Geoffrey Alexander*, Antonio M. Espinoza, and Jedidiah R. Crandall

Detecting TCP/IP Connections via IPID Hash Collisions

Abstract: We present a novel attack for detecting the presence of an active TCP connection between a remote Linux server and an arbitrary client machine. The attack takes advantage of side-channels present in the Linux kernel's handling of the values used to populate an IPv4 packet's *IPID* field and applies to kernel versions of 4.0 and higher. We implement and test this attack and evaluate its real world effectiveness and performance when used on active connections to popular web servers. Our evaluation shows that the attack is capable of correctly detecting the *IP-port* 4-tuple representing an active TCP connection in 84% of our mock attacks. We also demonstrate how the attack can be used by the middle onion router in a Tor circuit to test whether a given client is connected to the guard entry node associated with a given circuit.

In addition we discuss the potential issues an attacker would face when attempting to scale it to real world attacks, as well as possible mitigations against the attack. Our attack does not exhaust any global resource, and therefore challenges the notion that there is a direct one-to-one connection between shared, limited resources and non-trivial network side-channels. This means that simply enumerating global shared resources and considering the ways in which they can be exhausted will not suffice for certifying a kernel TCP/IP network stack to be free of privacy risk side-channels.

DOI 10.2478/popets-2019-0071
Received 2019-02-28; revised 2019-06-15; accepted 2019-06-16.

1 Introduction

We 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 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 [27]. 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 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 kernels changes from using one of its 2048 global IPID counters to using a per-connection TCP IPID counter, the attack we 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.

Our major contributions are as follows:

- We 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. Ours is the first such attack to infer the existence of a connection completely off-path without exhausting any global resource.
- We 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.

***Corresponding Author: Geoffrey Alexander:** University of New Mexico, E-mail: alexandg@cs.unm.edu
Antonio M. Espinoza: University of New Mexico, E-mail: amajest@cs.unm.edu
Jedidiah R. Crandall: University of New Mexico, E-mail: crandall@cs.unm.edu

– We 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 paper, compared to past work, is that it does not exhaust any global resource. To the best of our 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 [21] where the global fragment cache was filled and Cao *et al.* [7] where a global challenge ACK rate limit was reached. Note that we exclude attacks that require malicious code on the victim machine or an attacker machine behind the same NAT as the victim [10, 18–20, 29]. Our attack uses a per-destination (*i.e.*, not global) duplicate ACK limit for one non-default corner case that we 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 [4] 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 paper 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 paper is structured as follows: Section 2 discusses scenarios that motivate our work. Section 3 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 discusses the methods for using IPIDs and challenge `ACK`s as side-channels to detect the presence of an active TCP connection. Section 5 describes our experimental methodology. Then, we discuss our results in testing the attack in Section 6. In Section 7 we discuss the applicability of the attack, the challenges it faces "in the wild", common sources of error, and possible mitigations. We discuss related work in Section 8 and finish with our conclusions in Section 9.

2 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 [12] 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 [15], obfuscation protocols such as obfs4 [3] 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 we present. Note also that six out of 10 Tor directory servers are dual stack [31] 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 we 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 we describe in this paper provides the attacker with a primitive for inferring the existence of connections off-path, which violates assumptions often made by privacy tools. We focus our 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 paper assumes that the connection persists for roughly two minutes, and sometimes fails, we 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 results 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 some 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 after some time (*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.

In summary, this paper establishes a powerful attack primitive that is flexible enough to be implemented in a variety of attacker applications.

3 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.

3.1 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 paper we will focus on IPv4 as it is always guaranteed to have an IPID field present.

3.2 Linux Kernel IPv4 IPID Values

Early network stack implementations often used a global IPID counter that was incremented for each packet sent by the machine. However, work on *idle scans* and similar techniques [4, 16, 17] exploited information side-channels in the global IPID field to make measurements of off-path machines. This led to the Linux kernel moving to the adoption of per-destination IPID counters [13]. 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, *etc.*), and a random value generated on system boot.

