KubeCon
CloudNativeCon
North America 2024

Cilium, eBPF, WireGuard: Can We Tame the Network Encryption Performance Gap?

Anton Protopopov & Daniel Borkmann

KubeCon North America 2024

ISOVALENT
now part of cisco

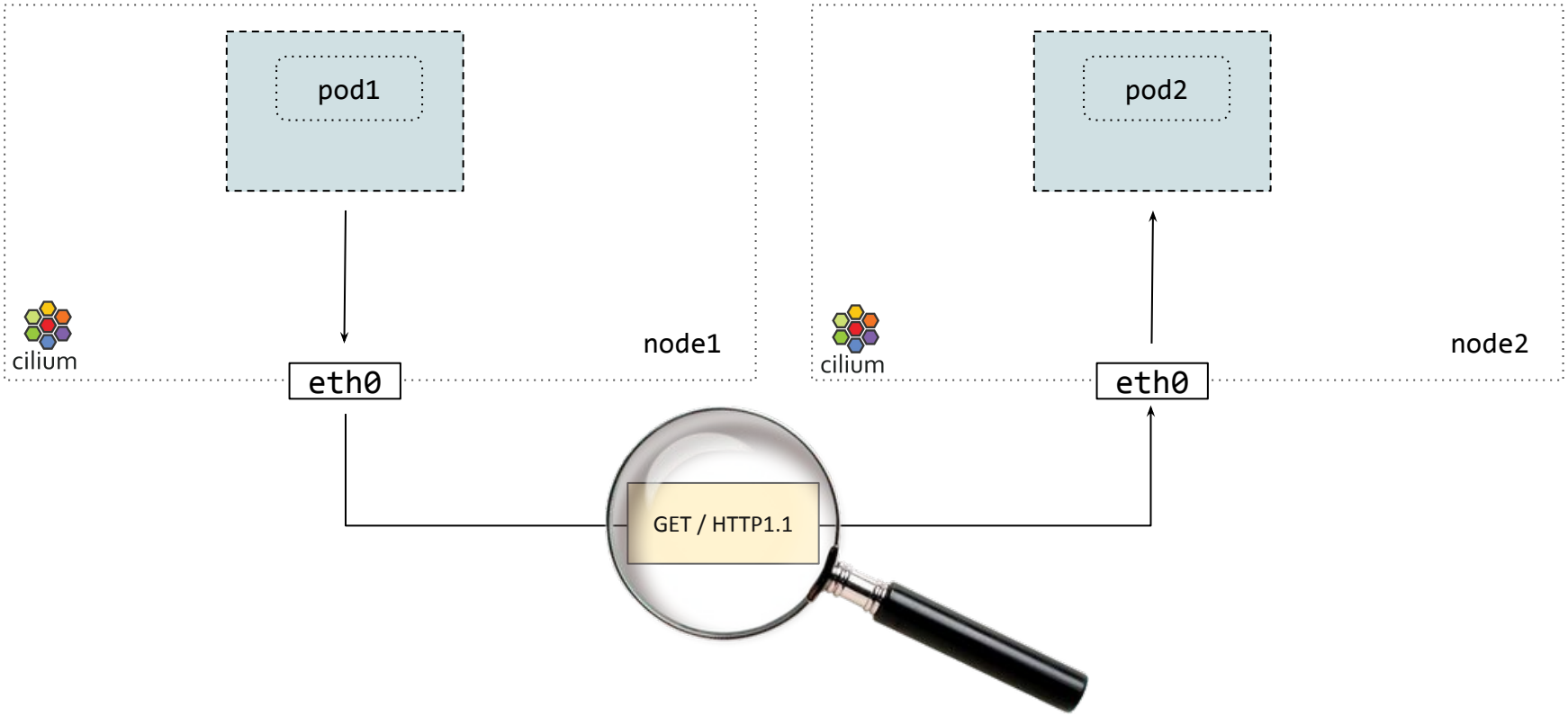
Experiment for the previous talk:

What would it take to turn the network performance knob to 11?

Experiment for *this* talk:

What would it take to maximize network performance for encrypted traffic?

Why Does ~~Anybody~~ Cilium Need Encryption?



Q: What are the options to enable transparent encryption within a cluster?

A: IPSec, WireGuard

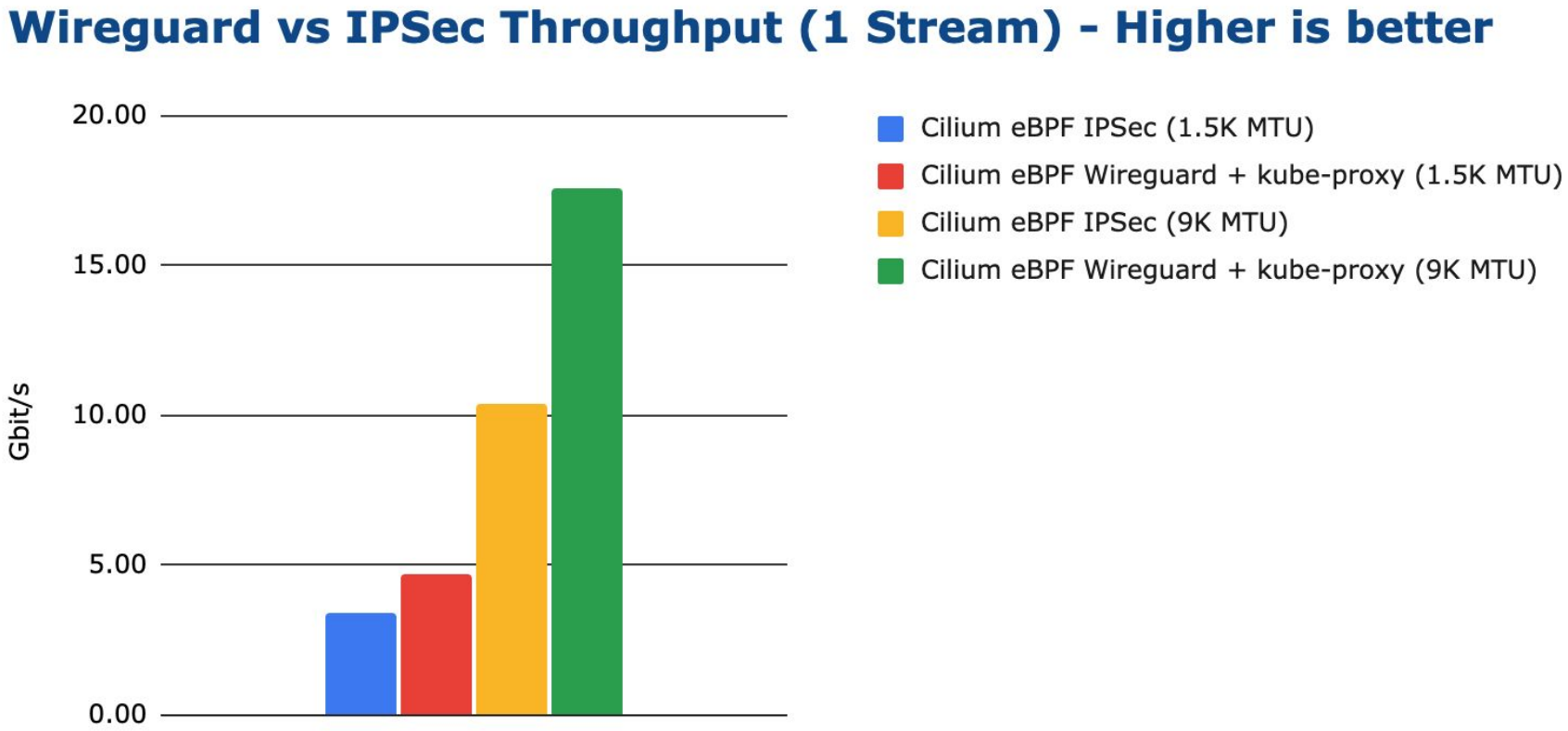
IPSec

- FIPS-compliant
- Performs better than WireGuard in some scenarios
- Normally, complex to manage, but Cilium takes care of most of the complexity
- Requires key management and rotation

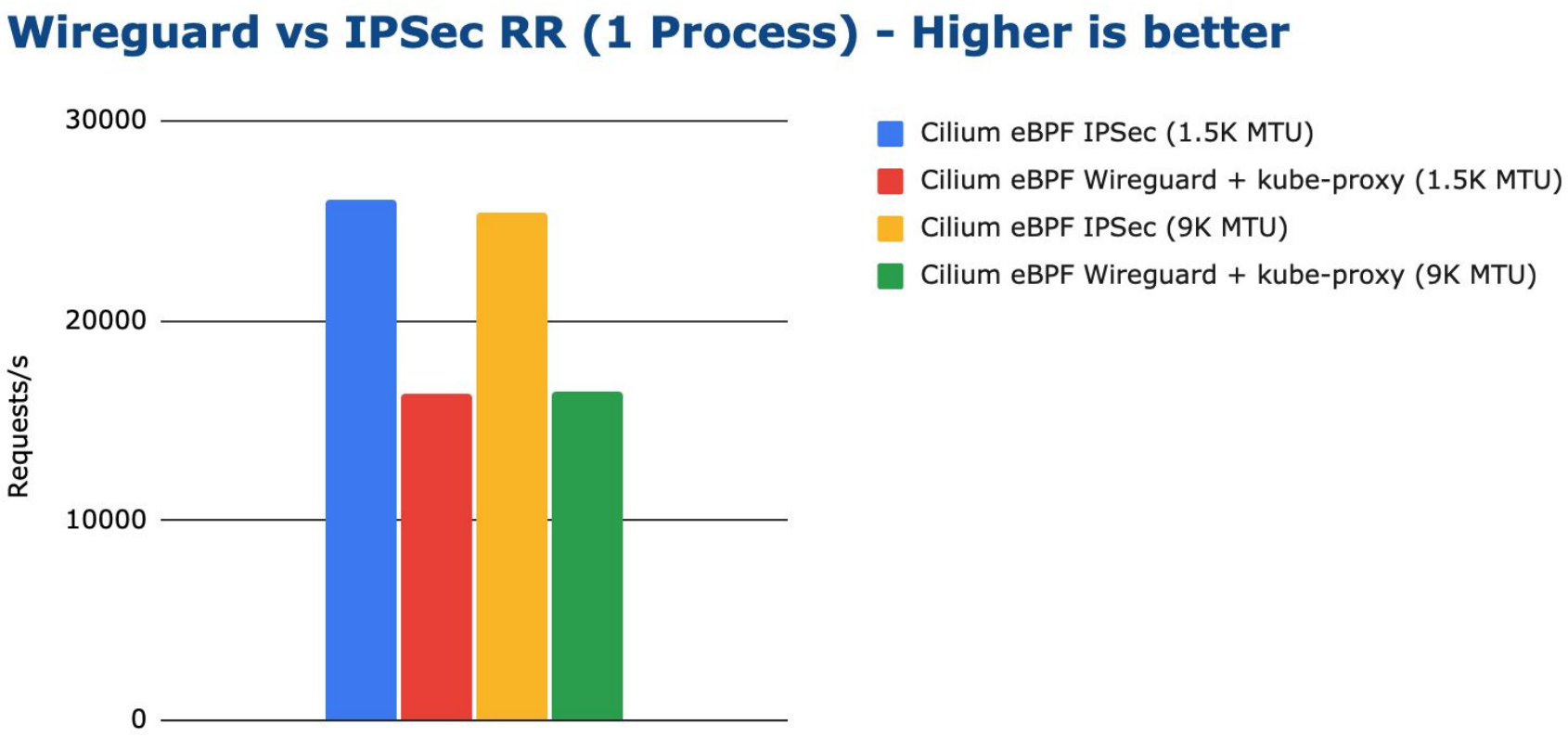
WireGuard

- Opinionated DJB crypto, formally verified
- Performs better than IPSec in some scenarios
- Easy to understand (source code is only ~7K lines)
- Easy to use and to manage
- Automatic key rotation and management

IPSec vs. WireGuard: Throughput



IPSec vs. WireGuard: Request/Response



req/resp-type workloads
stream-type workloads

WireGuard

- In the Linux kernel since v5.6
- The main design choice: be opinionated
  - Fixed crypto
  - Only L3 support

