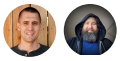
Userspace isn't slow, some kernel interfaces are!

Jordan Whited and James Tucker on December 13, 2022

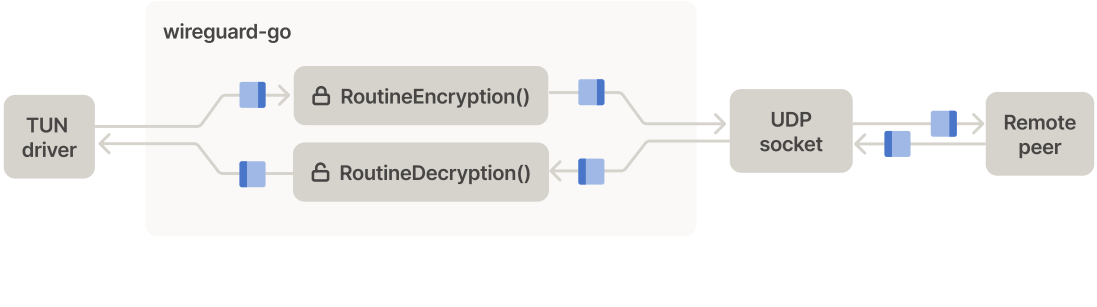


We made significant improvements to the throughput of wireguard-go, which is the userspace WireGuard® implementation that Tailscale uses. What this means for you: improved performance of the Tailscale client on Linux. We intend to upstream these changes to WireGuard as well.

You can experience these improvements in the current unstable Tailscale client release, and also in Tailscale v1.36, available in early 2023. Read on to learn how we did it, or jump down to the Results section if you just want numbers.

Background

The Tailscale client leverages wireguard-go, a userspace WireGuard implementation written in Go, for dataplane functionality. In Tailscale, wireguard-go receives unencrypted packets from the kernel, encrypts them, and sends them over a UDP socket to another WireGuard peer. The inverse flow is flipped — when receiving communications from a peer, wireguard-go first reads encrypted packets from a UDP socket, then decrypts them, and writes them back to the kernel. This is a simplified view of the pipeline inside of wireguard-go — the Tailscale client adds additional functionality, such as NAT traversal, access control, and key distribution.



Baseline

Network performance is a complicated topic in large part because networked applications can have drastically different requirements and goals. In this post, we will focus on throughput. By throughput, we mean the amount of data that can be transferred between Tailscale clients within a given timeframe.

Disclaimer about benchmarks: This post contains benchmarks! These benchmarks are reproducible at the time of writing, and we provide details about the environments we ran them in. Benchmark results tend to vary across environments, and they also tend to go stale as time progresses. Your mileage may vary.

We'll start with some baseline numbers for wireguard-go and in-kernel WireGuard. Toward the end we will show results of our changes. Throughput tests are conducted using iperf3 over a single TCP stream, with cubic-flavored congestion control. Ubuntu 22.04 is the operating system on all hosts.

For these baseline tests, we'll use two c6i.8xlarge virtual hosts in AWS. These instances have fast network interfaces and sufficient CPU capacity to handle encryption at network speeds. They are in the same region and availability zone:

```
ubuntu@thru6:~$ ec2metadata | grep -E 'instance-type:|availability-zone:'
availability-zone: us-west-2d
instance-type: c6i.8xlarge

ubuntu@thru7:~$ ec2metadata | grep -E 'instance-type:|availability-zone:'
availability-zone: us-west-2d
instance-type: c6i.8xlarge

ubuntu@thru6:~$ ping 172.31.56.191 -c 5 -q
PING 172.31.56.191 (172.31.56.191) 56(84) bytes of data.

--- 172.31.56.191 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4099ms
rtt min/avg/max/mdev = 0.098/0.119/0.150/0.017 ms
```