Research by Knockel and Crandall [21] 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 [13], 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 [14] 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. This discussion was initiated because of Knockel and Crandall's disclosure, but was about side-channels in the 2048 separate counters and not the original vulnerability that Knockel and Crandall disclosed [23]. For Linux's current IPID implementation, every time a packet is sent the chosen counter (among the 2048) is then incremented by a

```
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  }
```

Fig. 1. Linux kernel IPID selection.

random value, chosen from a uniform distribution between 1 and the number of system ticks (typically milliseconds) since the last packet transmission that used the same counter. To summarize, three different side-channels were discussed with respect to the 2014 patch in response to Knockel and Crandall: two pre-patch and one post-patch. Further improvements to the current IPID generation strategy (*e.g.*, a different distribution for added noise) could help mitigate side-channels in this data structure, but would not mitigate the side-channel we present in this paper because we need only detect that the resource is being used at all in our case.

3.3 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 3.1, 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 try and avoid fragmenting packets during transmission. PMTU Discovery changes how the Linux kernel chooses IPIDs when sending TCP packets. When PMTU Discovery is active the Linux kernel does not pick an IPID from one of the 2048 counters, discussed in Section 3.2, 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`s 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 1 shows the IPID selection behavior for Linux kernel version 4.16 [32]. 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 [32]. The attack we describe handles both cases when the Do Not Fragment flag is and is not set.

3.4 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. Next, 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 our tests and code review of the Linux kernel's implementation of RFC 5961, we verified that Linux sends challenge ACKs in response to unsolicited SYN/ACKs, as had been discussed by Cao *et al.* [7]. This is relevant to our attack because it means we do not need to account for sequence numbers in the spoofed SYN/ACKs we send (although doing so would simply be a matter of sending four sets of SYN/ACKs to ensure one set has sequence numbers in the expected window).

4 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 3.2 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 we describe requires the following primitives: First, 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. Second, a method for counting IPID changes caused by off-path traffic. Finally, a method for determining which counter off-path traffic used based on the counted IPID changes. We will discuss our implementations for all three primitives below and describe how they can be combined in the attack we describe.

4.1 Triggering Off-Path Traffic

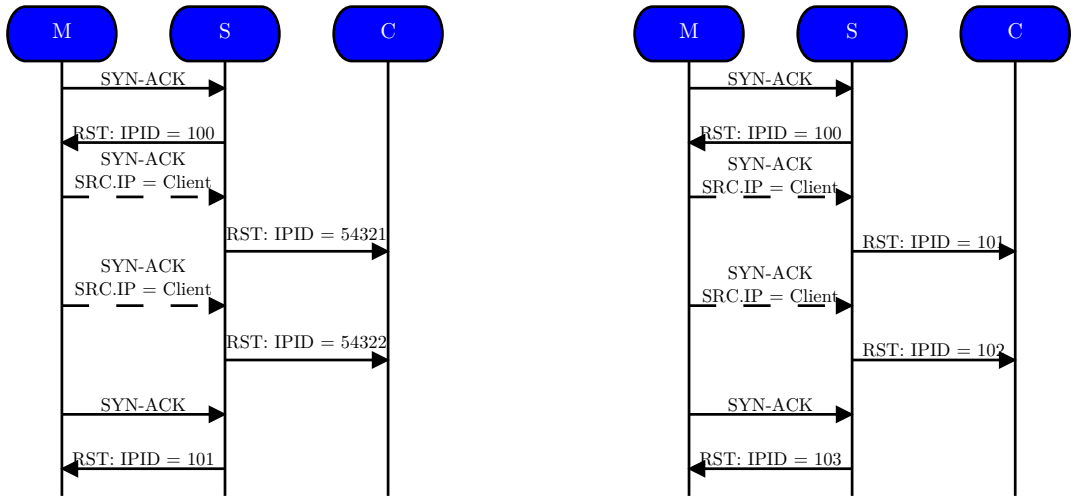
To trigger the desired off-path traffic an attacker would need to send packets to the targeted machine that cause an off-path response, sent over TCP, which will use different counters when a TCP connection is present on a given IP-Port 4-tuple and when a TCP connection is not present. As described in Section 3.4, 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. Our 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 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/ACKs in this way our 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.

