

Figure 1: Selected illustrations of packets we send at different stages of our algorithm to $\mathcal{L}$ or $\mathcal{M}$.

In this paper, we will show how to infer a Linux machine's per-destination counters using a different side channel. We utilize the information flow between a Linux machine $\mathcal{L}$'s per-destination IPID counter for machine $\mathcal{M}$ and $\mathcal{M}$'s fragment cache. Discovering the value of per-destination counters reintroduces the possibility of idle scans and eliminates the requirement that the zombie be idle. **Moreover, it also introduces a new attack where the existence of ICMP, UDP, and TCP communication between $\mathcal{L}$ and $\mathcal{M}$ can be inferred, and the number of packets sent in a given time period can be counted.**

The rest of the paper is structured as follows: we explain our method for inferring per-destination counters and inferring communication between $\mathcal{L}$ and $\mathcal{M}$ in Section 2. We then describe our proof-of-concept experiment and its results in Section 3, followed by related work in Section 4, and our discussion and our conclusions in Section 5.

2 Implementation

To measure communication between $\mathcal{L}$ and $\mathcal{M}$, our method first infers the value of $\mathcal{L}$'s per-destination counter for $\mathcal{M}$, which we call $C_{\mathcal{L}\to\mathcal{M}}$. At a high level, this consists of the following (see Figure 1 for an illustration):

1. We place fragments called *canaries* in $\mathcal{M}$'s fragment cache by sending fragments spoofed from $\mathcal{L}$ to $\mathcal{M}$ with IPID's carefully chosen to guess $C_{\mathcal{L}\to\mathcal{M}}$.
2. We spoof large echo requests from $\mathcal{M}$ to $\mathcal{L}$, causing $\mathcal{L}$ to reply with fragmented echo replies to $\mathcal{M}$ with IPID's chosen from $C_{\mathcal{L}\to\mathcal{M}}$.
3. If the IPID's of the echo replies match any of those of the canaries that we placed, then those canaries will be knocked out of $\mathcal{M}$'s fragment cache.
4. We then send and query *probes* from us to $\mathcal{M}$ to measure $S_{\mathcal{M}}$, the number of missing canaries.
5. If $S_{\mathcal{M}}$ is significantly high, we know that some of our canaries' guesses of $C_{\mathcal{L}\to\mathcal{M}}$ were correct.

By repeatedly using this technique to guess values of $C_{\mathcal{L}\to\mathcal{M}}$, we eventually discover $C_{\mathcal{L}\to\mathcal{M}}$.

Once we know $C_{\mathcal{L}\to\mathcal{M}}$, we track its value to count the number of ICMP or UDP packets $\mathcal{L}$ has sent to $\mathcal{M}$. Additionally, by spoofing carefully crafted TCP packets from $\mathcal{M}$ to $\mathcal{L}$, we can measure the existence of a TCP connection by tracking $C_{\mathcal{L}\to\mathcal{M}}$.

In the remainder of this section, we expound on our method in three parts. In the first part, we detail how to infer $C_{\mathcal{L}\to\mathcal{M}}$ using $S_{\mathcal{M}}$. In the second part, we set out how to use probes to measure $S_{\mathcal{M}}$ on a variety of operating systems. In the final part, we explain how to use knowledge of $C_{\mathcal{L}\to\mathcal{M}}$ to test if $\mathcal{L}$ is communicating with $\mathcal{M}$.

2.1 Measuring $C_{\mathcal{L}\to\mathcal{M}}$

To find $C_{\mathcal{L}\to\mathcal{M}}$, we utilize information flow between $C_{\mathcal{L}\to\mathcal{M}}$ and $\mathcal{M}$'s fragment cache. Like many other side-channel measurement techniques such as the idle scan, our technique utilizes packet *spoofing*, *i.e.*, sending packets with a forged source address. Our technique also utilizes the fact that fragment caches have finite memory, *i.e.*, they can only hold so many fragments at once. We will infer $C_{\mathcal{L}\to\mathcal{M}}$ by measuring $S_{\mathcal{M}}$, the number of canaries missing in $\mathcal{M}$'s fragment cache, by spoofing packets from $\mathcal{M}$ to $\mathcal{L}$.