Opinionated Crypto

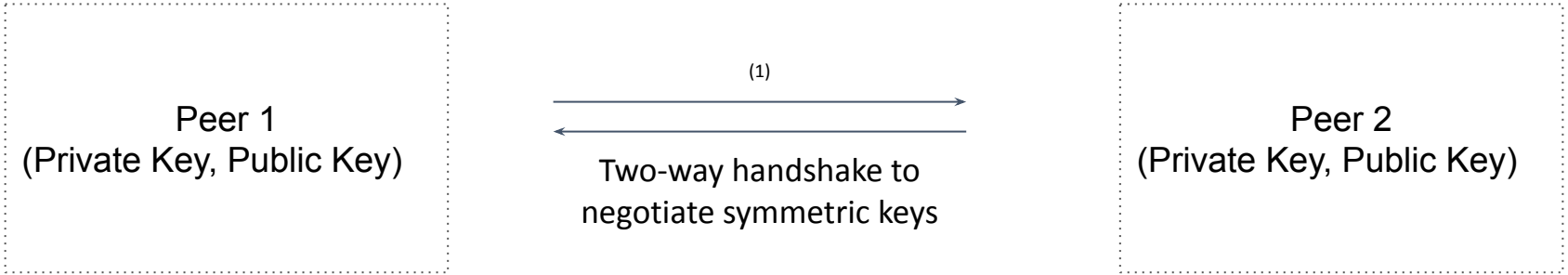
- Unlike IPSec/TLS WireGuard has fixed crypto, e.g.,
  - Curve25519 for ECDH
  - ChaCha20 for symmetric encryption, authenticated with Poly1305

WireGuard Setup, in a Nutshell

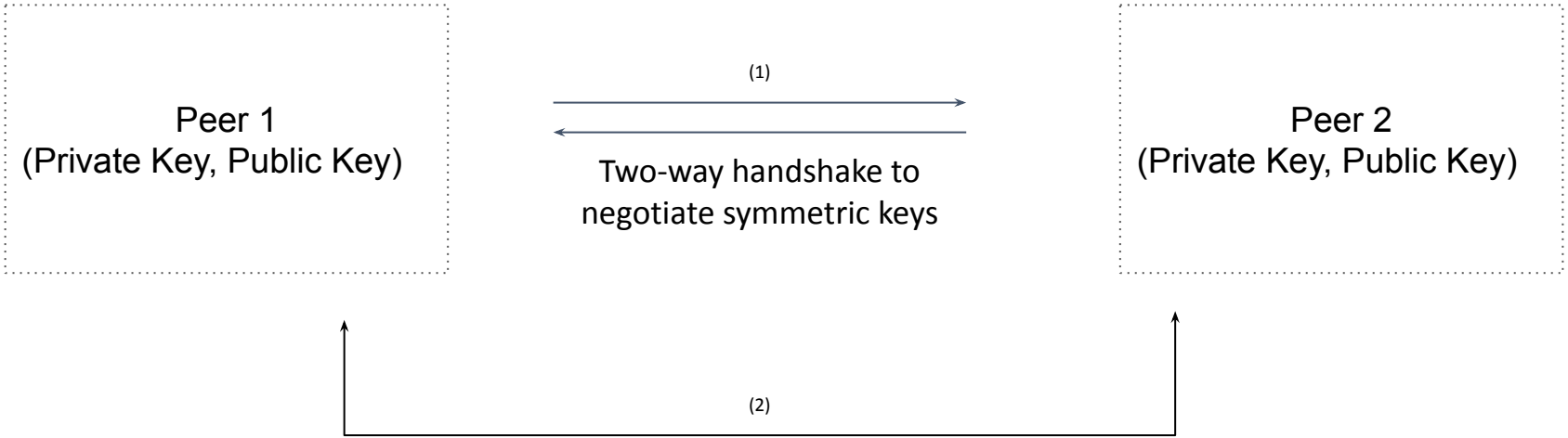
Peer 1
(Private Key, Public Key)

Peer 2
(Private Key, Public Key)

WireGuard Setup, in a Nutshell



WireGuard Setup, in a Nutshell



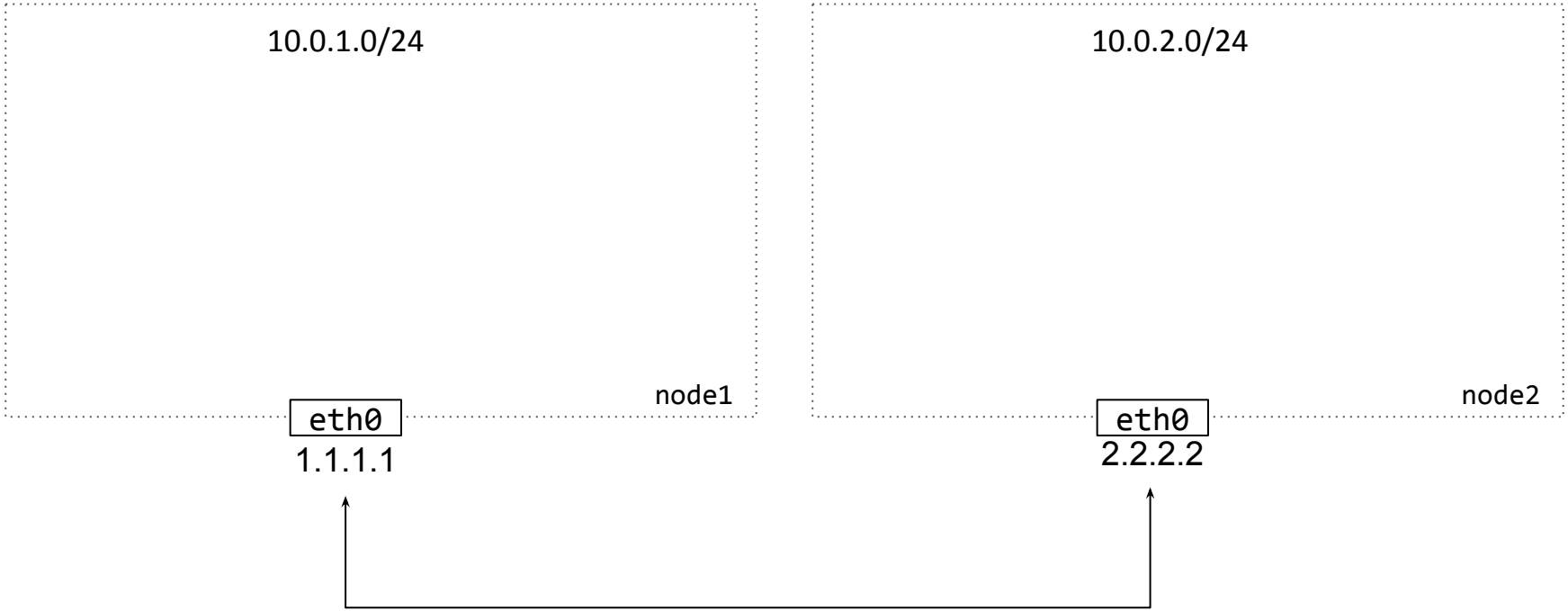
A UDP tunnel, encrypted using session symmetric keys. The session keys are auto-rotated thus providing Perfect Forward Secrecy (PFS)

Key Distribution Protocol

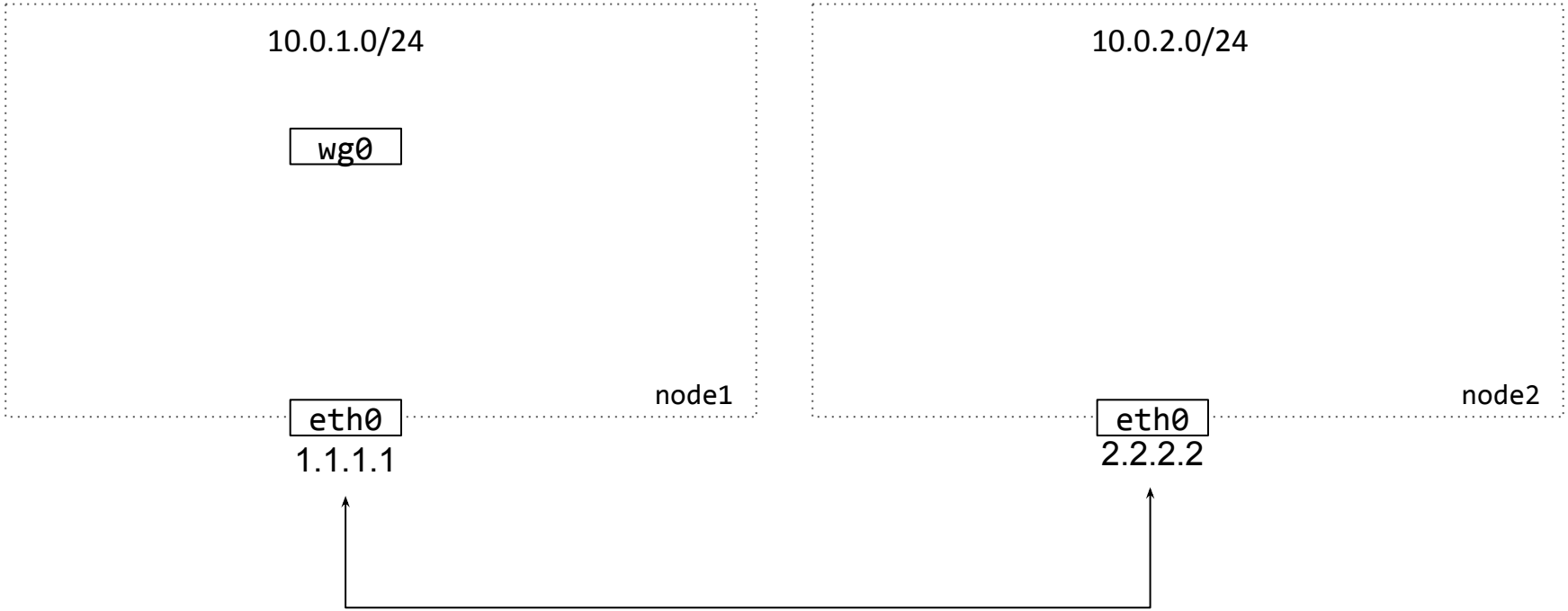
- BYOKDP* (inspired by SSH)

* BYOKDP: Bring Your Own Key Distribution Protocol

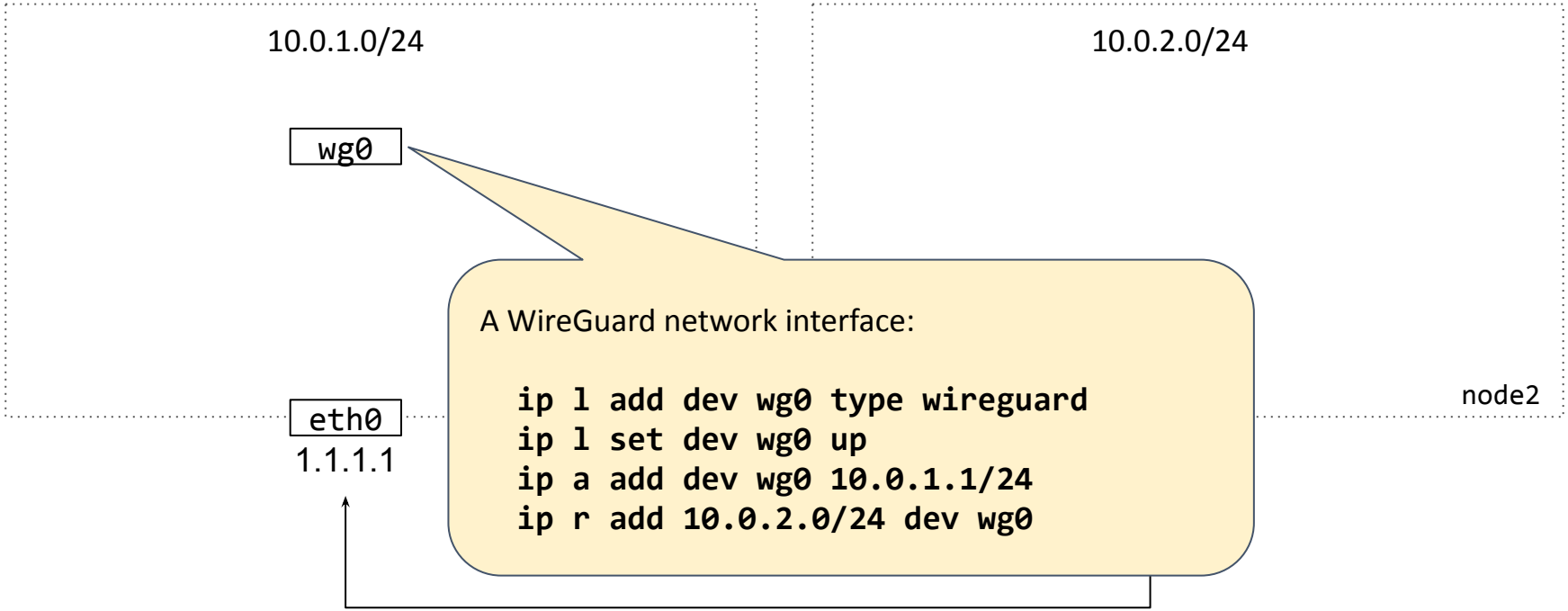
WireGuard Configuration: Cryptokey Routing