This first benchmark does not use wireguard-go. It sets a throughput baseline without any WireGuard overhead:

```
ubuntu@thru6:~$ iperf3 -i 0 -c 172.31.56.191 -t 10 -C cubic -V
iperf 3.9
Linux thru6 5.15.0-1026-aws #30-Ubuntu SMP Wed Nov 23 14:15:21 UTC 2022 x86_64
Control connection MSS 8949
Time: Thu, 08 Dec 2022 19:29:39 GMT
Connecting to host 172.31.56.191, port 5201
      Cookie: dcnjnuzjeobo4dne6djnj3waeq4dugc2fh7a
      TCP MSS: 8949 (default)
[  5] local 172.31.51.101 port 40158 connected to 172.31.56.191 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-10.00  sec  11.1 GBytes  9.53 Gbits/sec    0   1.29 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  11.1 GBytes  9.53 Gbits/sec    0             sender
[  5]   0.00-10.04  sec  11.1 GBytes  9.49 Gbits/sec                  receiver
CPU Utilization: local/sender 10.0% (0.2%u/9.7%s), remote/receiver 4.4% (0.3%u/4
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
```

This second benchmark uses in-kernel WireGuard:

```
ubuntu@thru6:~$ iperf3 -i 0 -c thru7-wg -t 10 -C cubic -V
iperf 3.9
```

```
Linux thru6 5.15.0-1026-aws #30-Ubuntu SMP Wed Nov 23 14:15:21 UTC 2022 x86_64
Control connection MSS 1368
Time: Thu, 08 Dec 2022 19:58:24 GMT
Connecting to host thru7-wg, port 5201
      Cookie: o5iu6xoxq47swoubx5un32monokel573kj6i
      TCP MSS: 1368 (default)
[  5] local 10.9.9.6 port 46284 connected to 10.9.9.7 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-10.00  sec  3.09 GBytes  2.66 Gbits/sec   81    987 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  3.09 GBytes  2.66 Gbits/sec   81             sender
[  5]   0.00-10.05  sec  3.09 GBytes  2.64 Gbits/sec                  receiver
CPU Utilization: local/sender 5.2% (0.2%u/5.0%s), remote/receiver 5.9% (0.1%u/5.
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
```

Now, over wireguard-go@bb719d3:

```
ubuntu@thru6:~$ iperf3 -i 0 -c thru7-wg -t 10 -C cubic -V
iperf 3.9
Linux thru6 5.15.0-1026-aws #30-Ubuntu SMP Wed Nov 23 14:15:21 UTC 2022 x86_64
Control connection MSS 1368
Time: Thu, 08 Dec 2022 19:30:49 GMT
Connecting to host thru7-wg, port 5201
      Cookie: zg7hsb2jrbklpaqez2gzhdi2kyxr4ibne5lf
      TCP MSS: 1368 (default)
[  5] local 10.9.9.6 port 51660 connected to 10.9.9.7 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-10.00  sec  2.82 GBytes  2.42 Gbits/sec  4711    415 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  2.82 GBytes  2.42 Gbits/sec  4711             sender
[  5]   0.00-10.04  sec  2.82 GBytes  2.41 Gbits/sec                  receiver
CPU Utilization: local/sender 5.7% (0.2%u/5.5%s), remote/receiver 7.3% (0.6%u/6.
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
```

One thing that's interesting to note: The TCP MSS is much higher on the first test (more on MSS/MTU later if you are unfamiliar). AWS supports a 9001 byte IP MTU. What happens when we increase the MTU on the wireguard-go interface?

```
ubuntu@thru6:~$ iperf3 -i 0 -c thru7-wg -t 10 -C cubic -V
iperf 3.9
Linux thru6 5.15.0-1026-aws #30-Ubuntu SMP Wed Nov 23 14:15:21 UTC 2022 x86_64
Control connection MSS 8869
Time: Thu, 08 Dec 2022 19:33:21 GMT
Connecting to host thru7-wg, port 5201
      Cookie: ov4nsnsdfxomict4cu2cxy2iwt4ncpi364d4
      TCP MSS: 8869 (default)
[  5] local 10.9.9.6 port 43416 connected to 10.9.9.7 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-10.00  sec  9.17 GBytes  7.88 Gbits/sec  1854   1.54 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
```