The packets we spoof from $\mathcal{M}$ to $\mathcal{L}$ are large ICMP echo requests or "pings." ICMP echo requests contain a variable-length data section, and this data is always echoed back in the echo reply. When $\mathcal{L}$ receives the spoofed packet, $\mathcal{L}$ (believing $\mathcal{M}$ to be the original sender) sends the echo reply to $\mathcal{M}$. We choose a size for our spoofed echo requests large enough so that $\mathcal{L}$'s echo replies to $\mathcal{M}$ will become fragmented. The IPID of this fragmented reply will be $C_{\mathcal{L}\to\mathcal{M}} - 1$ (since our spoofed packet has incremented $C_{\mathcal{L}\to\mathcal{M}}$), and thus by knowing its value, we know the value of $C_{\mathcal{L}\to\mathcal{M}}$.[2]

Before sending our spoofed echo requests, we first place *canaries* in $\mathcal{M}$'s fragment cache. These canaries are fragments that contain only the first 400 bytes of datagrams with spoofed source addresses from $\mathcal{L}$. We carefully choose the IPID's of these canaries. Any incoming

[2]For simplicity, from here on we exclude our spoofed packets' effects on $C_{\mathcal{L}\to\mathcal{M}}$ from our description, but an implementation must maintain bookkeeping to automatically track its effects on $C_{\mathcal{L}\to\mathcal{M}}$.

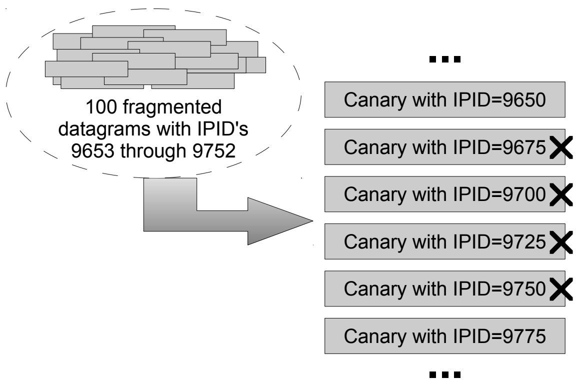


Figure 2: How $r = 100$ fragmented echo replies from $\mathcal{L}$ to $\mathcal{M}$ affect $\mathcal{M}$'s fragment cache.

fragmented echo reply from $\mathcal{L}$, if it has the same IPID as the canary, will overwrite and complete the canary's datagram, clearing its entry from the cache and increasing $S_{\mathcal{M}}$ by one. By choosing our canaries' IPID's, we can test different values of $C_{\mathcal{L}\to\mathcal{M}}$ based on its effect on $S_{\mathcal{M}}$.

Before describing how to measure $S_{\mathcal{M}}$, we will describe how to efficiently find $C_{\mathcal{L}\to\mathcal{M}}$ assuming that we can query for $S_{\mathcal{M}}$. One naive approach to measure $C_{\mathcal{L}\to\mathcal{M}}$ is to, for each of the 2^{16} possible values of $C_{\mathcal{L}\to\mathcal{M}}$, place a canary on $\mathcal{M}$ with IPID of that value, spoof an ICMP echo request from $\mathcal{M}$ to $\mathcal{L}$, and measure $S_{\mathcal{M}}$ to determine if $\mathcal{M}$ is missing a canary in its cache. However, this is time consuming, and querying $S_{\mathcal{M}}$ will be expensive, so we will wish to reduce the number of queries $S_{\mathcal{M}}$ as much as possible.

A simple improvement is to, instead of spoofing one ICMP echo request from $\mathcal{M}$ to $\mathcal{L}$, spoof r requests. This allows r values of $C_{\mathcal{L}\to\mathcal{M}}$ to be tested per query of $S_{\mathcal{M}}$. Alternatively, one may instead place c canaries, allowing c values of $C_{\mathcal{L}\to\mathcal{M}}$ to be tested per query of $S_{\mathcal{M}}$.

These two improvements can be combined. By spoofing r echo requests and placing c canaries whose IPID's $x_1, \ldots, x_c$ are r apart, *i.e.*, such that all $x_i - x_{i-1} = r$, we can test an interval of $c \cdot r$ values, $(x_1 - r, x_c]$, per single query of $S_{\mathcal{M}}$. See Figure 2 for an illustration.