4.2 Counting IPIDs

As discussed in Section 3.2 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 our 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.

4.2.1 Finding IPID Collisions

In order to find IPID collisions the attack uses a technique similar to that described by Zhang *et al.* [37], though only applied to IPv4 addresses. As described in



(a) No IPID Collision

(b) IPID Collision

Fig. 2. Example of behavior with and without an IPID hash collision.

Section 3.2 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 we define an IPID collision as: For a server with a secret value S and IPv4 address T we 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 our implementation uses only IPv4 and TCP packets we can treat the destination address and network protocol as constant. We assume that the server we are using as the destination of the attack does not reboot during our 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 we only need to find a IPv4 address that would result in the same hash as our client machine's IPv4 address. Note that while our 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 we did not explore this possibility.

To detect such a collision we can probe the server from a range of IPv4 addresses checking to see if responses to our TCP probes show the presence of off-path traffic using the same global IPID counter. First, we 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, we 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, we 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 same hash then their IPIDs will come from the same IPID counter. We 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 packet as described in Section 3.2 will never be larger than one. If the difference modulo 2^{16} is 1 then a `RST` was not sent to the client machine in between our probes using the counter the attacker's IPv4 address uses. If the difference is 2 then the `RST` sent to the client used the same IPID counter as the probe packets and we know that our 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 we have found such a collision we can then proceed with the attack. For our implementation we chose the simplest method of sequentially scanning for collisions between each possible client and attacker IPv4 address pair.

As an example consider Figure 2a and Figure 2b. In Figure 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 a collision had occurred. In Figure 2b we are using two IP addresses that will cause a collision. In this case we 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/ACKs 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 3.2, avoiding the per-socket counter described in Section 3.3.

4.2.2 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 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:

$$Pr(X = k) = 1 - (1 - p)^k \quad (1)$$

where k is the number of IPv4 addresses used and p is the probability of any single IPv4 address colliding, 1/2048.

Table 1 shows the probability of finding a collision when using common CIDR network sizes.

Number of Addresses (k)	Probability (p)
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

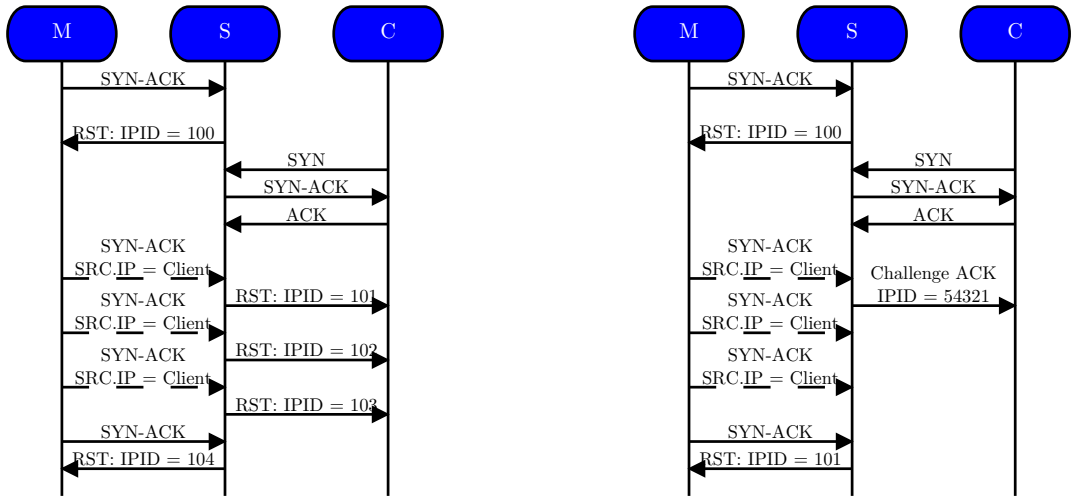
Table 1. Probability of a collision amongst k addresses

4.2.3 Example

As an example consider what happens when we 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 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/ACKs it does not increase the IPID counter used and the difference 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 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 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.