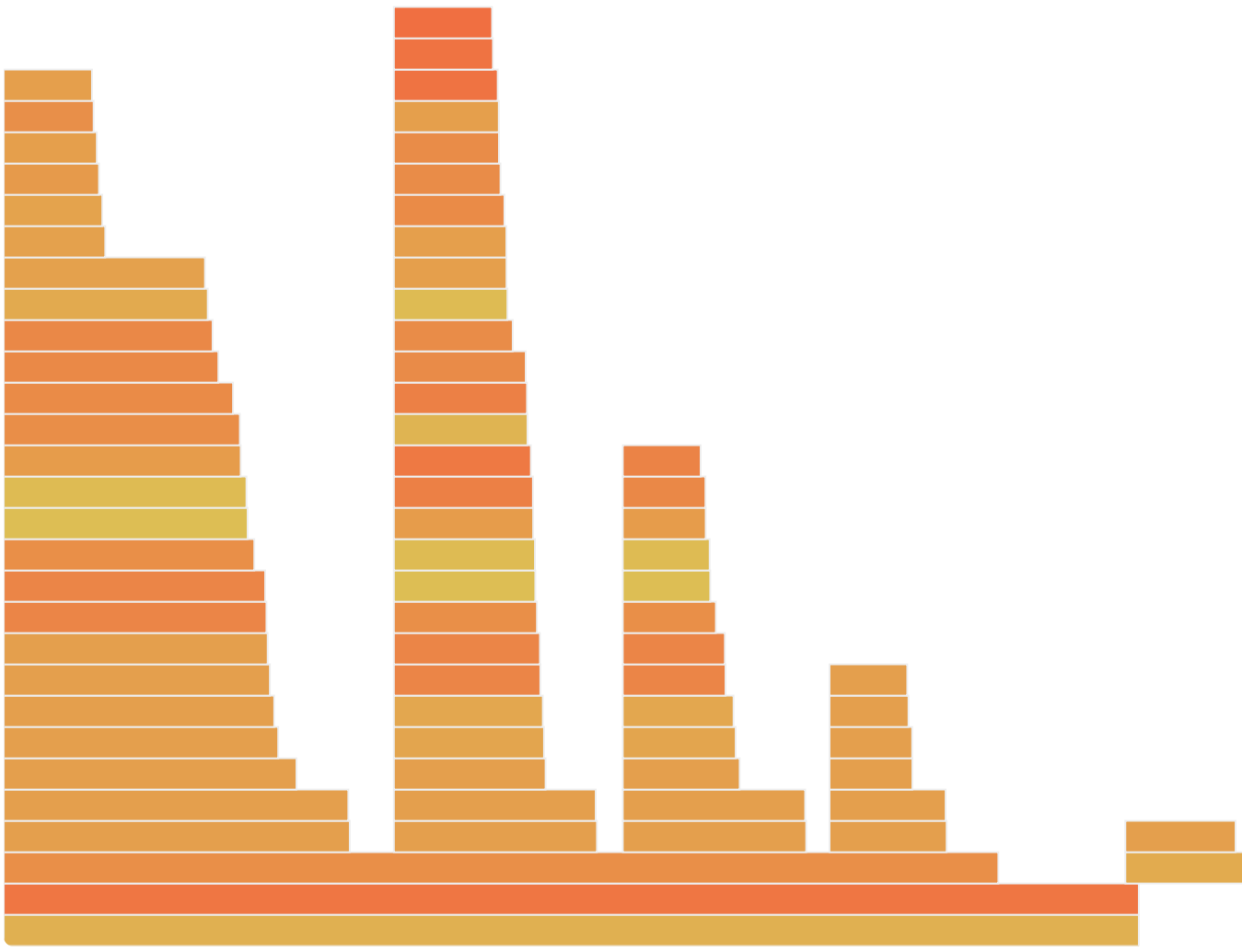
```
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  9.17 GBytes  7.88 Gbits/sec  1854             sender
[  5]   0.00-10.04  sec  9.17 GBytes  7.84 Gbits/sec                  receiver
CPU Utilization: local/sender 14.6% (0.2%u/14.4%s), remote/receiver 26.6% (1.4%u
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
```