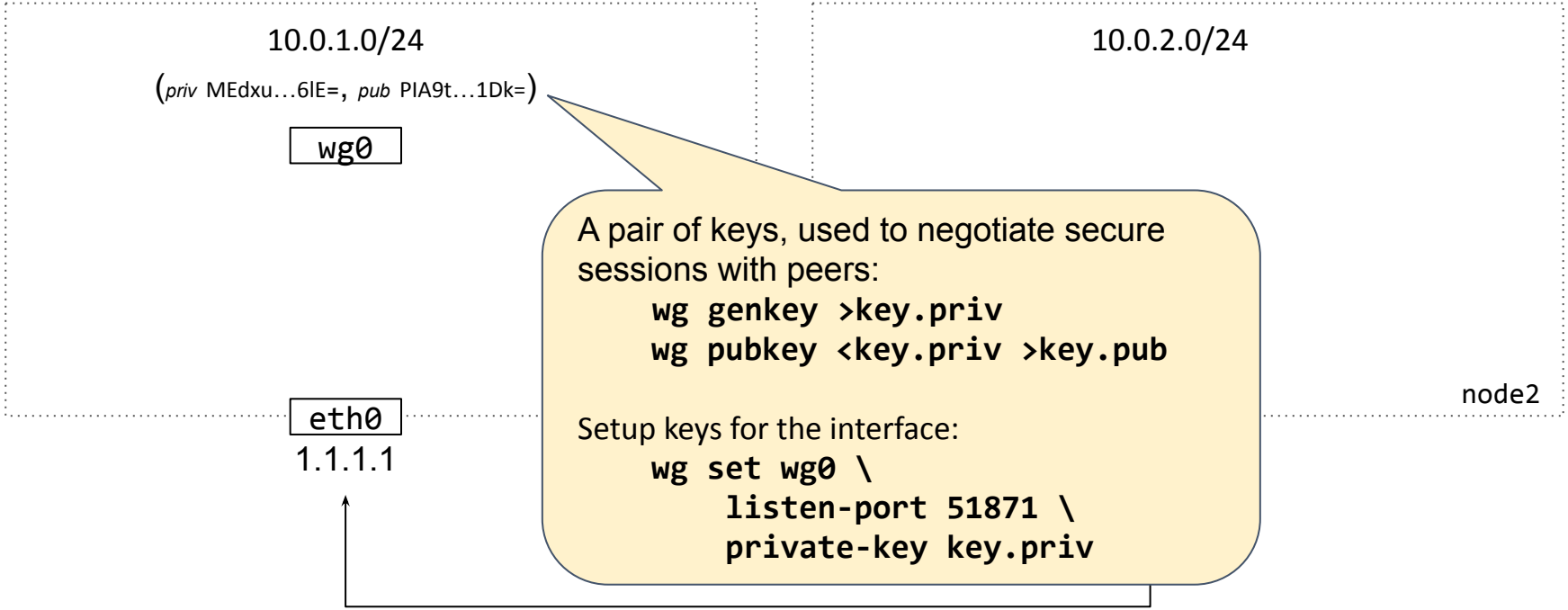
WireGuard Configuration: Cryptokey Routing



WireGuard Configuration: Cryptokey Routing



WireGuard Configuration: Cryptokey Routing



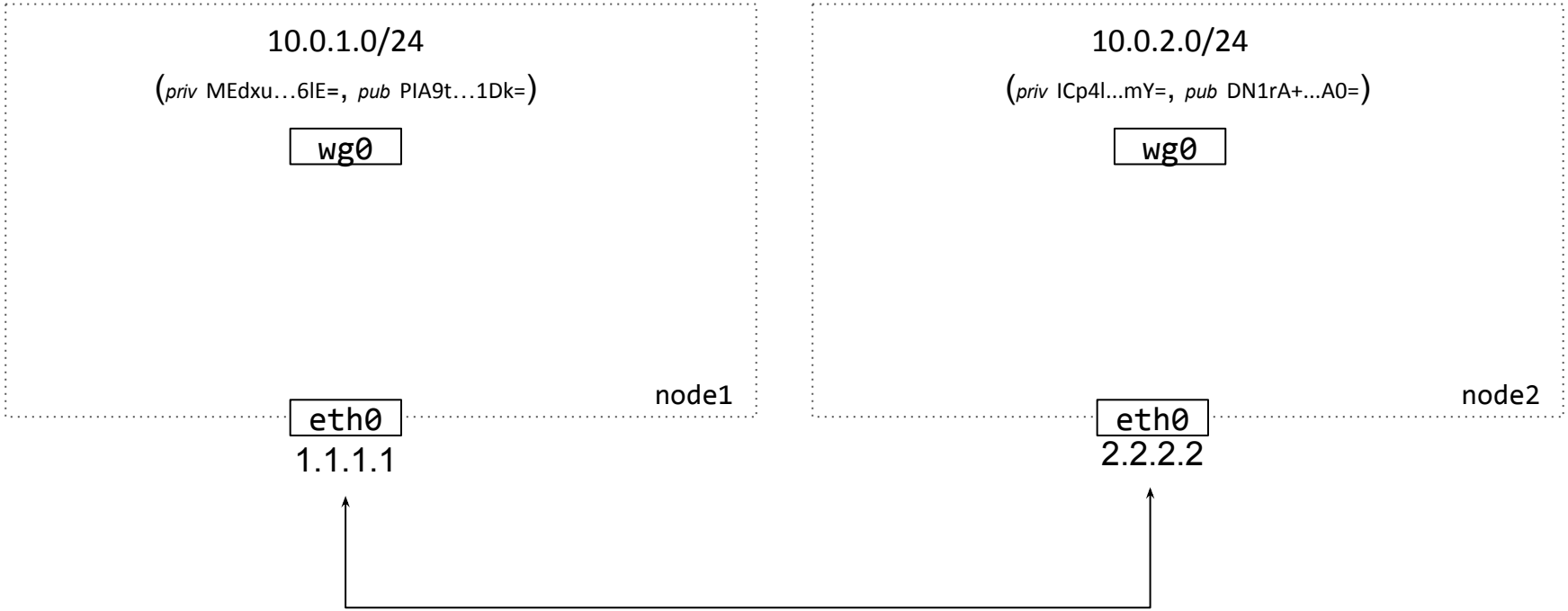
A pair of keys, used to negotiate secure sessions with peers:

```
wg genkey >key.priv
wg pubkey <key.priv >key.pub
```

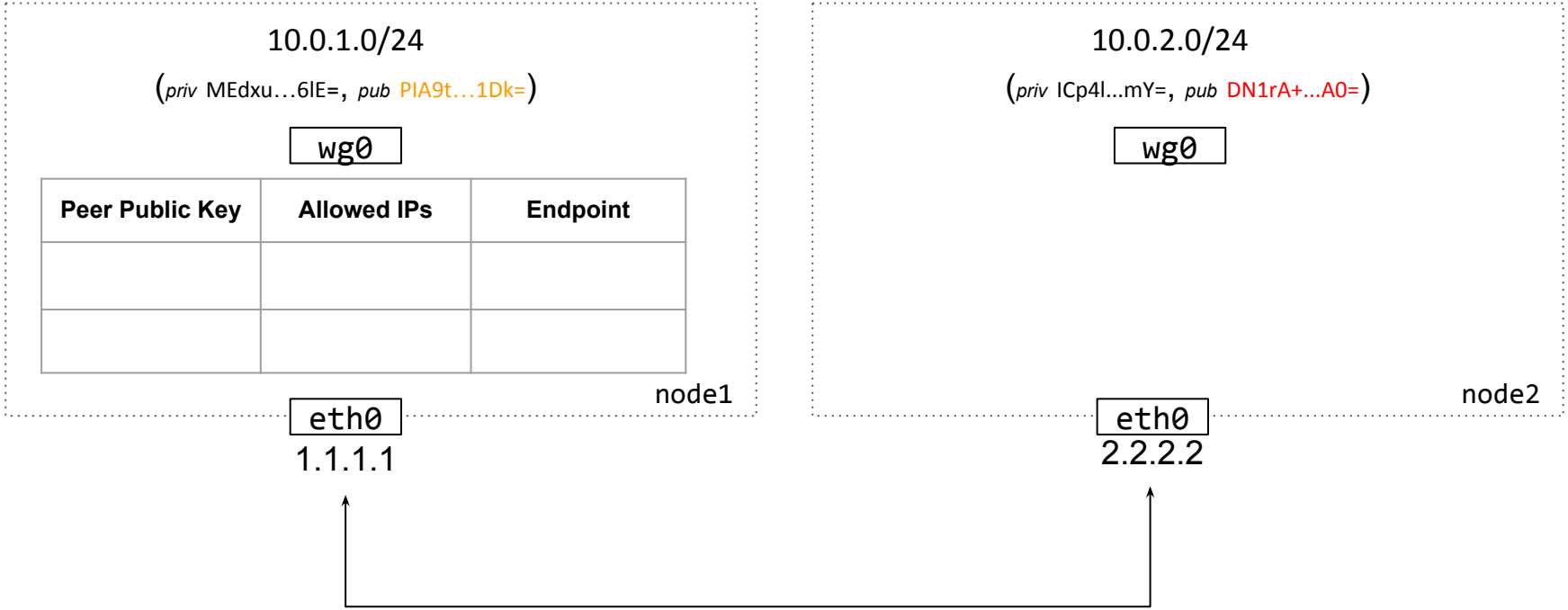
Setup keys for the interface:

```
wg set wg0 \
    listen-port 51871 \
    private-key key.priv
```

WireGuard Configuration: Cryptokey Routing



WireGuard Configuration: Cryptokey Routing



WireGuard Configuration: Cryptokey Routing

10.0.1.0/24

(*priv* MEdxu…6lE=, *pub* PIA9t…1Dk=)

wg0

Peer Public Key	Allowed IPs	Endpoint
DN1rA+...A0=	10.0.2.0/24	2.2.2.2

10.0.2.0/24

(*priv* ICp4l...mY=, *pub* DN1rA+...A0=)

wg0

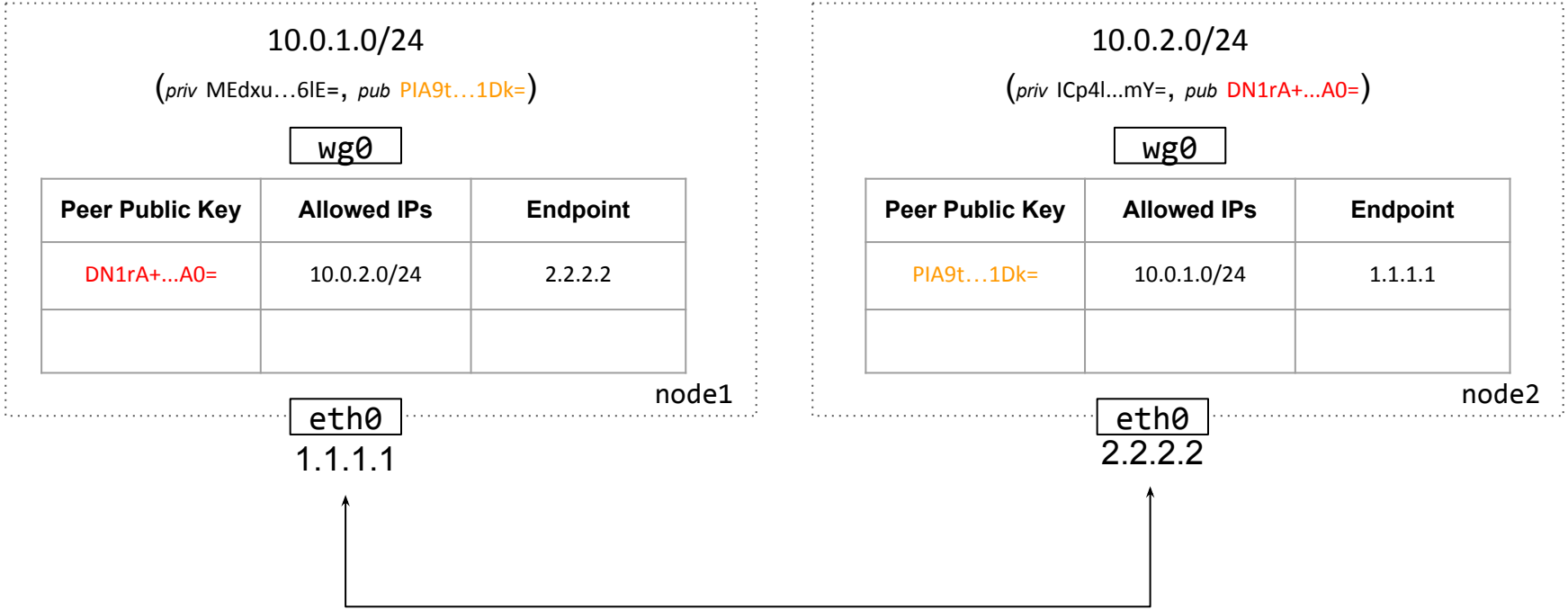
node2

eth0
2.2.2.2

Add a peer:

```
wg set wg0 \
        peer DN1rA+...A0= \
        allowed-ips 10.0.2.0/24 \
        endpoint 2.2.2.2:51871
```