This test will increase $S_{\mathcal{M}}$ by 1 when $C_{\mathcal{L}\to\mathcal{M}}$ is in the tested interval. However, we will want to increase $S_{\mathcal{M}}$ by more than 1 so that the effect of our test will be easily distinguished. If we wish our test to eliminate k canaries when $C_{\mathcal{L}\to\mathcal{M}}$ is in the tested interval, then we must increase the density of our canaries' IPID's by a factor of k, reducing our test interval size I to

$$I(c, r, k) = (c - (k-1)) \left\lfloor \frac{r}{k} \right\rfloor,$$

testing the interval

$$\left(x_1 - \left\lfloor \frac{r}{k} \right\rfloor, x_c - (k-1)\left\lfloor \frac{r}{k} \right\rfloor\right].$$

Due to bandwidth restrictions and the implementation and size of $\mathcal{M}$'s fragment cache, r and c cannot be arbitrarily large. Rather, they have max sizes $r_{\max}$ and $c_{\max}$. We can calculate the largest testable interval $I_{\max}(k)$ as

$$I_{\max}(k) = I(c_{\max}, r_{\max}, k) = (c_{\max} - (k-1)) \left\lfloor \frac{r_{\max}}{k} \right\rfloor.$$

For testing intervals of size less than $I_{\max}(k)$, we keep c and r fixed, provided that c does not exceed the tested interval size. This allows our tests to cause $\mathcal{M}$ to remove $k' > k$ canaries.

To efficiently find $C_{\mathcal{L}\to\mathcal{M}}$, we might want to perform binary search. However, we found that using a type of 3-ary search that splits the interval evenly into 3 different tested sections is more robust, namely at handling the case when $C_{\mathcal{L}\to\mathcal{M}}$ lay near the boundary of two tested sections. Instead of searching for the section with k' missing canaries, we instead search for the section or two adjacent sections with more than $k'/2$ missing canaries. If two adjacent sections are found, their combined section is recursively searched. (If any other pattern of sections is found whose tests resulted in more than $k'/2$ missing canaries, then we consider this anomalous, and our algorithm aborts the binary search and fails.)

Often $I_{\max}(k)$ is too small to perform 3-ary search over the initial search space of all 2^{16} possible values. In this event, we first perform one round of n-ary search, where $n = \lceil 2^{16}/I_{\max} \rceil$. We use the same criteria as above to determine which interval to recursively search.

Once the searched interval size is 20 or less, we perform a linear search over the remaining possible values in ascending order. For each of these possible values, we place one canary testing whether that value is $C_{\mathcal{L}\to\mathcal{M}}$, then wait one second, spoof one request, and then wait another second. We then repeat this another four times before querying $S_{\mathcal{M}}$.[3] If our guess is correct, we will eliminate 5 canaries. We select the first tested value that is missing 3 or more canaries to be $C_{\mathcal{L}\to\mathcal{M}}$.

Once we know $C_{\mathcal{L}\to\mathcal{M}}$, we must keep it from expiring from inactivity, as $\mathcal{L}$ may time out the counter's storage if its value has not been recently read by the kernel. One way we can indefinitely keep its counter from timing out on $\mathcal{L}$ is by spoofing echo requests from $\mathcal{M}$ to $\mathcal{L}$. However, this requires some bookkeeping, as each echo request will also increase $C_{\mathcal{L}\to\mathcal{M}}$. We found that we can keep $C_{\mathcal{L}\to\mathcal{M}}$'s counter from timing out without modifying its value by spoofing a fragment from $\mathcal{M}$ to $\mathcal{L}$ every minute. (Linux uses the same data structure that contains $C_{\mathcal{L}\to\mathcal{M}}$ to also maintain a counter of the number of fragments recently received from each host.) Note that the

[3] It may seem as if each of these canaries will fill the same entry in $\mathcal{M}$'s fragment cache, but each of our spoofed requests actually increments $C_{\mathcal{L}\to\mathcal{M}}$, and so our implementation will automatically increment its guessed IPID's too.

spoofed fragment must not contain the first 8 bytes of the datagram, else when the fragment expires in $\mathcal{L}$'s cache, $\mathcal{L}$ may send $\mathcal{M}$ an ICMP "reassembly time exceeded" message, incrementing $C_{\mathcal{L}\to\mathcal{M}}$ after all.

2.2 Measuring $S_{\mathcal{M}}$

Now we will discuss how we can query the number of missing canaries $S_{\mathcal{M}}$ in $\mathcal{M}$'s fragment cache. Although this procedure varies depending on $\mathcal{M}$'s operating system, all procedures we discuss will make use of sending *probes* to $\mathcal{M}$. Each probe consists of two parts: *planting* the probe in $\mathcal{M}$'s fragment cache and *querying* the probe to test if it is in there.