Interesting! More than a 3x improvement in throughput. This suggests that per-packet overhead is quite high. Let's grab some Linux perf data to confirm.

Linux perf and flame graphs

We can analyze performance using Linux perf to better understand where CPU time is spent. Flame graphs can be rendered from the perf data, and they help us visualize the stack traces. The wider the function, the more expensive it (and/or its children) are. The flame graphs below are interactive. You can click to zoom and hover to see percentages. The first flame graph is from the sender:

Sender



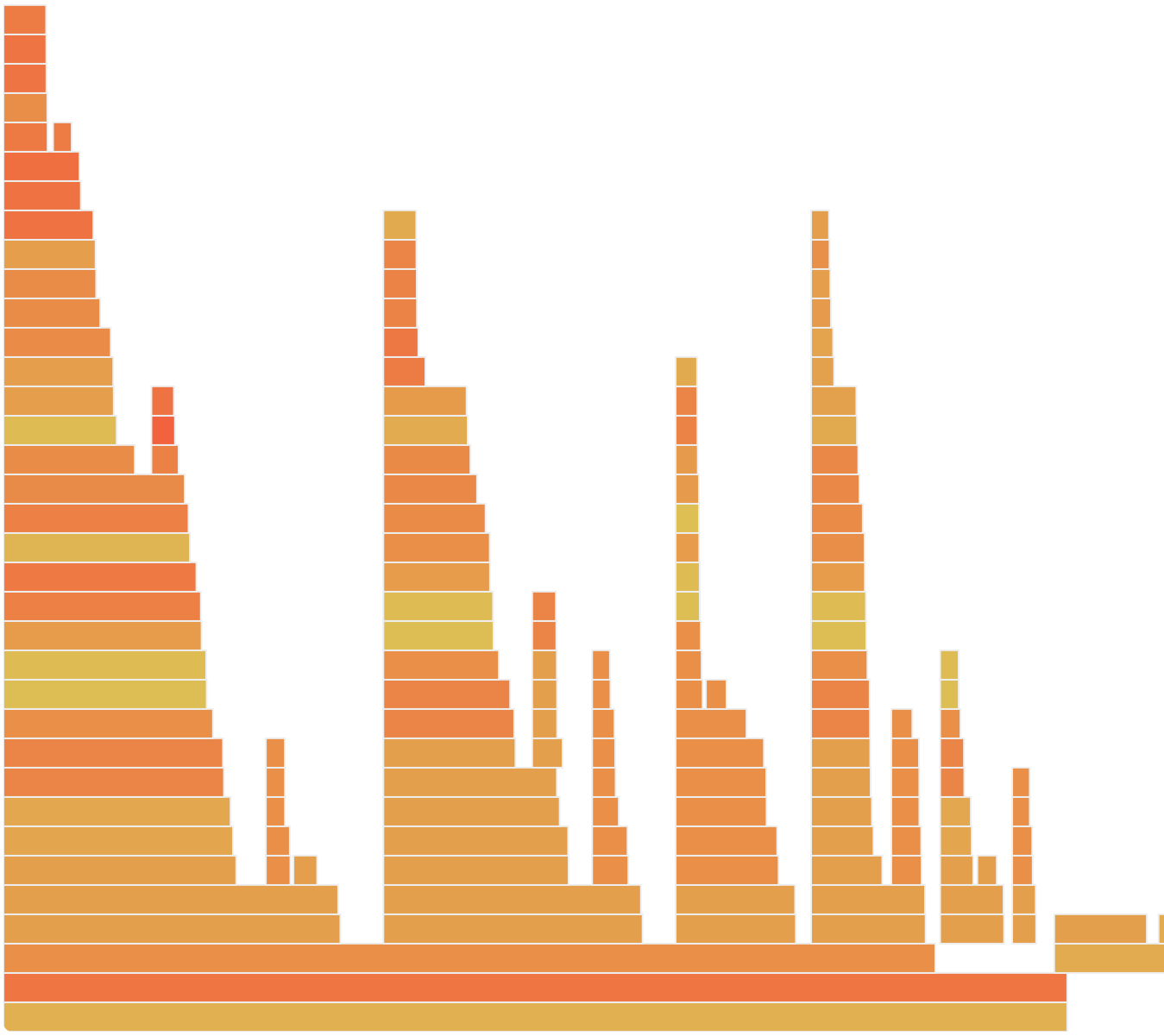
Hover rows for details.