WireGuard Configuration: Cryptokey Routing



node1

Peer Public Key	Allowed IPs	Endpoint
DN1rA+…A0=	10.0.2.0/24	2.2.2.2

node2

Peer Public Key	Allowed IPs	Endpoint
PIA9t…1Dk=	10.0.1.0/24	1.1.1.1

WireGuard Configuration: Cryptokey Routing

10.0.1.0/24

(*priv* MEdxu…6lE=, *pub* PIA9t…1Dk=)

wg0

Peer Public Key	Allowed IPs	Endpoint
DN1rA+…A0=	10.0.2.0/24	2.2.2.2
+52TbS…xM=	10.0.3.0/24	3.3.3.3

node1

eth0

1.1.1.1

10.0.2.0/24

(*priv* ICp4l…mY=, *pub* DN1rA+…A0=)

wg0

Peer Public Key	Allowed IPs	Endpoint
PIA9t…1Dk=	10.0.1.0/24	1.1.1.1
+52TbS…xM=	10.0.3.0/24	3.3.3.3

node2

eth0

2.2.2.2

10.0.3.0/24

(C0vld…19=, +52TbS…xM=)

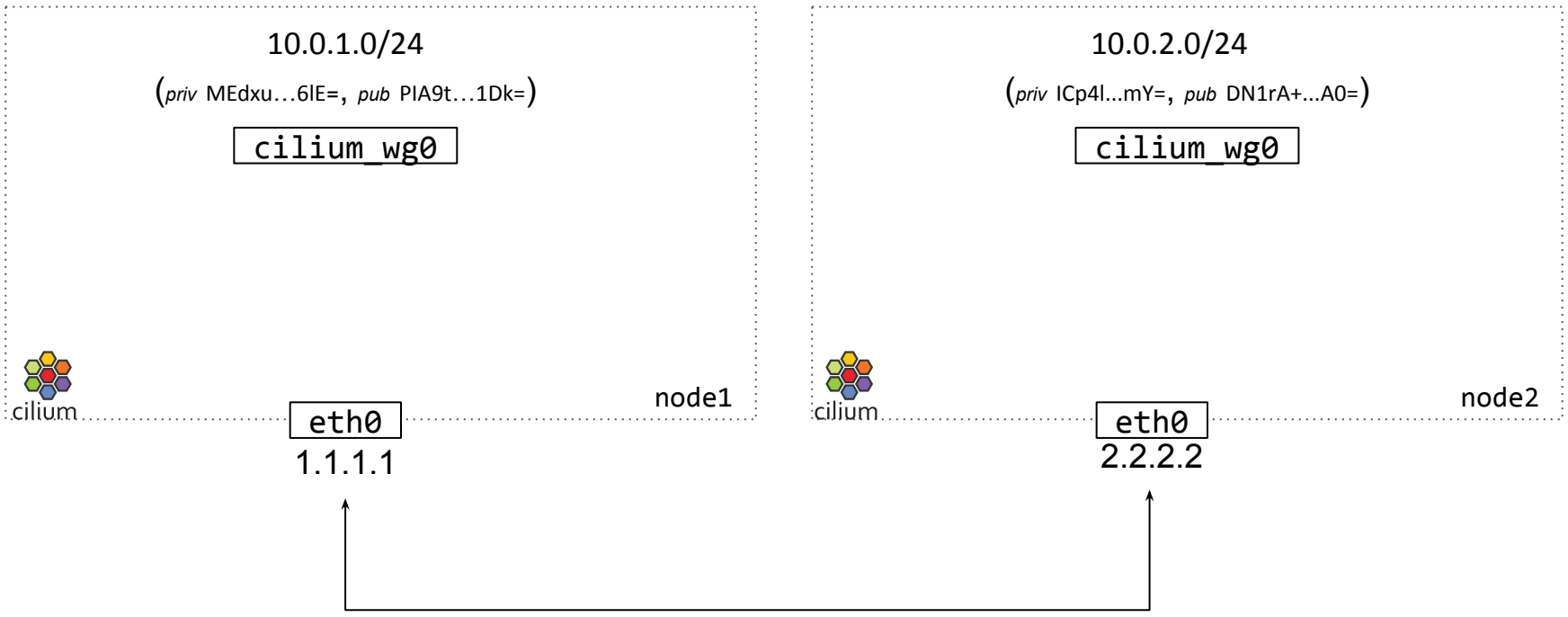
3.3.3.3

WireGuard in Cilium

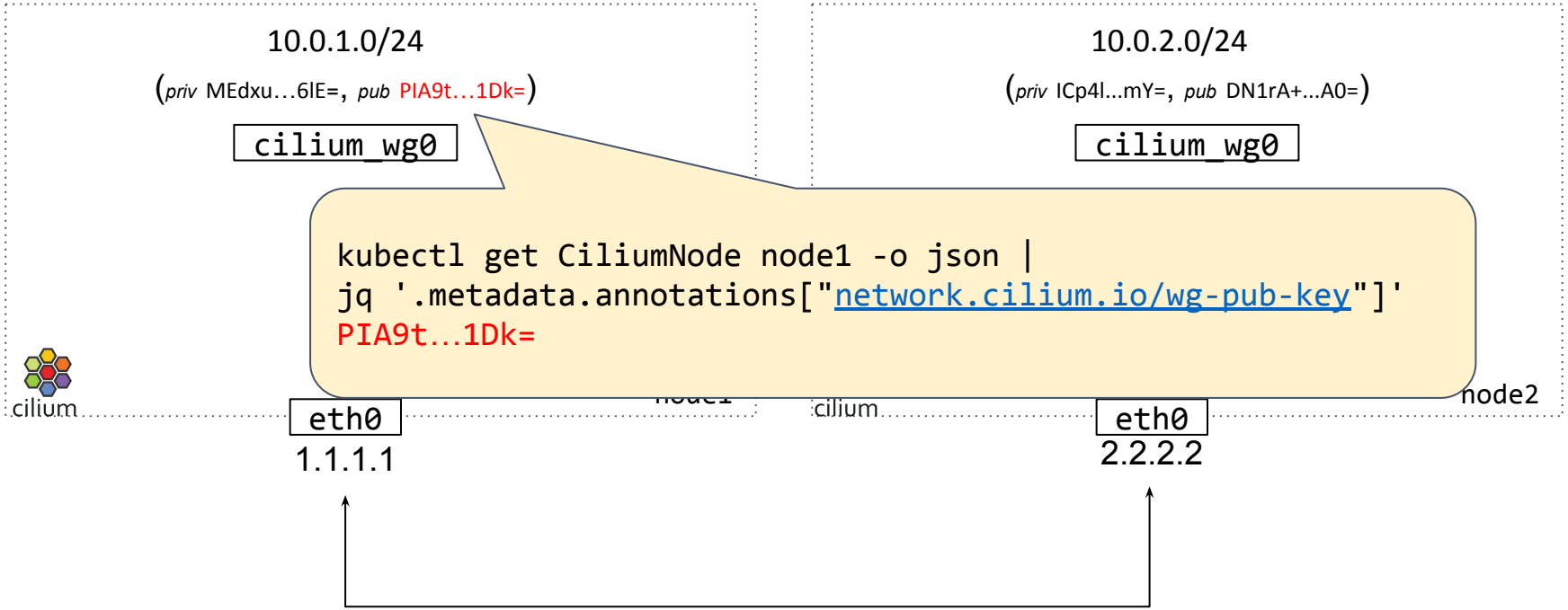
- Supported from Cilium 1.10 for encrypting Pod + node traffic
- To install Cilium with WireGuard enabled:

```
cilium install --set encryption.enabled=true
               --set encryption.type=wireguard
```

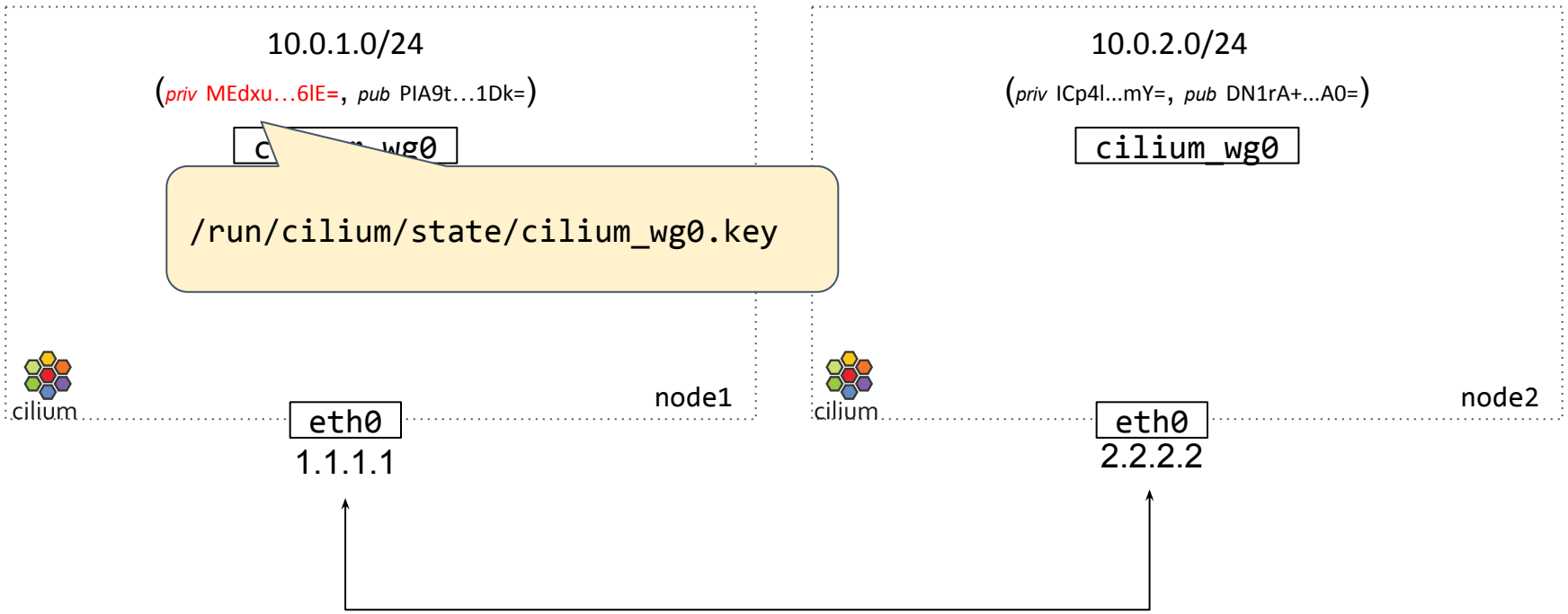
WireGuard in Cilium



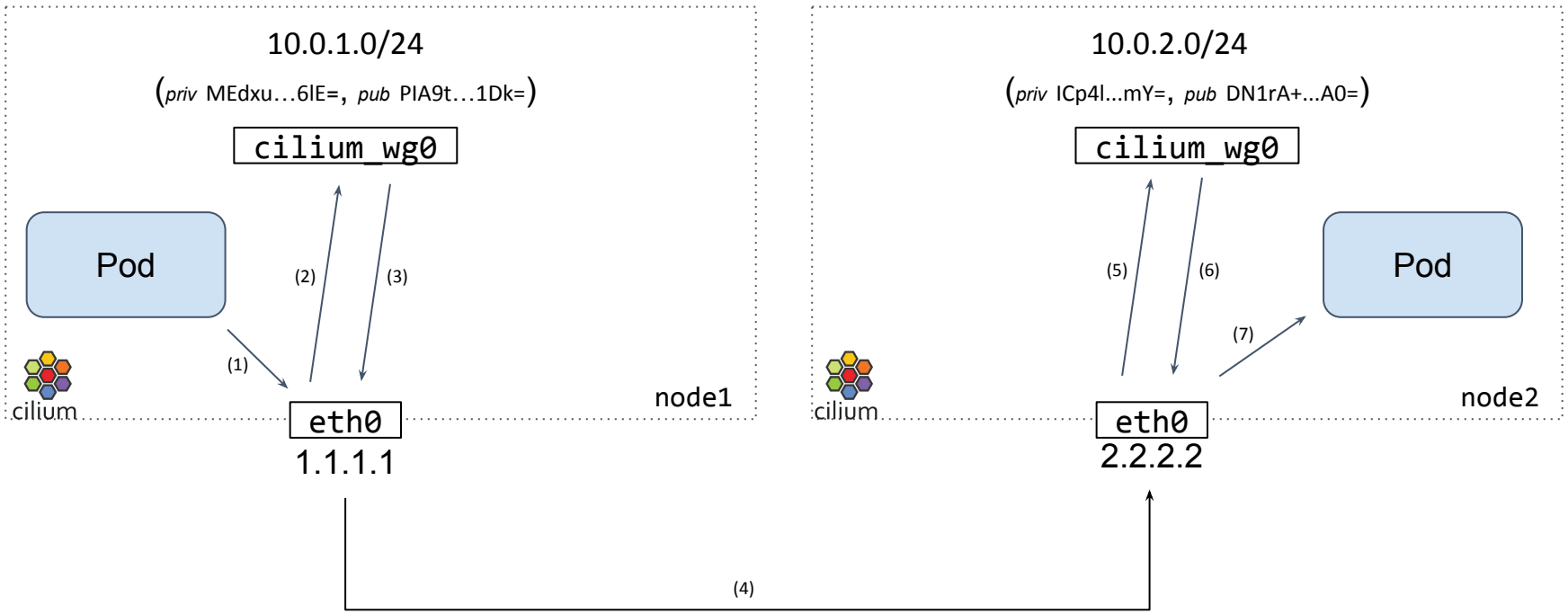
WireGuard in Cilium: Key Distribution



WireGuard in Cilium: Key Rotation



WireGuard in Cilium: Packet Path



* See the full list of possible packet flows and required configurations here

Transparent Encryption in Cilium: Labs

- Cilium Transparent Encryption with IPSec and WireGuard

WireGuard is natively integrated into Cilium. But, what about its performance?