(a) No Connection

(b) Active Connection

Fig. 3. Checking for a connection.

First, the attacker finds an IPv4 address that collides with the client IPv4 address in our 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 would need to 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.

We 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 we send M `SYN/ACK` probes to each 4-tuple in the bucket. We only query the IPID value before and after we send all the probes from a given bucket. Based on the challenge `ACK` behavior described in Section 3 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 we 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 we 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 we can simply send all our $M \cdot N$ 4-tuple probes as fast as possible since we do not have to check for changes after each one. If we detect a bucket that contains a valid 4-tuple we then scan each of the N 4-tuples in the bucket until we find a 4-tuple that represents an active connection. Using this technique to speed up the attack, our 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 our specific implementation we chose to send eight SYN/ACK packets for each individual port. We 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, we 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 we consider the port to be a potential TCP connection. We chose this threshold to allow for a small amount of noise from outside traffic to impact the IPID without greatly impacting our implementation. If the difference is greater than or equal to the number of SYN/ACK probes sent to the potential 4-tuple, minus one, we conclude that the 4-tuple does not represent a valid TCP connection and move to scan the next possible 4-tuple. A difference of 1 indicates that some sort of error or packet loss occurred.

5 Experimental Methodology

In testing the effectiveness of the attack we 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 we opened TCP connections with popular web servers, that met the attack's required criteria, and attempted to use our 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.

Our experiment proceeded as follows:

1. Using the top 250 sites from the Alexa Top Global Sites [1] we 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 we generated all unique pairs of IPv4 addresses from a set of 242 IPv4 addresses on our 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 we had generated all client and attacker IPv4 address pairs we scanned each server IPv4 address that met our 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. We 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 we close the TCP connection and log our results.

We 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.

5.1 Determining TCP Ports to Target

When choosing ephemeral ports to use as client source ports we 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] [11]. 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 try and maximize the number of client machine's whose connections could be attacked while also avoiding attempting to scan all possible ephemeral ports. For our experiments we 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 our measurement network. The measurement machine we 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 our experimental setup is identical to a real attack in terms of Internet traffic considerations.

5.2 Testing Against Persistent Connections

In addition to testing the attack "in the wild" against popular web servers, we also set out to test it against privacy tools such as Tor which create persistent, long lasting connections. Whereas tools such as VPNs that use TCP create connections that persist for as long as the user uses the VPN, Tor creates a new circuit every 10 minutes (MaxCircuitDirtiness defaults to this value). For our mock targeted attack we assume that we are the middle router chosen in a Tor circuit, and whenever a new circuit is created through us we carry out the side-channel attack to see if a given target IP address currently has a connection to the entry guard of the circuit. Part of Tor's security model is the assumption that the middle onion router knows the entry guard and exit node, but cannot gain information about either endpoint of the socket connection tunneled through Tor. This includes the client that connects to the entry guard and the server they are connecting to, for demonstration purposes we focus on the former though the latter can also be inferred.

In order to avoid testing against circuits in the actual Tor network and potentially interfering with legitimate Tor traffic, we tested against Tor clients within a small, virtual network. We also implemented some of the possible improvements discussed in Section 7.2 to see if they improve the attack's accuracy.

This virtual network is constructed as follows:

- Four machines used to create a Tor network.
  - A machine serving as a Tor directory server.
  - A machine serving as a Tor guard node and the entry node of the Tor circuit.
  - A machine serving as an internal Tor relay node and the attacker machine.
  - A machine serving as the Tor exit node.
- One server machine which client machines connect to through a created Tor circuit.
- Two client machines. In each round, a random client connects to the server via Tor, while the second client machine does not.
- One internal IP router node to route traffic between the various machines and simulate Internet-like packet loss, variable delay, and reordering.

Each machine is assigned an IP address on a separate /24 subnet, except our "attacker" machine which is assigned an IP address on a /18 subnet to ensure the attacker can spoof a large enough number of IP addresses to find an IPID hash collision on both client machines. We add a latency of 70-75ms across the entire network to simulate "real world" network delays and packet reordering, as well as a randomized 5% packet loss.