A large portion of CPU time (unrelated to crypto) on the sender is spent in:

1. `sendmsg()` on the UDP socket
2. `write()` towards the TUN driver
3. `read()` from the TUN driver

Now, for the receiver:

Receiver



Hover rows for details.

A large portion of CPU time (unrelated to crypto) on the receiver is spent in:

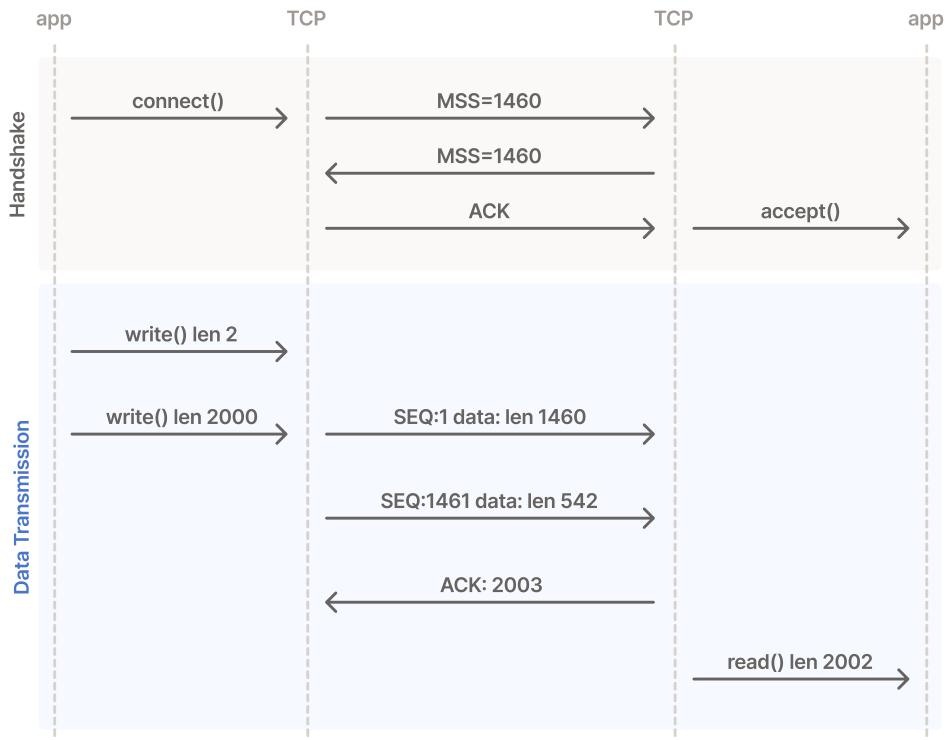
1. `write()` towards the TUN driver
2. `recvmsg()` on the UDP socket
3. `sendmsg()` on the UDP socket

This confirms that per-packet overhead is high. By increasing the MTU of the TUN interface, we reduced the frequency of the system calls for I/O on the TUN and UDP socket. Since those dominate these profiles, it makes sense that throughput would increase as a result. So how do we reduce the frequency of these system calls while still maintaining an MTU that will work across the general internet?