Recently our attention was attracted by the "Userspace isn't slow, some kernel interfaces are!" article by Tailscale.

WireGuard driver vs WireGuard-go

- WireGuard-go got support for UDP GRO/GSO
  - Blog: “Userspace isn't slow, some kernel interfaces are!”
  - Blog: “Surpassing 10Gb/s over Tailscale”

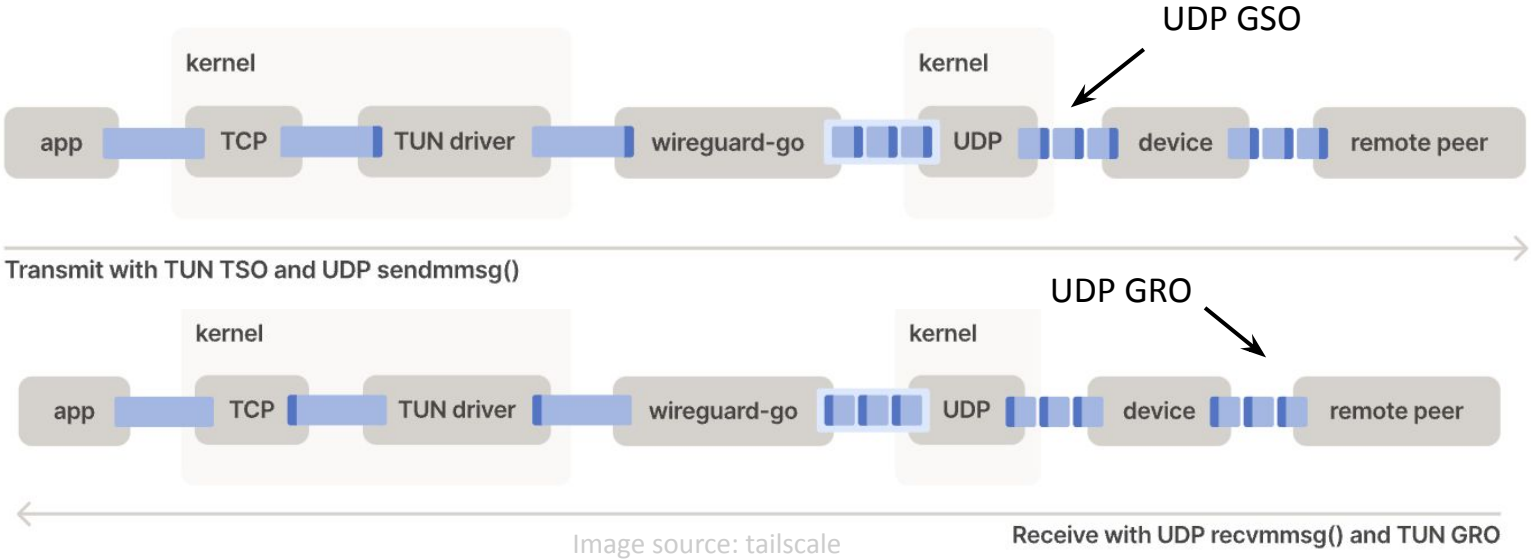


Image source: tailscale

Cilium supported WireGuard-go in the past as a fallback mechanism for old kernels.

WireGuard-go support was removed from Cilium due to widespread native kernel support, and connectivity disruption upon agent restarts.

WireGuard-go support was removed from Cilium due to widespread native kernel support, and connectivity disruption upon agent restarts. WireGuard kernel driver does not suffer from this.

Now onto benchmarking unencrypted traffic vs WireGuard driver vs WireGuard-go ...

Step 1: Getting to a stable baseline.

```
root@lab-node-3:/home/darkstar/setup# netperf -H 10.0.0.2 -t TCP_STREAM -l30
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.0.0.2 () port 0 AF_INET : demo
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

262144  32768  32768    30.00    97314.19
```

Hardware tl;dr:

- AMD Ryzen 9 3950X @ 3.5 GHz, 128G RAM @ 3.2 GHz, PCIe 4.0
- 100Gbit/s dual port ConnectX-6 Dx (mlx5)

```
root@lab-node-3:/home/darkstar/net# cat /proc/cpuinfo | grep processor | wc -l
16

0a:00.0 Ethernet controller: Mellanox Technologies MT2892 Family [ConnectX-6 Dx]
0a:00.1 Ethernet controller: Mellanox Technologies MT2892 Family [ConnectX-6 Dx]

[    2.993541] mlx5_core 0000:0a:00.1: firmware version: 22.32.2004
[    2.993835] mlx5_core 0000:0a:00.1: 252.048 Gb/s available PCIe bandwidth (16.0 GT/s PCIe x16 link)
```

Software tl;dr:

- Linux kernel: 6.12.0-rc5
- Default config not always suited for network-heavy workloads

```
# CONFIG_INIT_STACK_ALL_PATTERN is not set
# CONFIG_INIT_STACK_ALL_ZERO is not set
# CONFIG_INIT_ON_ALLOC_DEFAULT_ON is not set
# CONFIG_INIT_ON_FREE_DEFAULT_ON is not set

# CONFIG_CPU_MITIGATIONS is not set
```

```
CONFIG_PREEMPT_NONE=y
# CONFIG_PREEMPT_VOLUNTARY is not set
# CONFIG_PREEMPT is not set
# CONFIG_PREEMPT_RT is not set

CONFIG_MAX_SKB_FRAGS=20
```

Software tl;dr:

- Contention/overhead in IOMMU: `iommu=off`
- Enabling LRO: `ethtool -K eth0 large-receive-offload on`
- Adapting to 8k MTU, allowing better frag placement for GRO

Software tl;dr:

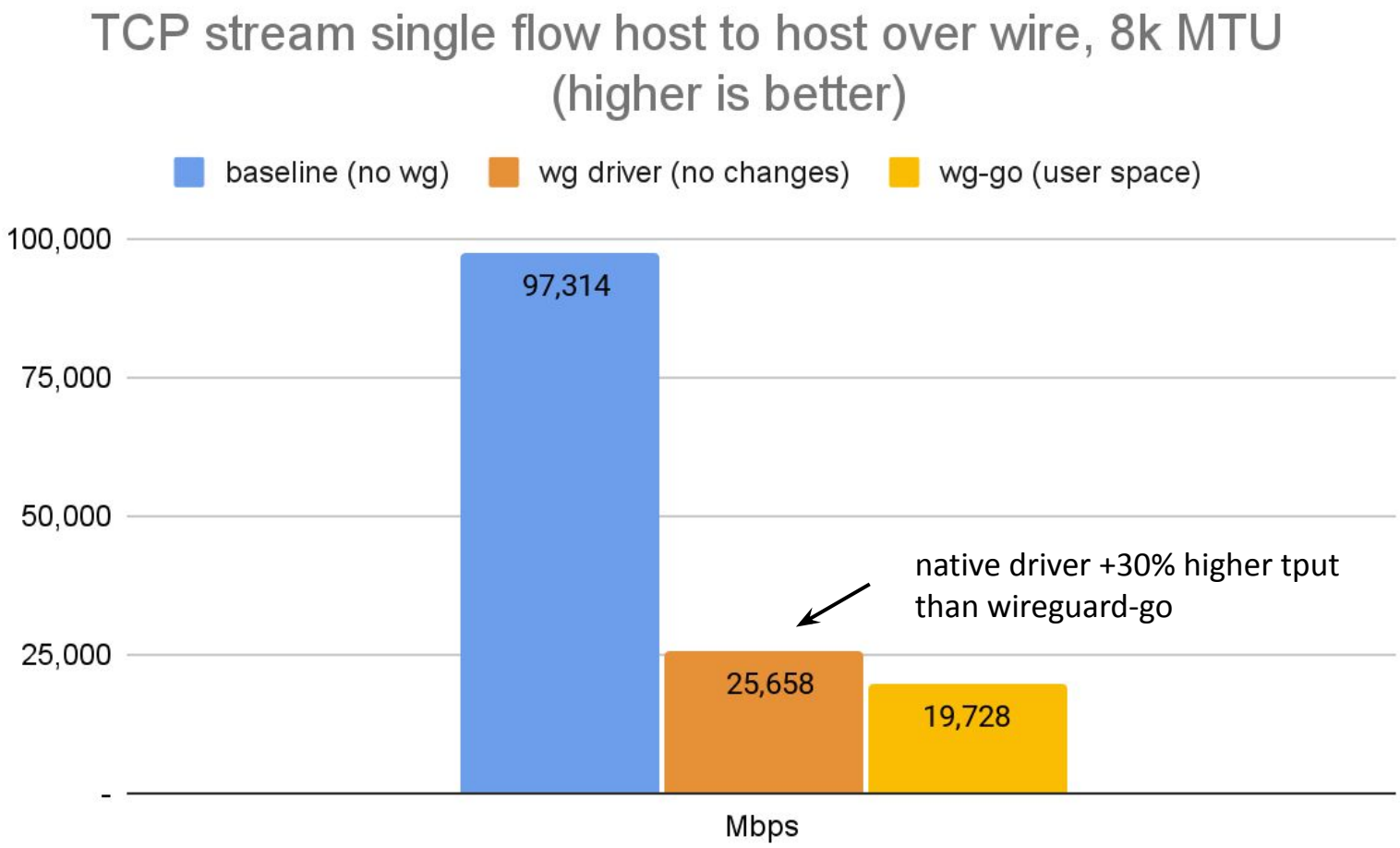
- Pinning NIC RX/TX IRQ affinity 1:1 to CPUs