In each round the attacker node will attempt to determine whether or not a connection between a given client machine and the guard node exists, starting when it detects that a circuit has been created and via communication with the guard node itself. The client machine using Tor initiates an SSH connection to the server, through Tor. This SSH connection is closed shortly after but the client maintains the Tor connection to the circuit *via* the entry guard for 10 minutes (default Tor behavior) before attempting to create a new Tor circuit.

Number of 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 2. Overall accuracy of the attack.

6 Results

In this section we describe the results of our experiment and discuss potential factors that could impact the attack's effectiveness. As described in Section 5 we chose web sites from the Alexa Top 250 as the servers to detect connections to. After performing DNS A lookups for each host we obtained 464 unique IPv4 addresses to use as server machines. Of these 464 IPv4 addresses we found 182 (39.22%) that met our required behavior criteria: a machine running a Linux kernel version 4.0 or newer, that responded to "unsolicited" `SYN/ACK` packets with `RST` packets. Of these 182 IPv4 addresses we were able to find valid IPID collisions between a pair of our 242 measurement IPv4 addresses and 136 (74.73%) server addresses. We 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 our own network. Each scan attempted to detect the presence of a TCP connection before and during an active TCP connection.

6.1 Analysis of the Attack's Accuracy

When describing our results we 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 2 shows our overall results.

6.2 Analysis of the Attack's Runtime Performance

In addition to analyzing the accuracy of the attack we 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 6.1 we 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 our 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?).

6.3 Accuracy Against Persistent Connections

As discussed in Section 5.2 we also tested our attack against the persistent connections created by Tor for each circuit. These long lasting (10 minutes by default),

persistent connections provide a longer window of time and allow an attacker to implement strategies such as those discussed in Section 7.2 to improve the attack's accuracy. For our tests we chose to implement the simplest of these strategies, re-checking possible connection 4-tuples multiple times to verify potential positive results. We chose to check each potentially valid 4-tuple 8 times in total and accepted it as a 4-tuple representing a connection if over half these tests returned a positive result (*i.e* 5 or more of the 8 checks returned the same 4-tuple). Each of these re-checks does not scan the full ephemeral port range but instead only scans those ports for which a potential connection was detected. Since these re-checks only check a handful of ports the added time taken is typically quite small and can allow an attacker to try and verify if a persistent connection exists or not.

We find that re-checking potential, positive 4-tuples combined with the increased connection life of long lasting, persistent Tor connections allows an attacker to improve the attack's accuracy in a simulated network. For cases where a connection between a client machine and the Tor entry node exists our tests show the attack has an accuracy of 96% with a false negative rate of 4% and no false positives. In cases where there is no connection our simulated testing has a true negative rate of 100% and no false positives. False negatives could be tolerated in a real attack of this kind because there is some chance that any given circuit created by the victim will not go through an attacker-controlled relay in the first place, so the attack is already probabilistic.

Tor and VPNs are not the only contexts where TCP connections may persist for a significant amount of time. An increasingly common practice among web clients and servers is to use TCP or HTTP keep-alives to maintain persistence of idle connections, to avoid the overhead of starting new TCP connections for subsequent requests. As one example, the Firefox browser maintains idle connections for up to 115 seconds after completing an HTTP request. These persistent connections increase the number of situations where our attack could be used. Our attack is general to TCP and can be applied beyond plain web browsing traffic. Some applications such as IRC create connections that persist for days. Our testing shows that these persistent connections provide an attacker with sufficient time to implement strategies to better handle sources of error and improve the attack's accuracy.

7 Discussion

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

7.1 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 [24] and the Great Firewall of China's attempts to actively probe Tor bridges [15] 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 we describe in this paper, 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.

We assume that the attacker has some 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 [5] 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.

7.2 Sources of Error

During the course of our experiments we 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. One method for accounting for this would be to try to account for the added noise in the IPID counter by modeling the noise. Ensafi *et al.* [15, 16] used ARMA modeling to account for noise affecting the global IPID counters. Pearce *et al.* [26] accounted for noisy IPID counters by using Sequential Hypothesis Testing, while ONIS [37] modeled noise and accounted for it by using the Akaike information criterion. The attack we have described does not attempt to handle detected cases of noise; instead we leave this possibility for future work.