We have implemented two probes. The first type is a TCP ACK probe. We plant the probe by sending $\mathcal{M}$ the first 400 bytes of a TCP datagram with only the ACK flag set. We query the probe by simply sending the remaining bytes to complete the datagram and waiting for a TCP RST in response. The second type is an ICMP echo probe. We plant the probe by sending $\mathcal{M}$ the first 400 bytes of an ICMP echo request. We query the probe by again simply sending the remaining bytes and waiting for an ICMP echo reply.[4]

Choosing which probe to use is first determined by which probes $\mathcal{M}$ responds to. For example, $\mathcal{M}$ may filter blind ACK's, making the TCP ACK probe ineffective. In general, we prefer the TCP ACK probe to the ICMP echo probe because, unlike the ICMP echo probe whose response is the same size as the request, the ACK's RST response is of a fixed, small size, reducing $\mathcal{M}$'s required upstream bandwidth.

Now we will show how to use probes to query $S_{\mathcal{M}}$ across a variety of operating systems.

2.2.1 Windows

Windows XP implements its fragment cache as a queue. Its queue has a configurable limit of n datagram entries (100 by default), and once its queue is full, it no longer accepts new datagram entries until another has been removed due to either being completed or due to timing out [10].

For this fragment cache, we place up to $c_{\max} = n$ canaries. To measure $S_{\mathcal{M}}$, we then plant $11n/10$ probes. For each canary that is missing, we will fit one additional probe into the fragment cache. We then query all probes in the reverse order that we had originally sent them. If p is the number of probes that respond, then $S_{\mathcal{M}} = p - (n - c)$.

We found via experimentation and reverse engineering that Windows Vista, 7, 8, and 8.1 have a more complicated fragment cache implementation. Their cache has a limit of N bytes for storing all fragments and their corresponding data structures, where N is 1/128 the number of bytes of the Windows machine's physical RAM. Once the cache becomes over half full, if the cache is presently using M bytes of storage, incoming fragments are probabilistically rejected with probability $2M/N - 1$. Although this cache still leaks information about $S_{\mathcal{M}}$, the random component in deciding whether to accept fragments makes measuring $S_{\mathcal{M}}$ difficult. We have not yet implemented a measurement for $S_{\mathcal{M}}$ for these newer Windows caches.

2.2.2 FreeBSD and OS X 10.9

We found via experimentation and source code analysis that FreeBSD and OS X 10.9 implement their fragment caches as a 64-bucket hash table. Incoming fragments are hashed into buckets according to the hash function

$$h(src, id) = id \oplus (src \ \& \ 0\text{xf}) \oplus ((src \ \& \ 0\text{xf00}) >> 4)$$

(mod 64), where *src* is the source address of the incoming fragment in host byte order and *id* is its IPID.

Both caches have a configurable limit of n datagram entries (800 by default in FreeBSD, 1024 in OS X). Unlike Windows, when the fragment cache is full and a fragment is received for a new datagram, it is not rejected, but rather another datagram entry is chosen to be *evicted* from the cache to make room for the new entry.

If the bucket that the incoming fragment was hashed to is non-empty, then the oldest fragment in that bucket is evicted. Otherwise, the hash table's buckets are scanned in ascending order, and the oldest fragment is evicted from the first non-empty bucket found.

When probing these caches, we found that this eviction behavior was difficult to analyze. To simplify our analysis, we might choose IPID's for our probes such that our probes always hash to bucket 0, thus emulating LRU eviction for our probes. However, this only allows us to plant $2^{16}/64 = 1024$ probes, which is insufficient for probing OS X's default cache size. Instead, we alternate sending probes that hash to buckets 0 and 1, allowing us to plant 2048 probes, which is enough to probe OS X.

To measure $S_{\mathcal{M}}$, before we place our canaries, we first plant $n/2$ probes in $\mathcal{M}$'s fragment cache. We then place up to $c_{\max} = 2n/5$ canaries. To later measure $S_{\mathcal{M}}$, we plant another $3n/5$ probes. For each additional canary that is missing, one fewer probe that we had originally planted will be evicted by these probes. We then query all $11n/10$ probes in the reverse order that we had originally sent them. If p is the number of probes that respond, then again we have $S_{\mathcal{M}} = p - (n - c)$.

2.2.3 Linux

We studied the fragment caches of Linux across a range of versions via experimentation and source code analy-

[4]Another possible probe would be to send no packets in our query and wait for an ICMP "reassembly time exceeded" message to test whether our probe was still present in $\mathcal{M}$'s cache, but this type of probe requires more time, and many operating systems do not send these messages, and so we did not implement it.

sis. All versions of Linux that we analyzed use a hash table to implement their fragment caches. The Linux fragment cache has a configurable limit of N bytes of storage. Our canaries and probes are both the same size, 400 bytes, so in our measurement, this is equivalent to a fragment cache with a datagram entry limit of n entries, where n is some value less than $N/400$ that can be experimentally measured. We will hereafter model the Linux cache as thus.