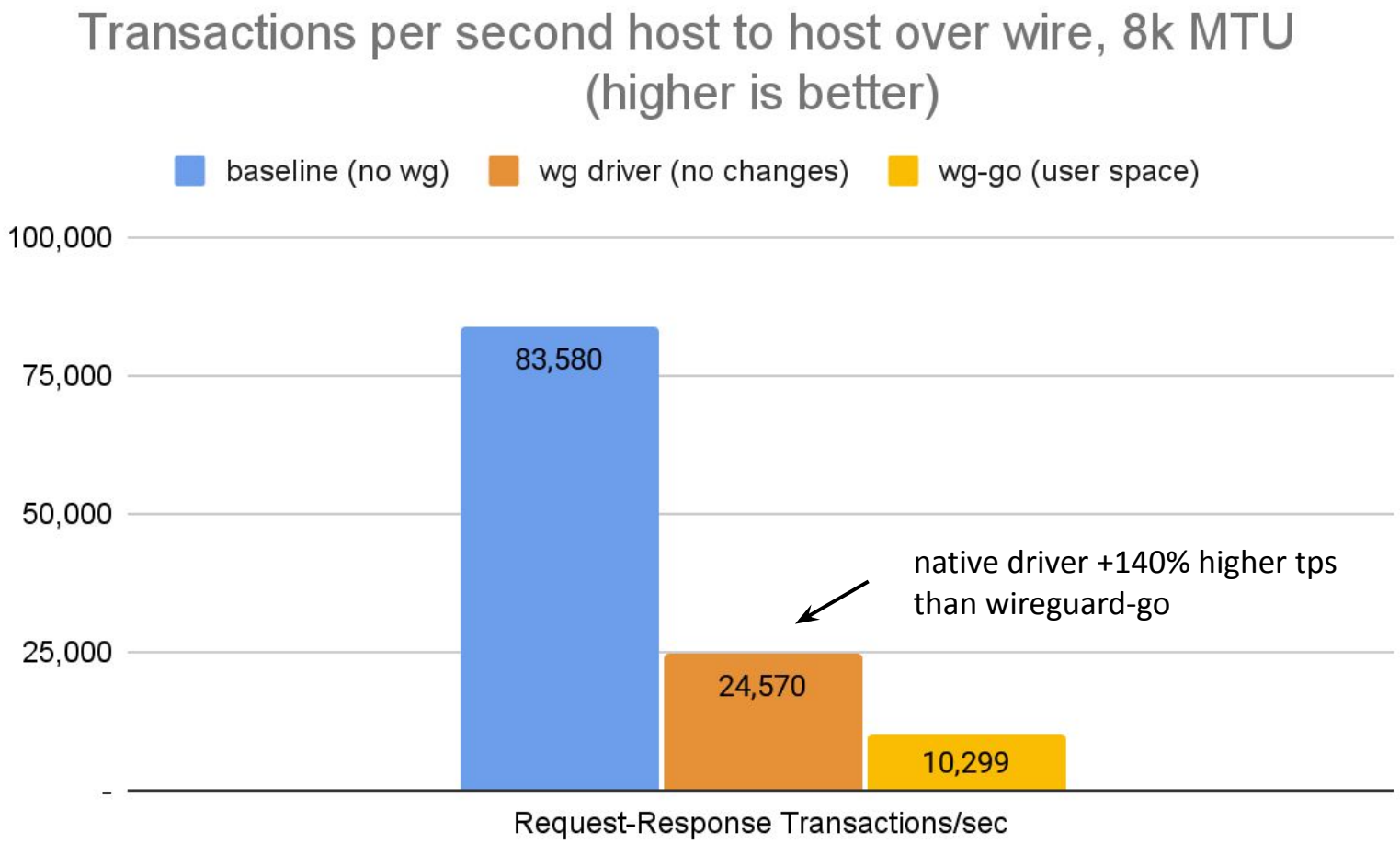
```
eth0-rx-0  CPU0
eth0-rx-1  CPU1
eth0-tx-0  CPU0
eth0-tx-1  CPU1
...
```

```
for CPU in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor; do
        echo performance > $CPU
done
```

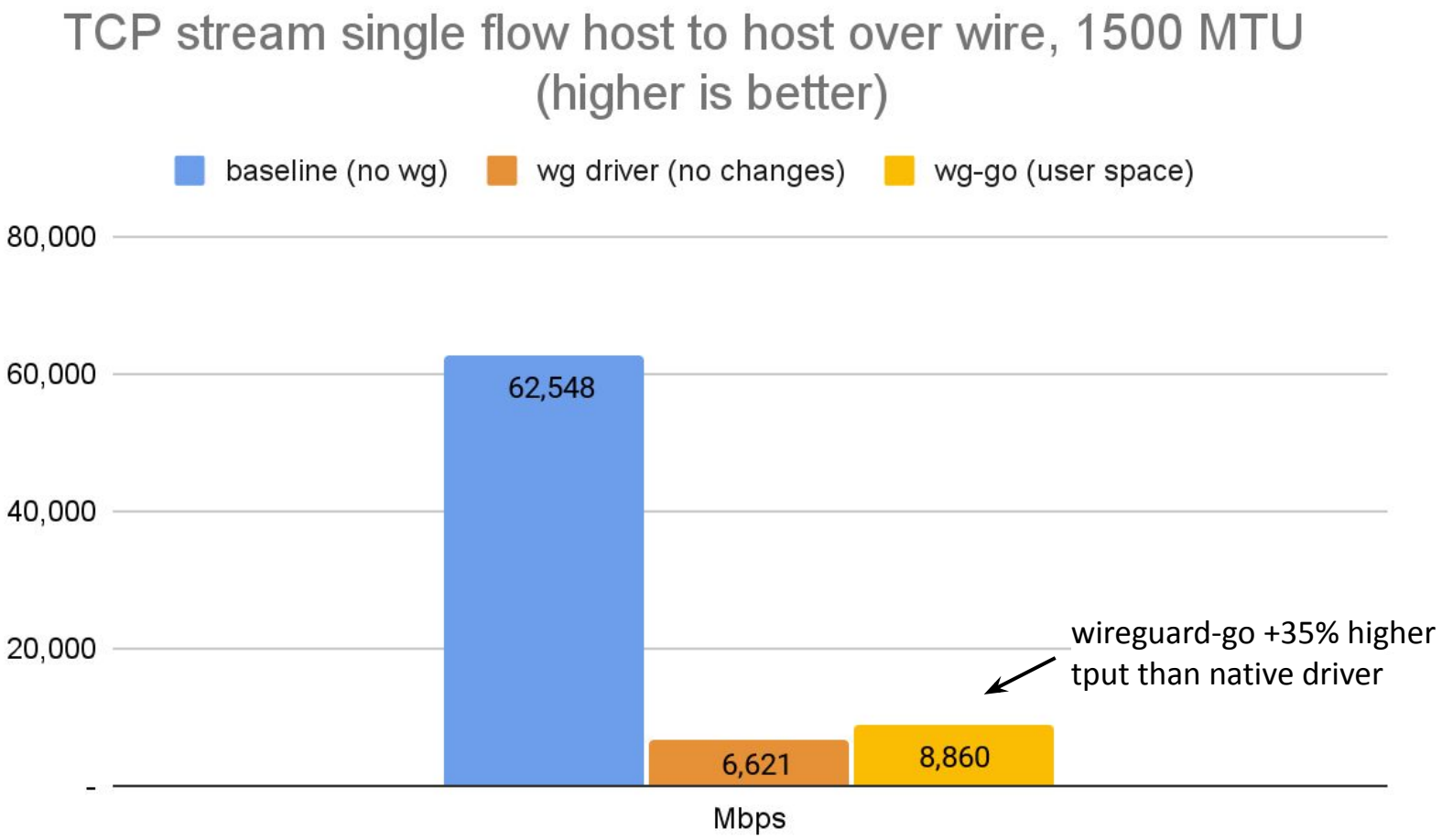
Software tl;dr:

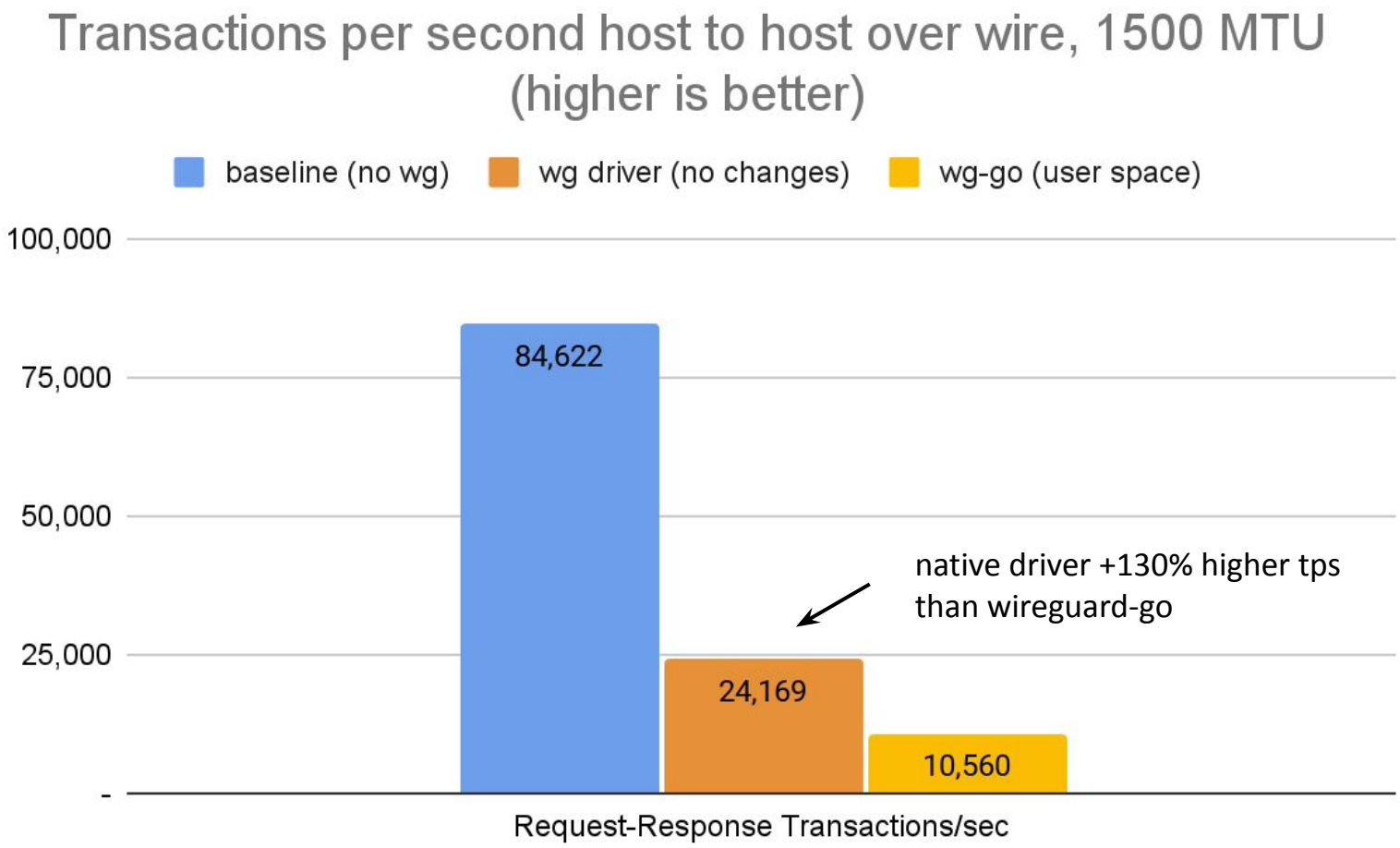
- Enabling BIG TCP for more aggressive GRO/TSO aggregation
- Tweaking net.ipv4.tcp_{r,w}mem and net.core.{r,w}mem_max
- tuned-adm profile network-latency


TCP stream single flow host to host over wire, 8k MTU
(higher is better)
baseline (no wg)
wg driver (no changes)
wg-go (user space)
100,000
75,000
50,000
25,000
-
97,314
25,658
19,728
Mbps
native driver +30% higher tput
than wireguard-go


Transactions per second host to host over wire, 8k MTU
(higher is better)
baseline (no wg)
wg driver (no changes)
wg-go (user space)
100,000
75,000
50,000
25,000
-
83,580
24,570
10,299
native driver +140% higher tps
than wireguard-go
Request-Response Transactions/sec

How does it compare to default 1500 MTU?


TCP stream single flow host to host over wire, 1500 MTU
(higher is better)
baseline (no wg)
wg driver (no changes)
wg-go (user space)
80,000
60,000
40,000
20,000
-
62,548
6,621
8,860
Mbps
wireguard-go +35% higher
tput than native driver


Transactions per second host to host over wire, 1500 MTU
(higher is better)
baseline (no wg)
wg driver (no changes)
wg-go (user space)
100,000
75,000
50,000
25,000
-
84,622
24,169
10,560
native driver +130% higher tps
than wireguard-go
Request-Response Transactions/sec

Step 2: Can we do better for single streams?

How does WireGuard handle GRO/GSO?