The second cause of errors we noticed is that some IPv4 addresses we used as servers would change the IPID counter being used after successfully finding an IPID hash collision. Since we cached all located IPID collisions as described in Section 4 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. We 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 3 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 some type of load balancing or 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 some type of load balancing technique to improve overall performance and an attacker would need to detect whenever this case has occurred and appropriately handle when it has. We leave the development of such a technique to future work, and note that prior TCP/IP side channel attacks (*e.g.*, Cao *et al.* [7] and Knockel and Crandall [21]) are also affected by layer 4 load balancing, making it an interesting open problem in general.

The third source of error we noticed was the largest cause of false positives in our 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. At two points during the attack we 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. Our 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 somewhere 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 our low false positive rate of less than 5% this does not occur frequently and could be mitigated by rerunning the attack multiple times. We leave the development of techniques to detect and better handle packet reordering to future work.

7.3 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 [21]. 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 an IPID of zero 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 an IPID of zero for such packets would make it impossible for the attack we 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 we 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 not be desirable.

7.4 Ethical Considerations

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

7.4.1 Disclosure

We 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 we 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 we have described. However, the underlying side channel still remains for other protocols and could be used to attack user privacy in applications that are, *e.g.*, UDP-based (such as DNS).

8 Related Works

One of the first uses of network side-channels as a measurement technique was the original Idle Scan proposed by Antirez [4] to scan for open ports on a target machine. Morbitzer [25] proposed a technique for an IPv6 Idle Scan. Ensafi *et al.* [17] proposed another side-channel using the `SYN`-backlog for the same purpose. Subsequent work by Ensafi *at al.* [16] combined the two side-channels to provide a technique for measuring intentional blocking of ports by firewalls. Alexander and Crandall [2] use the `SYN`-backlog side-channel for off-path network measurements of packet round-trip time while Zhang *et al.* [35] improved on their technique. Zhang *et al.* also used side-channels to detect machines hidden behind firewalls [36] and describe ONIS [37] for port scanning. Chen *et al.* [9] 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 [6] and Kohno *et al.* [22] made use of IPIDs to fingerprint remote devices. None of these works can be used to infer the existence of a TCP/IP connection off-path.

Detecting the presence of an active TCP/IP connection is often the first step in TCP reset attacks or connection hijacking, such as those described by Watson [34]. Knockel and Crandall [21] discuss a technique for inferring the presence of active IPv4 connections on Linux machines using the IPID field and IP fragmentation reassembly behavior. Qian *et al.* [28, 29] use firewall based network side-channels to infer TCP sequence numbers and perform off-path TCP/IP connection hijacking. Cao *et al.* [7, 8] 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.* [30] later scanned the Internet to discover how quickly popular web servers patched the vulnerability discovered by Cao *et al.* Chen and Qian [10] describe a side-channel present in the IEEE 802.11 protocol that allows for off-path TCP injection attacks. Of all of these side-channel attacks, only two (Knockel and Crandall [21] and Cao *et al.* [7, 8]) do not require non-privileged malicious code running on the victim client. As discussed previously, both of these works exhaust a global resource.

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 [18]. 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 [20]. 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 we 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 agents and TCP/IP side-channels to attack the Same Origin Policy and carry out TCP injections [19]. All of the work by Gilad and Herzberg requires the use of an agent or attacker machine behind the same NAT as the victim.