TCP Segmentation

TCP enables transmission of an arbitrary stream of bytes between two sockets. A userspace application interacts with a TCP socket using `write()`- and `read()`-like kernel interfaces once it is in a connected state. The application may `write()` 2 bytes, followed later by a 2,000 byte `write()`. The applications on either end do not need to be involved in the retransmission, ordering, or framing of messages between them. These are all handled by the TCP implementation on the kernel side of the system calls.

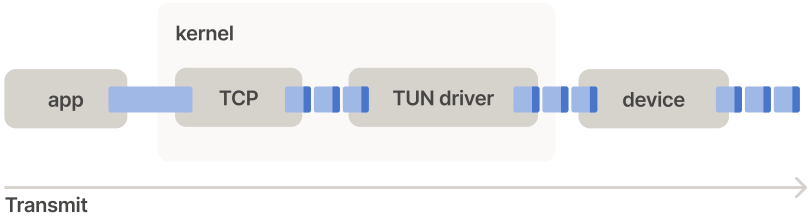
While the application writes arbitrary-sized data per system call, the TCP implementation must segment the data before it gets sent over the network. There is a finite limit to the size of a TCP segment, called the maximum segment size (MSS). MSS is advertised during the TCP three-way handshake. The lowest value wins, and neither side will transmit a segment exceeding it. MSS is typically derived from the maximum transmission unit (MTU) of the egress network interface. The MTU represents the maximum size of a single packet at the network layer. MSS describes the segment size limit at a higher layer protocol (TCP) than MTU (typically IP), and should always be less than it as a result.



There are consequences when a packet exceeds the MTU of a network path. The network device enforcing the limit may choose to fragment the packet into multiple smaller packets, or just drop the oversized packet. Both of these outcomes have negative impacts on performance. There are also mechanisms for signaling to the endpoints that their frames are too large, which we will not go into here. In summary,

there is a finite packet size for packet-switched networks, and TCP implementations try to respect it.

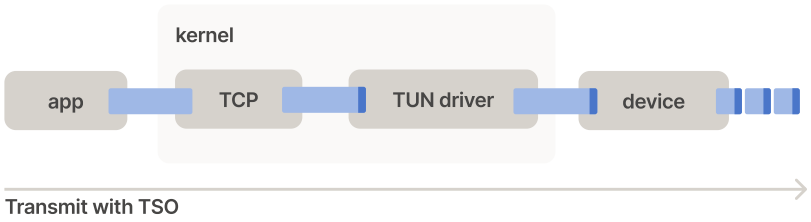
Knowing that TCP is responsible for segmenting the data, we can picture the journey of packets through the host to look roughly something like this:



There is a nonzero cost for each of these layers to handle an individual TCP segment. If we can reduce the number of traversals through this stack, we can win back a lot of CPU time and pipeline latency. Enter TCP segmentation offload (TSO).

TCP Segmentation Offload (TSO)

TSO is an offloading technique where the final fit-within-MSS segmentation is performed by the network interface device. This enables up to 64KB sized segments to traverse the stack, while still fitting within MSS before entering the network. Recent work in the Linux kernel extends the 64KB limit to 256KB. TSO requires extra metadata to be passed along with the oversized segment describing the size to segment to, along with where to find the TCP checksum field. The TCP checksum must be recomputed post-segmentation as the payload is shortened, and various TCP header fields will differ between segments.