When the Linux fragment cache is full and needs more room for an incoming fragment, unlike FreeBSD, which evicts one entry, Linux evicts entries until there are only $n' < n$ entries in the queue. The value of n' is configurable, but by default $n' = 3n/4$. This behavior complicates our analysis, since, unlike with FreeBSD, we cannot exactly measure $S_{\mathcal{M}}$, as our probes are no longer being evicted one at a time as a function of the remaining canaries. However, we will show how to still test whether $S_{\mathcal{M}}$ is greater than some value. Namely, we will show how to still test whether $S_{\mathcal{M}}$ is greater than $k'/2$.

Before we place our canaries, we first plant $n/2$ probes in $\mathcal{M}$'s fragment cache and then place up to $c_{\max} = n/2$ canaries. Afterwards, to test $S_{\mathcal{M}}$, we plant another $n/2 - c + k'/2 + 1$ probes. Unless greater than $k'/2$ are missing, this will exceed the fragment cache's limits, and $n - n'$ probes will be removed. We then query the probes in the reverse order that they were sent. If p is the number of probes that responded, we conclude that $S_{\mathcal{M}} > k'/2$ iff $p < n - c + k'/2 + 1 - (n - n')/2$, *i.e.*, if more than $(n - n')/2$ probes are missing.

The fragment cache in Linux 2.4.21 and later maintains an LRU queue to ensure that entries can be evicted in LRU order; however, the eviction order in versions before 2.4.21 requires special handling similar to FreeBSD. These older Linux versions reduce the number of cache entries to no more than n' by removing the oldest entry (if present) from all 64 buckets in the cache's hash table, then checking if enough entries have been removed yet, and finally repeating if necessary. These versions use a hash function that takes the xor of all bytes of the destination address, source address, IPID, and protocol number of the incoming fragment (mod 64). To test $S_{\mathcal{M}}$, to ensure that only our probes are removed by our measurement and not our canaries, we first plant $b = 64\lceil n/(4 \cdot 64)\rceil$ probes such that each sequence of 64 probes hashes to each of the 64 different buckets. By planting $\lceil n/(4 \cdot 64)\rceil$ probes into each bucket (assuming that $n' = 3n/4$), we ensure that our probes are removed before the canaries in all possible $\lceil n/(4 \cdot 64)\rceil$ rounds of eviction. After this, we are now able place our canaries and proceed as before, where now $c_{\max} = n - b$.

2.3 Measuring communication

Now that we know $C_{\mathcal{L}\to\mathcal{M}}$, we can determine if $\mathcal{L}$ and $\mathcal{M}$ are exchanging ICMP or UDP packets. Specifically, we can measure if (and optionally, how many) packets $\mathcal{L}$ sends to $\mathcal{M}$ outside of a TCP connection. Since sending these packets increments $C_{\mathcal{L}\to\mathcal{M}}$, to measure if $\mathcal{L}$ is sending $\mathcal{M}$ packets, we can verify that $C_{\mathcal{L}\to\mathcal{M}}$ has not changed a minute, hour, or day later, to test if any such packets have been sent in the last minute, hour, or day, respectively. (We assume that $C_{\mathcal{L}\to\mathcal{M}}$'s counter is being kept from timing out on $\mathcal{L}$ as explained in Section 2.1.) If we determine that $\mathcal{L}$ is sending such packets to $\mathcal{M}$, we can optionally re-measure $C_{\mathcal{L}\to\mathcal{M}}$ to determine how many of these packets have been sent.

Measuring TCP communication is more difficult, since packets belonging to a TCP connection draw from that connection's per-connection counter, not $C_{\mathcal{L}\to\mathcal{M}}$. However, by spoofing certain TCP packets, we can cause $\mathcal{L}$ to send packets to $\mathcal{M}$ outside of any connection. A naive way we might measure if $\mathcal{L}$ is connected to $\mathcal{M}$ is to, for each port in $\mathcal{M}$'s e-many ephemeral ports, spoof an ACK from $\mathcal{M}$ to $\mathcal{L}$. If $\mathcal{L}$ is not connected to $\mathcal{M}$ on a port, $\mathcal{L}$ will send $\mathcal{M}$ a RST drawing from $C_{\mathcal{L}\to\mathcal{M}}$. Otherwise, $\mathcal{L}$ will send $\mathcal{M}$ an ACK from that connection's counter. If $C_{\mathcal{L}\to\mathcal{M}}$ increases by e, then we know that $\mathcal{L}$ is not connected to $\mathcal{M}$. Otherwise, if it increases by less than e, we might conclude that $\mathcal{L}$ is connected, and even calculate the number of connections. Unfortunately, this is in many cases impractical, as it may be difficult to distinguish between $\mathcal{L}$ having one connection to $\mathcal{M}$ and one of our spoofed ACK packets being dropped.