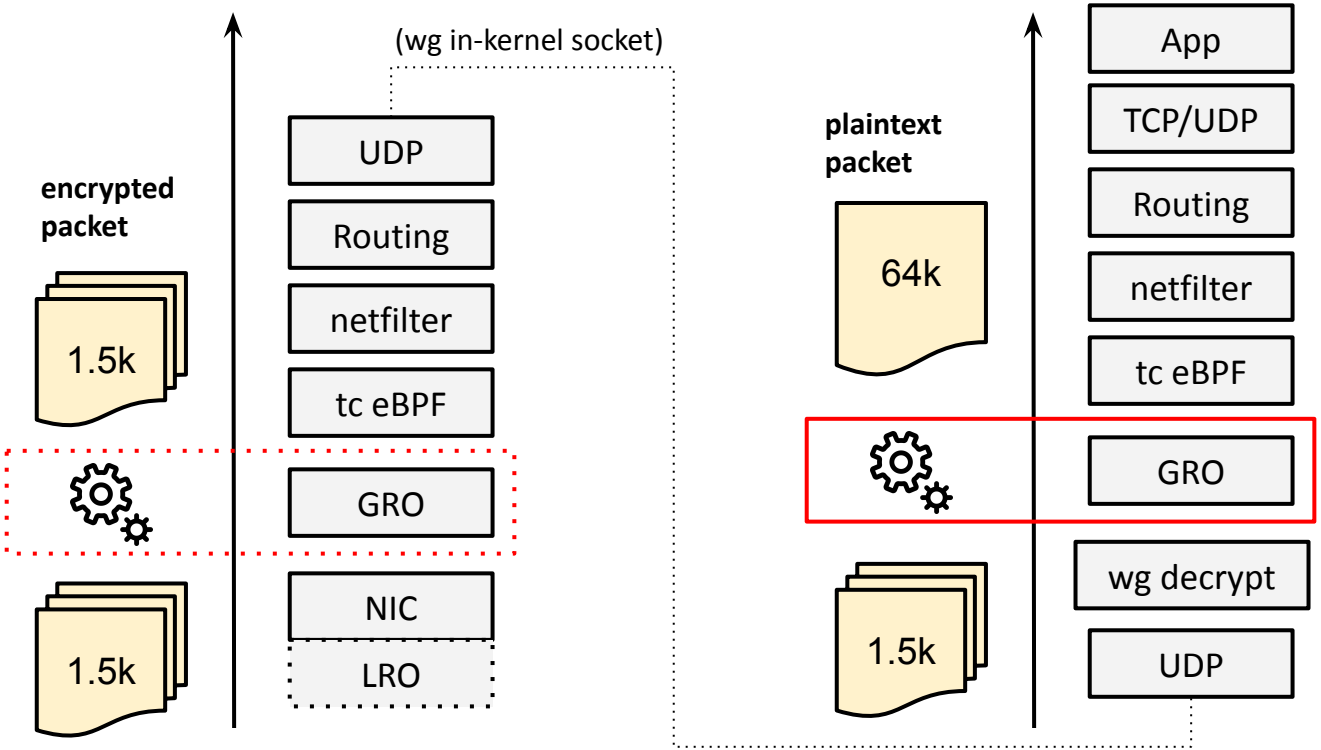
Receive Path 1/2:

```
swapper       0 [004]  1806.987105: net:netif_receive_skb: dev=enp10s0f0np0 skbaddr=0xffff8f0963a4da00 len=8172
        ffffffffa952c1e0 __netif_receive_skb_core.constprop.0+0x5f0 ([kernel.kallsyms])
        ffffffffa952c1e0 __netif_receive_skb_core.constprop.0+0x5f0 ([kernel.kallsyms])
        ffffffffa952ce7e __netif_receive_skb_list_core+0xee ([kernel.kallsyms])
        ffffffffa952d4f3 netif_receive_skb_list_internal+0x1d3 ([kernel.kallsyms])
        ffffffffa952dde4 napi_complete_done+0x74 ([kernel.kallsyms])
        ffffffffc0664cce mlx5e_napi_poll+0x16e ([kernel.kallsyms])
        ffffffffa952df2d __napi_poll+0x2d ([kernel.kallsyms])
        ffffffffa952e5ee net_rx_action+0x20e ([kernel.kallsyms])
        ffffffffa88dfe22 handle_softirqs+0xe2 ([kernel.kallsyms])
        ffffffffa88e00f4 __irq_exit_rcu+0xa4 ([kernel.kallsyms])
        ffffffffa88e036e irq_exit_rcu+0xe ([kernel.kallsyms])
        ffffffffa98152c8 common_interrupt+0x98 ([kernel.kallsyms])
        ffffffffa9a00da7 asm_common_interrupt+0x27 ([kernel.kallsyms])
        ffffffffa9819765 cpuidle_enter_state+0xd5 ([kernel.kallsyms])
        ffffffffa94af1de cpuidle_enter+0x2e ([kernel.kallsyms])
        ffffffffa893cf22 call_cpuidle+0x22 ([kernel.kallsyms])
        ffffffffa8942d73 do_idle+0x1d3 ([kernel.kallsyms])
        ffffffffa8942f89 cpu_startup_entry+0x29 ([kernel.kallsyms])
        ffffffffa888d653 start_secondary+0x113 ([kernel.kallsyms])
        ffffffffa8845fdd common_startup_64+0x13e ([kernel.kallsyms])
```

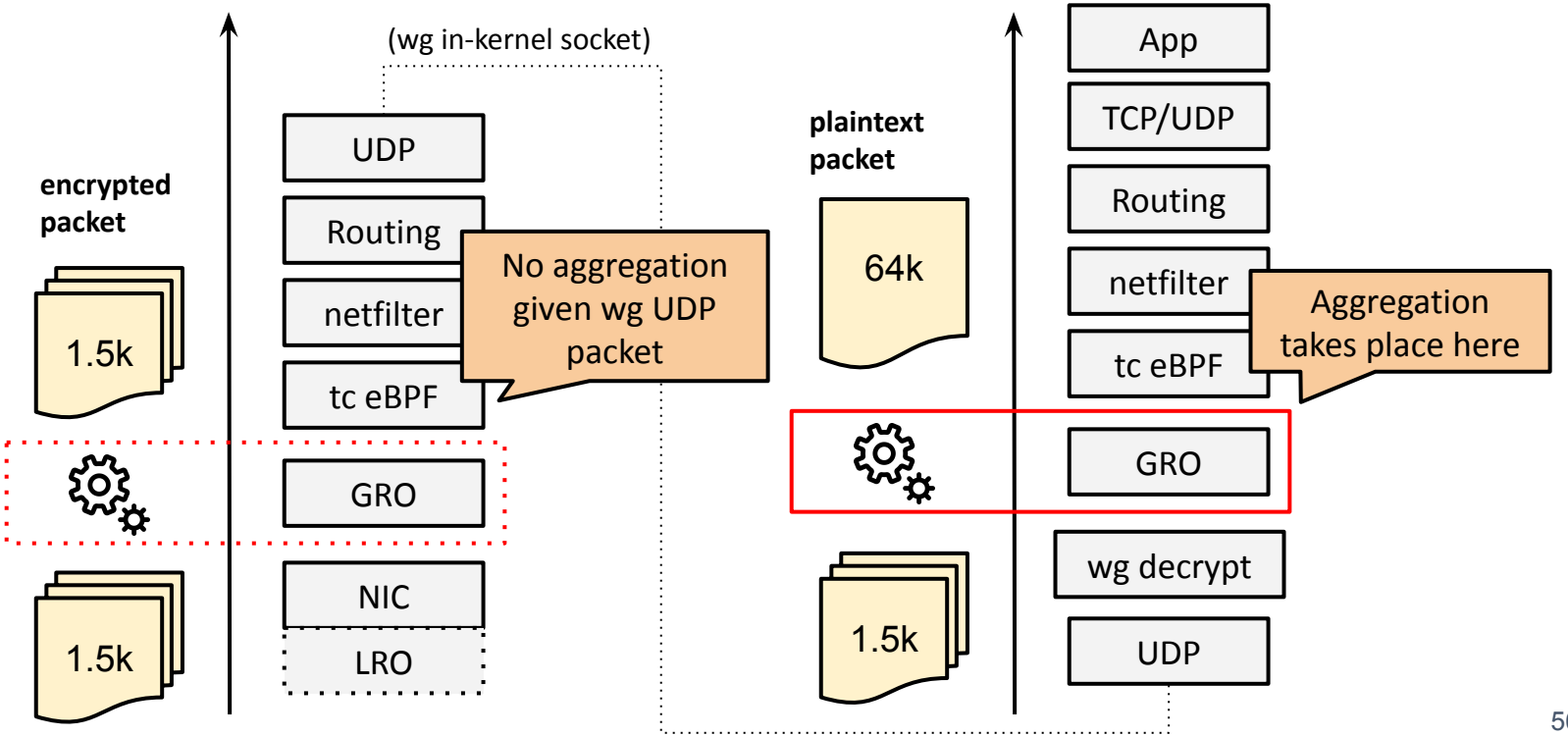
Receive Path 2/2:

```
kthreadd    5926 [003]  1806.987116: net:netif_receive_skb: dev=wg0 skbaddr=0xffff8f0963a4c700 len=48412
        ffffffffa952c1e0 __netif_receive_skb_core.constprop.0+0x5f0 ([kernel.kallsyms])
        ffffffffa952c1e0 __netif_receive_skb_core.constprop.0+0x5f0 ([kernel.kallsyms])
        ffffffffa952ce7e __netif_receive_skb_list_core+0xee ([kernel.kallsyms])
        ffffffffa952d4f3 netif_receive_skb_list_internal+0x1d3 ([kernel.kallsyms])
        ffffffffa952dde4 napi_complete_done+0x74 ([kernel.kallsyms])
        ffffffffc187c8ba wg_packet_rx_poll+0x40a ([kernel.kallsyms])
        ffffffffa952df2d __napi_poll+0x2d ([kernel.kallsyms])
        ffffffffa952e5ee net_rx_action+0x20e ([kernel.kallsyms])
        ffffffffa88dfe22 handle_softirqs+0xe2 ([kernel.kallsyms])
        ffffffffa98282f0 __softirqentry_text_start+0x10 ([kernel.kallsyms])
        ffffffffa88dfb9f do_softirq.part.0+0x3f ([kernel.kallsyms])
        ffffffffa88e016e __local_bh_enable_ip+0x5e ([kernel.kallsyms])
        ffffffffa982784d _raw_spin_unlock_bh+0x1d ([kernel.kallsyms])
        ffffffffc187ce57 wg_packet_decrypt_worker+0x57 ([kernel.kallsyms])
        ffffffffa88fccc6 process_one_work+0x176 ([kernel.kallsyms])
        ffffffffa88fe616 worker_thread+0x2c6 ([kernel.kallsyms])
        ffffffffa89065e1 kthread+0xe1 ([kernel.kallsyms])
        ffffffffa88568e6 ret_from_fork+0x36 ([kernel.kallsyms])
        ffffffffa8808611 ret_from_fork_asm+0x11 ([kernel.kallsyms])
```

Receive Path:



Receive Path:



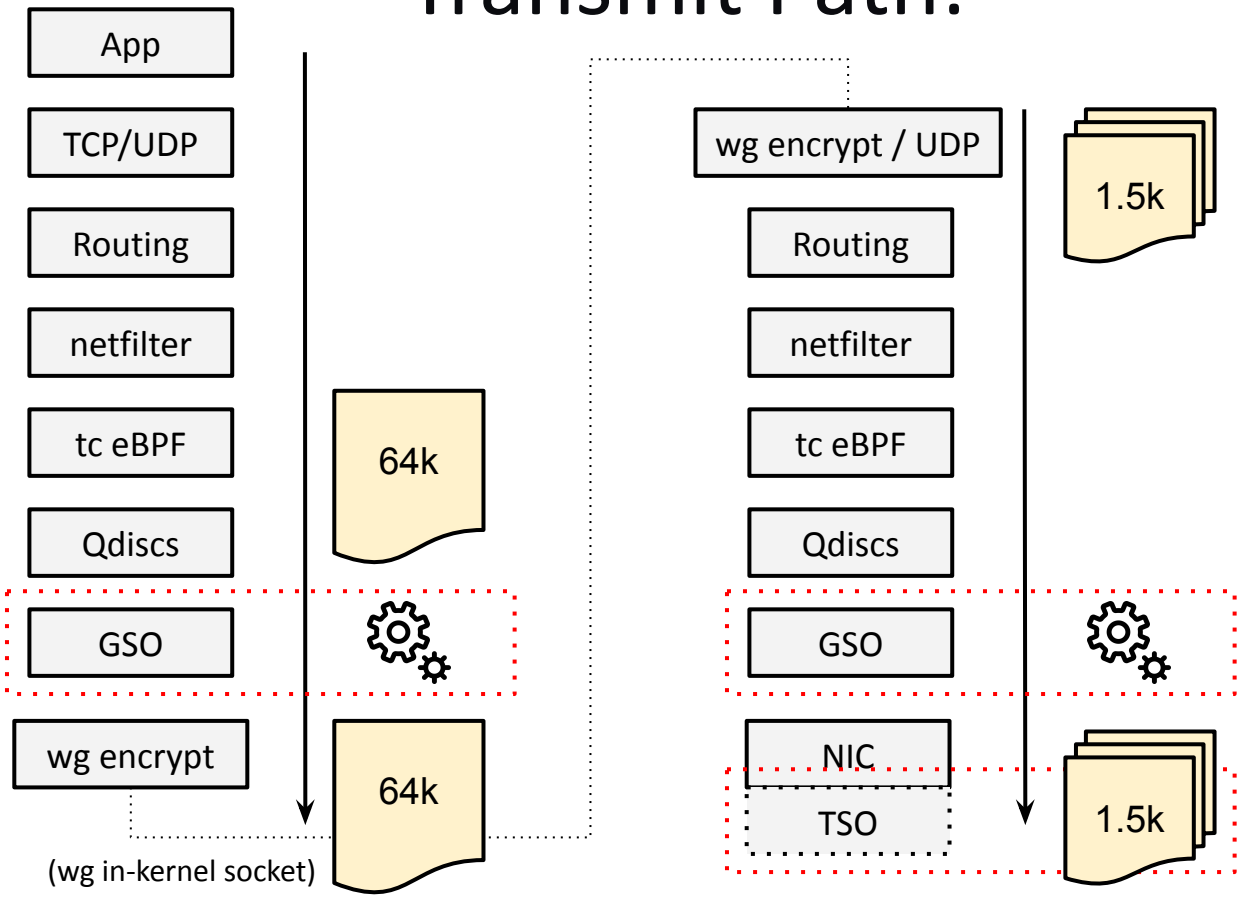
Transmit Path 1/2:

```
netperf   22122 [008]  9985.530527: net:net_dev_start_xmit: dev=wg0 queue_mapping=0 protocol=0x0800 ip_summed=3 len=32292 data_len=32240
        ffffffffb452a17c dev_hard_start_xmit+0x11c ([kernel.kallsyms])
        ffffffffb452a17c dev_hard_start_xmit+0x11c ([kernel.kallsyms])
        ffffffffb452a547 __dev_queue_xmit+0x227 ([kernel.kallsyms])
        ffffffffb4537b67 neigh_connected_output+0xc7 ([kernel.kallsyms])
        ffffffffb45dfc66 ip_finish_output2+0x196 ([kernel.kallsyms])
        ffffffffb45e0bb7 __ip_finish_output+0x87 ([kernel.kallsyms])
        ffffffffb45e0c99 ip_finish_output+0x29 ([kernel.kallsyms])
        ffffffffb45e0dcf ip_output+0x5f ([kernel.kallsyms])
        ffffffffb45e2607 __ip_queue_xmit+0x1a7 ([kernel.kallsyms])
        ffffffffb45e2935 ip_queue_xmit+0x15 ([kernel.kallsyms])
        ffffffffb46080e0 __tcp_transmit_skb+0x540 ([kernel.kallsyms])
        ffffffffb460a2a8 tcp_write_xmit+0x658 ([kernel.kallsyms])
        ffffffffb460b41c tcp_push_one+0x3c ([kernel.kallsyms])
        ffffffffb45f202a tcp_sendmsg_locked+0x31a ([kernel.kallsyms])
        ffffffffb45f2d1c tcp_sendmsg+0x2c ([kernel.kallsyms])
        ffffffffb4636025 inet_sendmsg+0x35 ([kernel.kallsyms])
        ffffffffb44f7de0 __sock_sendmsg+0x40 ([kernel.kallsyms])
        ffffffffb44f9fb9 __sys_sendto+0x109 ([kernel.kallsyms])
        ffffffffb44fa024 __x64_sys_sendto+0x24 ([kernel.kallsyms])
        ffffffffb3808ee7 x64_sys_call+0x8a7 ([kernel.kallsyms])
        ffffffffb4811f1e do_syscall_64+0x7e ([kernel.kallsyms])
        ffffffffb4a000ac entry_SYSCALL_64_after_hwframe+0x4b ([kernel.kallsyms])
            7b831c32bead __send+0x1d (/usr/lib/x86_64-linux-gnu/libc.so.6)
```