To the best of our knowledge, our work is the first purely off-path TCP/IP side channel attack that can infer the existence of a connection without exhausting a global resource. This informs efforts to protect kernel network stacks against unknown side-channels in two ways. First, enumerating shared, global resources (such as rate limits, buffers, and caches) and then evaluating each in isolation will not reveal all TCP/IP side-channels. Rather, efforts to reason about potential undiscovered side-channels for the existence of a connection should consider all code paths that differ when a connection exists. And second, the fact that Linux's IPID behavior for unsolicited packets had to be changed in response to the vulnerability presented in this paper is significant in that this route was chosen because strategies to randomize resource usage (such as the randomized global challenge ACK rate limit in response to Cao *et al.*, or the uniformly distributed noise added to IPIDs combined with the removal of per-destination IPID counters in response to Knockel and Crandall) are not effective for attacks that do not exhaust any global resource. The Linux kernel's response to our vulnerability disclosure, which was to set IPIDs to zero for RST packets, amounted to not using the resource in question (IPID counters) at all. What if a TCP/IP side-channel is discovered that, like the one presented in this paper, does not exhaust any global resource, but where simply not using a resource is not an option as it was in this case?

9 Conclusion

We 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.

We have provided an evaluation of the attack "in the wild" and discussed its effectiveness and performance. We 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 we have discussed the potential applicability of the attack, sources of error that would affect its applicability, and possible mitigations against the attack. Finally, we have discussed what this attack can tell us about yet undiscovered TCP/IP side channels that past attacks could not.

Acknowledgements

The authors would like to thank our anonymous reviewers for their feedback. We would also like to thank Jeffrey Knockel for his feedback and discussions on IPID hash collisions.

This material is based upon work supported by the U.S. National Science Foundation under Grant Nos. #1518878, #1518523, and #1801613.

References

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

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

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

[23] Personal communication. security@kernel.org.

[24] 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.

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

[26] 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.

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

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

[29] 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.

[30] 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.

[31] The Tor Project. Tor Metrics.

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

[33] L. Torvalds. Linux Kernel V4.16. https://github.com/torvalds/linux/blob/v4.16/net/ipv4/tcp_input.c#L3388, March 2018.

[34] P. Watson. Slipping in the Window: TCP Reset attacks. *Presentation at*, 2004.

[35] 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.

[36] 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.

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

```
3394  static void tcp_send_challenge_ack(...) {

          ...

3404      if (__tcp_oow_rate_limited(net,
               LINUX_MIB_TCPACKSKIPPEDCHALLENGE,
               &tp->last_oow_ack_time))
3405          return;

              ...

3419          count = READ_ONCE(challenge_count);
3420          if (count > 0) {
3421              WRITE_ONCE(challenge_count, count - 1);
3422              NET_INC_STATS(net,
                       LINUX_MIB_TCPCHALLENGEACK);
3423              tcp_send_ack(sk);
3424          }
3425  }
```

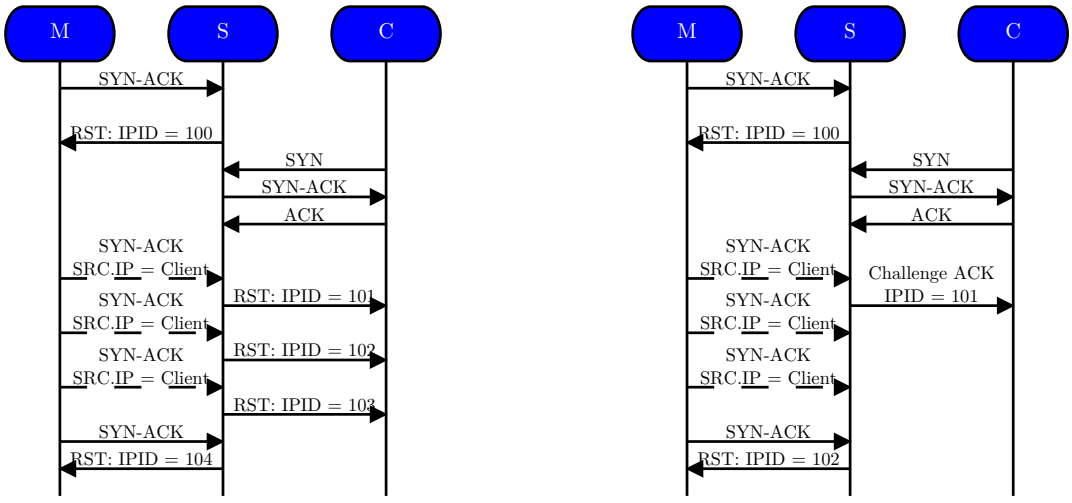
Fig. 4. Linux kernel challenge ACK behavior