Although the previous method measures if the server has a connection in the ESTABLISHED state with the client, we discovered another measurement method that measures if the server has a closed TCP connection in the TIME-WAIT state with the client. This measurement also has the desired property that $C_{\mathcal{L}\to\mathcal{M}}$ will only increase if there is a connection. A server reaches the TIME-WAIT when it initiates a connection close, which is common with protocols such as HTTP, where servers close persistent HTTP connections when they go unused for too long. The purpose of this state is to continue to reserve the closed connection's addresses and ports in case any stray packets from that connection trickle in later. On Linux, we found that this state typically lasts for at least 60 seconds.

We measure if $\mathcal{L}$ has a connection in TIME-WAIT with $\mathcal{M}$ by, for each port in $\mathcal{M}$'s e-many ephemeral ports, spoof a SYN from $\mathcal{M}$ to $\mathcal{L}$. If there is no connection, then $\mathcal{L}$ will send a SYNACK to $\mathcal{M}$ with IPID zero (Linux always sends SYNACK's with IPID zero). Otherwise, if there is a connection in TIME-WAIT, $\mathcal{L}$'s behavior will depend on the sequence number of our spoofed

SYN. If the sequence number is newer[5] than the last sequence number received, then $\mathcal{L}$ interprets this as a new connection request and responds with a SYNACK anyways. Otherwise, if with $1/2$ probability we guess a sequence number that is older, $\mathcal{L}$ responds with an ACK with an IPID from and incrementing $C_{\mathcal{L}\rightarrow\mathcal{M}}$.

2.3.1 Network Address Translation

When $\mathcal{M}$ is a NAT router, we are effectively measuring whether $\mathcal{L}$ is communicating not just with that router but also any host behind it. Our canaries and probes enter the router's fragment cache, and so we measure according to the operating system of $\mathcal{M}$, the router, and not any of the hosts behind it.

3 Experimental setup and results

To verify that our technique can measure communication between hosts, we created a network consisting of the following machines:

- Ubuntu 14.04 measurement machine
- Ubuntu 14.04 server running Apache 2
- Windows XP client using Mozilla Firefox
- OS X 10.9 client using Google Chrome
- Linksys NAT router (running Linux 2.4.20), behind which was
  - OS X 10.7 client using Apple Safari

We first inferred the value of the IPID counter between the server and each of the three machines, choosing $r_{\max} = 500$ and $k = 10$ for our experiment. For Windows XP and OS X 10.9, we used the TCP ACK probe, but for the Linksys router, we used the ICMP echo probe as it filtered blind ACK's. For Windows XP, OS X 10.9, and the Linksys router, inferring their counters took 75.2, 46.8, and 69.2 minutes, respectively. We found that the OS X machine took less time because it had a larger fragment cache that could hold more canaries. These times, while they may seem long, are one-time setup costs that can be considered analogous to installing a wiretap.

For each machine, we then began spoofing SYN packets from that machine to the server, cycling through each machine's ephemeral port range every 60 seconds. After an hour, we then measured that none of the server's per-destination IPID counters to these machines had increased.

Finally, we had Windows XP, OS X 10.9, and OS X 10.7 (the latter from behind the router) browse to the default index page that came installed with the server. To simulate browsing behavior, we then refreshed each client's browser every 60 seconds for an additional four page loads. We then searched linearly for their new counter's values starting from their last known value and found that Window XP's, OS X 10.9's, and the router's counters increased by 3, 7, and 4, respectively, successfully measuring their communication with the server.

[5]We say that a sequence number y is newer than x if y occurs in x's upcoming half of the entire sequence space.