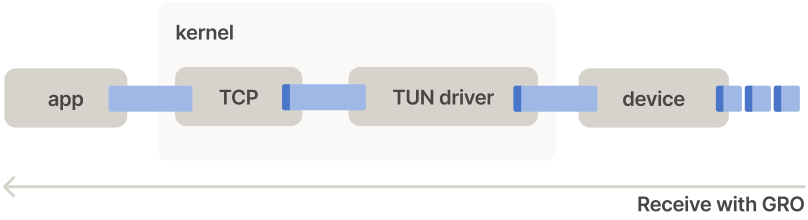


With a typical MTU of 1500 bytes and IPv4 TCP MSS of 1460 bytes, TSO could be used to reduce stack traversals by up to 44x.

Note: Modern Linux makes use of generic segmentation offload (GSO). This enables the networking stack to postpone segmentation as late as possible, even if the driver and device do not support TSO.

But, what about the other direction?

Generic receive offload (GRO) is the inverse of TSO. The network interface device coalesces packets together that belong to the same TCP stream, following a set of rules to prevent breaking the TCP implementation.



(Discovery of) TSO and GRO in the TUN driver

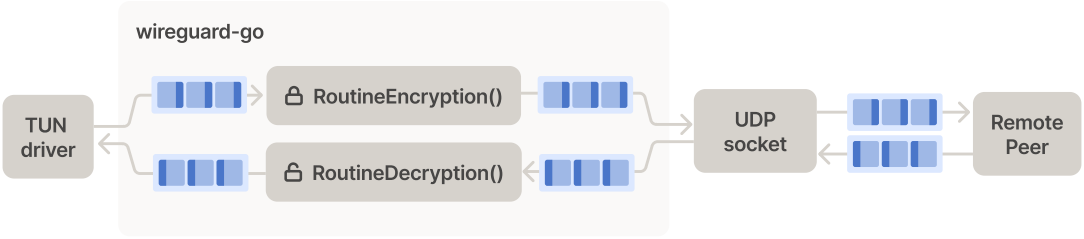
The Linux kernel contains a network device driver referred to as TUN/TAP. This driver is used in wireguard-go in order to present the application as a network device to the kernel. That is, a packet sent out of the TUN interface is received by the userspace application, and the userspace application can inject packets back toward the kernel by writing in the other direction (received by the TUN interface).

When we set out to improve performance, we started by reading the TUN driver code. We had initially hoped to use a multi-message API via a packet socket, but unfortunately the kernel does not expose this to userspace. Instead, we started to explore the set_offload() function. This function is responsible for controlling the offloads supported by the TUN device, and we could enable TSO/GRO through it via `ioctl()`. This functionality has been in the Linux kernel since v2.6.27 (2008), but seems to have gone largely unnoticed outside of the kernel-side virtio framework uses that it was originally added for.

With TSO/GRO enabled on the TUN, the application acting as the TUN device becomes responsible for implementing the offload techniques (segmentation and coalescing). Once we've segmented, where do we transmit the smaller segments? What represents the "network" sitting on the other side of wireguard-go and tailscaled? The answer is "it depends" in the Tailscale client, but typically it's a UDP socket. After receiving packets from the TUN device, wireguard-go handles encryption prior to transmission out of said UDP socket. The same is true in reverse: We receive packets from a UDP socket, they are decrypted, and then written back toward the TUN.

`sendmmsg()` and `recvmmsg()`

The `sendmmsg()` and `recvmmsg()` system calls enable the transmission and reception of multiple messages in a single system call. With a vector of packets now available from reads at the TUN driver, we can leverage `sendmmsg()` when transmitting out of the UDP socket. The inverse direction starts with `recvmmsg()` at the UDP socket, potentially returning multiple packets, which are candidates for coalescing just before writing to the TUN driver. Putting this together with TSO and GRO, we are able to reduce I/O system calls on both ends of the pipeline.