A Finding an IPID collision

One of the first steps in the attack is to find an IPv4 addresses whose IPID counter hash collides with the client's IPID counter hash on the server. Our implementation uses a technique similar to that used by ONIS [37]. This is the most time intensive portion of the attack. In cases where a collision was found our implementation took on average 1,669.65 seconds, *i.e.* 27 minutes and 49 seconds, to find the collision. The fastest that our implementation found a collision was 1.73 seconds and the longest time to find a collision was 8,453.34 seconds, *i.e.* 2 hours, 20 minutes, and 53 seconds. Since each attempt to find a collision requires one round trip time per attacker IPID probe these results are heavily influenced by the round trip time between the attacker and the server. As a result collisions with shorter round trip times between the attacker and the server required less time than those with longer round trip times. As discussed previously our implementation scans for collisions and caches any that are found *before* checking for TCP connections and reuses these cached collisions to avoid having to rediscover collisions every time the attack is run. In addition our implementation uses a naive, sequential scan of all possible IP pairs that could result in a hash collision. A more advanced, parallel scan or other less naive implementations could speed up the process of finding collisions and decrease the time needed.

B The attack when the Do Not Fragment Flag is not set

Starting with version 4.0, the Linux kernel began rate limiting duplicate ACK packets to reduce resource consumption in cases where many duplicate ACK packets could be caused by a remote host. The attack takes advantage of this rate limit when applied to challenge ACKs when the Do Not Fragment flag for an outgoing IPv4 packet is not set. When the Linux kernel needs to send a challenge ACK packet it first checks to see if it was already reached its duplicate ACK rate limit as seen in line 3404 of Figure 4. The default value for this rate limit is one duplicate challenge ACK packet sent every one half



(a) No connection

(b) Active connection without DF flag set

Fig. 5. Checking for a connection without the DF flag set

of a second. If a given challenge ACK causes the kernel to exceed this rate limit, the function simply returns and no challenge ACK is sent. If the packet does not exceed the rate limit, the kernel instead checks the global challenge ACK rate limit in line 3420. If this limit has not been reached, the challenge ACK is sent, line 3423, otherwise no challenge ACK is sent. A full source code listing for the Linux kernel's challenge ACK implementation can be seen at [33].

Each spoofed SYN/ACK causes the server to send a RST which increments the IPID counter by one as shown in Figure 5a. If the 4-tuple does represent an active TCP connection and the server is not using the Do Not Fragment flag on outgoing packets the server will send a challenge ACK for the first SYN/ACK but not for subsequent packets. This is because the first challenge ACK will trigger the duplicate ACK rate limit causing each subsequent challenge ACK to be rate limited and not sent. The challenge ACK that is sent will use one of the 2048 IPID counters based on its connection information hash and not the per-connection IPID counter, as described in Section 3.2. Due to this the IPID of this counter increases by one causing the difference between the first and second RST packets sent in response to the attacker's probe packets to be two. This case is shown in Figure 5b.

B.1 Using the Duplicate ACK Rate Limit as a Side-Channel

When an "unsolicited" SYN/ACK is received by the Linux kernel on a port where no active connection exists the kernel will send a RST packet. For ports where an active connection does exist the kernel instead sends a challenge ACK, assuming that the given connection has not hit its duplicate ACK rate limit. If a connection has hit its duplicate ACK rate limit then no packet is sent in response to the SYN/ACK.

If we send multiple SYN/ACKs to a given connection 4-tuple that does not represent an active TCP connection each SYN/ACK will cause the kernel to respond with a RST. However, if a given 4-tuple does represent an active TCP connection then each SYN/ACK will cause the kernel to respond with a challenge ACK. Assuming all SYN/ACK packets arrive in under one half of a second the first challenge ACK response will cause the connection to reach its duplicate ACK rate limit which causes the kernel to not send all subsequent challenge ACK packets. This difference in the number of packets sent in each of the cases can be used as a side-channel to detect if a given 4-tuple represents an active TCP connection or not.