4 Related work

The work most closely related to ours is the body of work by Gilad and Herzberg. They demonstrated that IP fragmentation could be used to allow off-path interception and denial-of-service [7], in scenarios that involve NAT or tunneling and where a zombie agent behind the same NAT or tunnel-gateway as the victim or a puppet agent on the victim run attacker code. Part of their attacks involve inferring the Linux per-destination IPID, but their inference techniques rely heavily on the zombie or puppet whereas our technique for inferring the Linux per-destination IPID can be done completely remotely and off-path. Gilad and Herzberg also explored three types of side channels for inferring traffic between hosts and assessed impact on the Tor network [6]. The three side channels considered were a global IPID counter, packet processing delays, and bogus-congestion events. Per-host IPIDs have a much higher signal-to-noise ratio than any of these three side channels, especially when the server is communicating with many other clients—as might be expected in a scenario such as the Tor network. Gilad and Herzberg also demonstrated attacks on the Same Origin Policy by combining puppets with TCP/IP side channels to perform TCP injection [8]. Our attack requires no zombies or puppets.

There has been some work on detecting stepping stones [18, 19], which is a problem that the side-channel technique we present in this paper could perhaps be applied to.

TCP-IP hijacking [13, 2, 11] has a long history, but typically assumes an attacker on the path between the client and server. TCP-IP hijacking also requires that the sequence number be guessed, which we do not attempt to do in this paper. For an off-path attack on TCP, see Qian *et al.* [17]. Our work is not an attack on TCP/IP, but rather an attack on the assumption that IP hosts can communicate with each other without revealing this fact to off-path third parties. Note that our attack can infer ICMP and UDP traffic in a more straightforward way than for TCP. Our attack is very much an attack on the IP protocol itself.

Using global IPID counters for inference is very common. Chen *et al.* [4] use the IPID field to perform advanced inferences about the amount of internal traffic generated by a server, the number of servers in a load-balanced setting, and one-way delays. Bellovin describes a technique for counting NAT'd hosts [3]. Kohno *et al.* use the IPID for remote device fingerprinting [12]. Our work, as far as we know, is the first work to infer a per-destination IPID without assuming any puppet agents or zombies on the victim's network.

5 Discussion and Conclusion

The most obvious way to address the attack we present in this paper would be to fix a vulnerability. It is not clear that there is any specific vulnerability that we have exploited. Information flow that reveals IPID's is apropos to the requirement in RFC 791 [15] that states that IPID's must be unique for every in-flight packet. There will always be non-zero information flow in any shared sequences of numbers that has restrictions on repetition, thus non-interference [9] with respect to off-path traffic inference cannot be achieved by any IP implementation that is RFC-compliant.

That said, we recommend that the Linux per-destination IPID scheme be redesigned. Various flavors of BSD have implemented randomized IPID schemes that leak much less information. Even the long-denounced practice of global IPID counters, such as implemented by Microsoft Windows, leaks less information than the Linux per-destination IPID implementation because it aggregates information from all traffic to all hosts, and is therefore a noisier signal.

To ameliorate this attack, Linux could also maintain IPID counters per-protocol in addition to per-destination. Since RFC 791 [15] specifies that datagrams be combined according to a fragment's id, source, destination, and protocol number, the counters for each protocol could be maintained independently without fear of fragments from different protocols being erroneously combined, even if they have the same IPID. Since our attack only leaks a Linux machine's per-destination IPID's *via* its fragmented ICMP echo replies to other machines, this modification would prevent an attacker from counting UDP packets or inferring TCP connections via our attack. However, this approach would still allow an attacker to count ICMP packets, and future attacks may allow an attacker to count packets specific to additional protocols.

Until the per-destination IPID scheme of Linux is redesigned, implementation and deployment efforts for onion routing (*e.g.*, Tor [5]), Virtual Private Networks (VPN's), and other privacy technologies should consider using TCP instead of UDP. Many servers for privacy technologies are based on Linux. Our attack is much more straightforward and powerful against UDP than it is against TCP, so the combination of UDP and Linux should be considered dangerous.

Linux servers that need to mitigate against the attack presented in this paper until the issue is fixed (*e.g.*, Tor relays) should consider the role of IP fragments in the attack. It may be possible to mitigate the attack without impacting legitimate traffic by placing limitations on IP fragments via firewall rules or kernel runtime parameters.

While our attacks assume that one machine is running Linux and we take advantage of information flows particular to Linux, this does not necessarily mean that servers running other operating systems are less susceptible to off-path attacks that infer their communications with clients. Modern network stacks are highly complex and have many inter-dependent modules and shared, limited resources. Even for Linux we do not yet know the extent of possible information flow leaks.

Effectively, the attack presented in this paper gives malicious actors the following capability (based on some assumptions, of course): given two arbitrary IP addresses anywhere on the Internet, measure the exact number of packets sent from one machine to the other between two points in time. This has implications for privacy, but also for connectivity and censorship circumvention technologies, as well. For example, "Is a remote server communicating with Tor directory authorities?" is a very powerful way to detect bridges irrespective of any protocol obfuscation that occurs between a client and bridge.