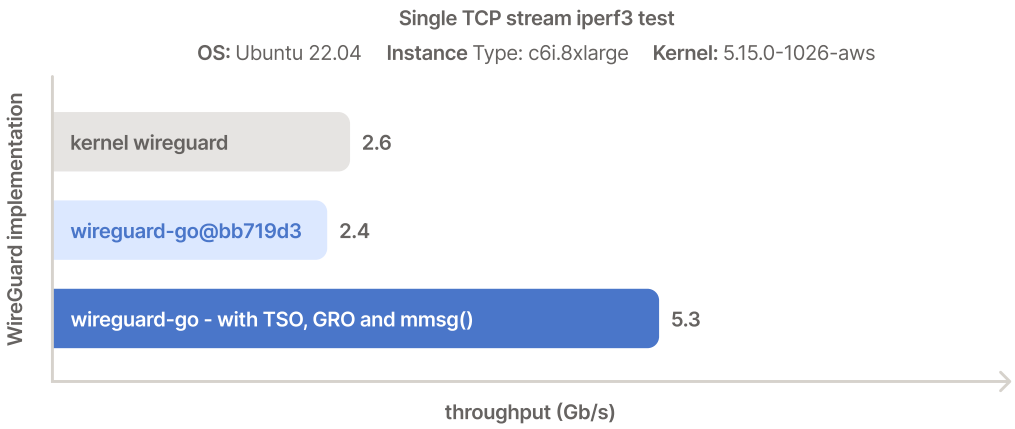
Results

And now for the results!

Applying TCP segmentation offload, generic receive offload, and the mmsg() system calls resulted in significant throughput performance improvements in wireguard-go, and so also in the Tailscale client. Using the same tests we conducted previously, we delivered a best case 2.2x improvement to wireguard-go. And, we improved the throughput of Tailscale on Linux by up to 33%. We intend to continue working on improving the performance of Tailscale in all areas, including throughput, as well as across more platforms.

wireguard-go with TSO, GRO, and mmsg():

```
ubuntu@thru6:~$ iperf3 -i 0 -c thru7-wg -t 10 -C cubic -V
iperf 3.9
Linux thru6 5.15.0-1026-aws #30-Ubuntu SMP Wed Nov 23 14:15:21 UTC 2022 x86_64
Control connection MSS 1368
Time: Thu, 08 Dec 2022 19:39:43 GMT
Connecting to host thru7-wg, port 5201
      Cookie: y4x75uvr2uupa3urdguks6m5ovn2ucdrjxrs
      TCP MSS: 1368 (default)
[  5] local 10.9.9.6 port 58314 connected to 10.9.9.7 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-10.00  sec  6.24 GBytes  5.36 Gbits/sec    0   3.02 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  6.24 GBytes  5.36 Gbits/sec    0             sender
[  5]   0.00-10.04  sec  6.24 GBytes  5.33 Gbits/sec                  receiver
CPU Utilization: local/sender 9.1% (0.1%u/9.0%s), remote/receiver 0.5% (0.0%u/0.
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
```



Surprisingly, we improved the performance of wireguard-go (running in userspace) enough to make it faster than WireGuard (running in the kernel) in the best conditions. But, this point of comparison likely won't be long-lived: we expect the kernel can do similar things.

Conclusions

In our journey to overcome our biggest overhead in packet processing, we came very close to wanting a new or different kernel interface. We gladly found that one was already available in the Linux kernel — and one that has been around long enough for us to use everywhere. Performance can always become somewhat of an arms race, but our results here demonstrate that we can keep up with our kernel counterparts provided that we are using the right kind of kernel interface – userspace isn't slow, some kernel interfaces are!

Thanks to Adrian Dewhurst for his detailed review and thanks to Jason A. Donenfeld for his ongoing review of our patches. Thanks to our designers Danny Pagano for the illustrations, and Ross Zurowski for incorporating d3-flame-graph.

To learn more, watch our discussion with Jordan Whited and James Tucker on improving Tailscale's throughput.

← Back to index

Subscribe

Too much email? RSS Twitter

tailscale