To summarize the bigger picture, *a large fraction of research and practice on free and open communications on the Internet is based on the assumption that the specifications and implementations for protocols such as IP, UDP, TCP, and ICMP have non-interference properties with respect to off-path traffic.* Such non-interference properties were never a design goal of any of these protocols, and implementations of them are even less careful about preventing information leaks.

This problem is analogous to the observation by Ptacek and Newsham [16] (and concurrently by Paxson [14]) that network intrusion detection systems (NIDS) can be easily defeated if they are built on the assumption that the NIDS's view of network traffic is the same as the end host's view. Similar to how NIDS systems can never expect to perform perfect traffic normalization, we can expect that the Internet and its integral protocols will never have perfect non-interference properties. However, in keeping with the NIDS analogy, it is important that we research the information flow properties of all of the Internet's integral protocols and their implementations so that we have a better understanding of how to mitigate attacks that threaten fundamental assumptions on which privacy enhancing systems and censorship circumvention technologies are based.

Acknowledgments

We would like to thank the anonymous FOCI reviewers for very helpful feedback. This material is based upon work supported by the National Science Foundation under Grant Nos. #0844880, #0905177, #1017602, and

#1314297. Jed Crandall is also supported by the Defense Advanced Research Projects Agency CRASH program under grant #P-1070-113237.

References

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

[2] BELLOVIN, S. M. Security problems in the tcp/ip protocol suite. *SIGCOMM Comput. Commun. Rev. 19*, 2 (Apr. 1989), 32–48.

[3] BELLOVIN, S. M. A technique for counting NATted hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment* (New York, NY, USA, 2002), IMW '02, ACM, pp. 267–272.

[4] CHEN, W., HUANG, Y., RIBEIRO, B. F., SUH, K., ZHANG, H., DE SOUZA E SILVA, E., KUROSE, J., AND TOWSLEY, D. Exploiting the IPID field to infer network path and end-system characteristics. In *Proceedings of the 6th International Conference on Passive and Active Network Measurement* (Berlin, Heidelberg, 2005), PAM'05, Springer-Verlag, pp. 108–120.

[5] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association.

[6] GILAD, Y., AND HERZBERG, A. Spying in the dark: TCP and Tor traffic analysis. In *Proceedings of the 12th International Conference on Privacy Enhancing Technologies* (Berlin, Heidelberg, 2012), PETS'12, Springer-Verlag, pp. 100–119.

[7] GILAD, Y., AND HERZBERG, A. Fragmentation considered vulnerable. *ACM Trans. Inf. Syst. Secur. 15*, 4 (Apr. 2013), 16:1–16:31.

[8] GILAD, Y., AND HERZBERG, A. Off-path TCP injection attacks. *ACM Trans. Inf. Syst. Secur. 16*, 4 (Apr. 2014), 13:1–13:32.

[9] GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *IEEE Symposium on Security and Privacy* (1982), pp. 11–20.

[10] HOLLIS, K. IPv4 fragmentation –> The Rose Attack. http://seclists.org/bugtraq/2004/Mar/351.

[11] JONCHERAY, L. A simple active attack against TCP. Tech. rep., Merit Network, Inc., FX: (313) 747-3185, April 1995.

[12] KOHNO, T., BROIDO, A., AND CLAFFY, K. C. Remote physical device fingerprinting. *IEEE Trans. Dependable Secur. Comput. 2*, 2 (Apr. 2005), 93–108.

[13] MORRIS, R. T. A weakness in the 4.2BSD Unix TCP/IP software. Computing Science Technical Report No. 117, AT&T Bell Laboratories, Murray Hill, New Jersey, 1985.

[14] PAXSON, V. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium* (Berkeley, CA, USA, 1998), USENIX Association.

[15] POSTEL, J. RFC 791: Internet Protocol, Sept. 1981. Obsoletes RFC0760. See also STD0005. Status: STANDARD.

[16] PTACEK, T. H., AND NEWSHAM, T. N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Tech. rep., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[17] QIAN, Z., AND MAO, Z. M. Off-path TCP sequence number inference attack - how firewall middleboxes reduce security. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 347–361.

[18] STANIFORD-CHEN, S., AND HEBERLEIN, T. L. Holding intruders accountable on the Internet. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995* (1995), pp. 39–49.

[19] ZHANG, Y., AND PAXSON, V. Detecting stepping stones. In *Proceedings of the 9th USENIX Security Symposium* (Berkeley, CA, USA, 2000), USENIX Association.