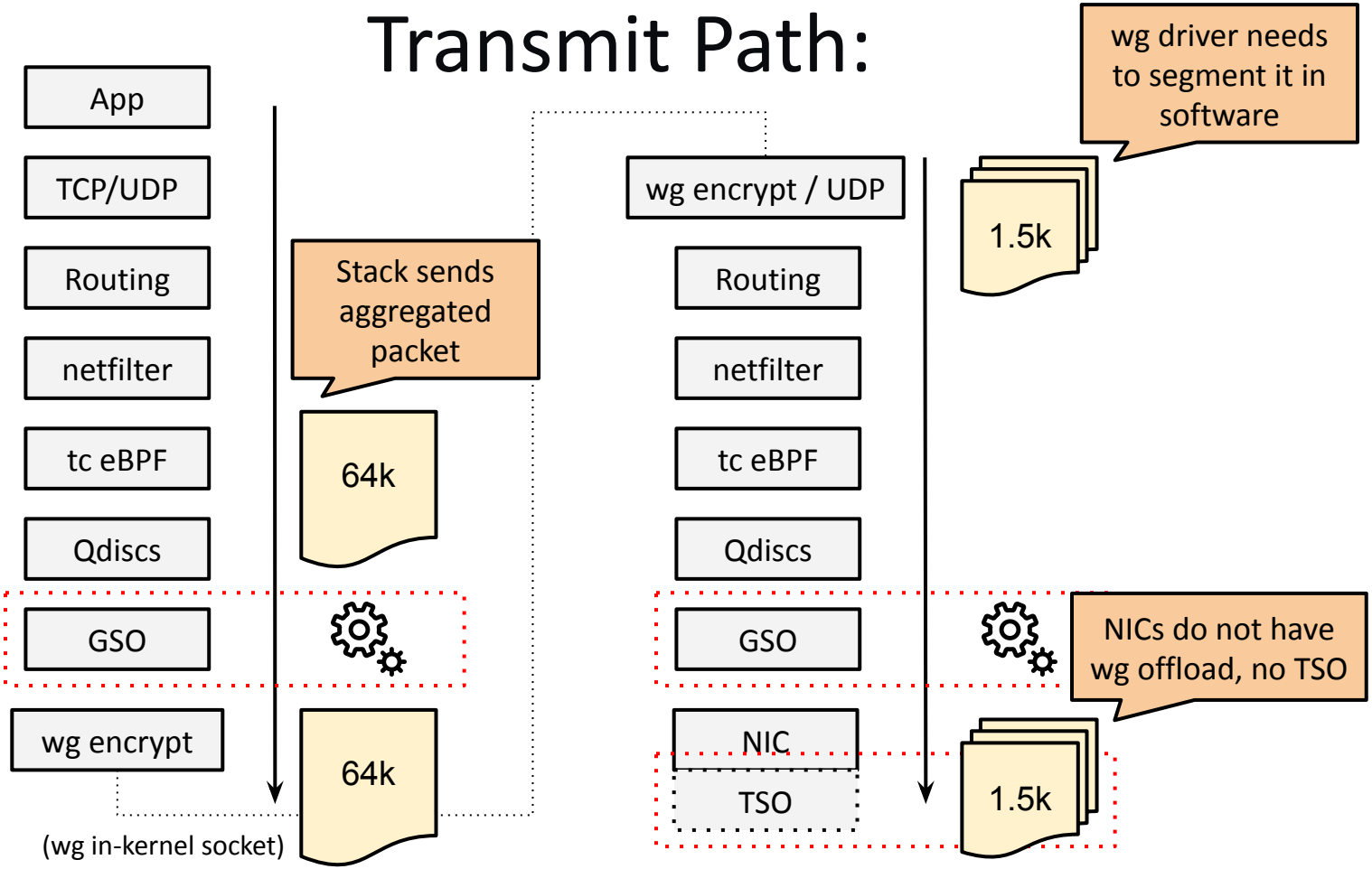
Transmit Path 2/2:

```
kthreadd   22123 [001]  9985.530549: net:net_dev_start_xmit: dev=enp10s0f0np0 protocol=0x0800 ip_summed=3 len=8186 data_len=0
        ffffffffb452a17c dev_hard_start_xmit+0x11c ([kernel.kallsyms])
        ffffffffb452a17c dev_hard_start_xmit+0x11c ([kernel.kallsyms])
        ffffffffb4598083 sch_direct_xmit+0xf3 ([kernel.kallsyms])
        ffffffffb452af89 __dev_queue_xmit+0xc69 ([kernel.kallsyms])
        ffffffffb453c00f neigh_resolve_output+0x10f ([kernel.kallsyms])
        ffffffffb45dfc66 ip_finish_output2+0x196 ([kernel.kallsyms])
        ffffffffb45e0bb7 __ip_finish_output+0x87 ([kernel.kallsyms])
        ffffffffb45e0c99 ip_finish_output+0x29 ([kernel.kallsyms])
        ffffffffb45e0dcf ip_output+0x5f ([kernel.kallsyms])
        ffffffffb45e2235 ip_local_out+0x35 ([kernel.kallsyms])
        ffffffffb464cbf5 iptunnel_xmit+0x185 ([kernel.kallsyms])
        ffffffffc16262f9 udp_tunnel_xmit_skb+0xe9 ([kernel.kallsyms])
        ffffffffc16b089f send4+0x11f ([kernel.kallsyms])
        ffffffffc16b0ea0 wg_socket_send_skb_to_peer+0xa0 ([kernel.kallsyms])
        ffffffffc16addd4 wg_packet_tx_worker+0xa4 ([kernel.kallsyms])
        ffffffffb38fccc6 process_one_work+0x176 ([kernel.kallsyms])
        ffffffffb38fe616 worker_thread+0x2c6 ([kernel.kallsyms])
        ffffffffb39065e1 kthread+0xe1 ([kernel.kallsyms])
        ffffffffb38568e6 ret_from_fork+0x36 ([kernel.kallsyms])
        ffffffffb3808611 ret_from_fork_asm+0x11 ([kernel.kallsyms])
```

Transmit Path:



Transmit Path:



On transmit, the driver internally does the segmentation. This is needed for replay protection in the wg header (counter).

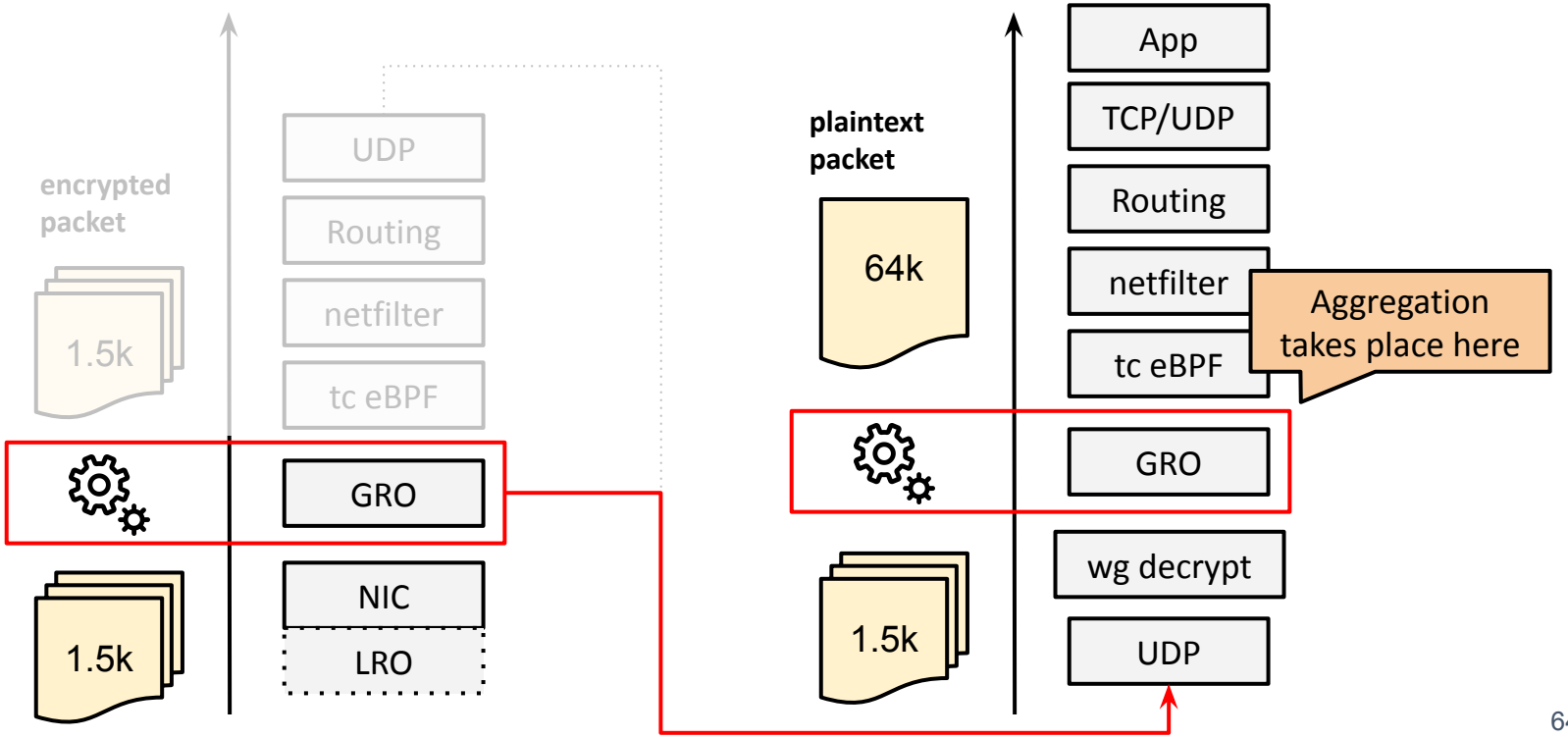
```
> Ethernet II, Src: a2:e3:ac:4a:b5:39 (a2:e3:ac:4a:b5:39),
> Internet Protocol Version 4, Src: 10.0.4.1 (10.0.4.1), D
> User Datagram Protocol, Src Port: 51820, Dst Port: 47278
v WireGuard
    Type: Transport Data (4)
    Reserved
    Receiver: 0x768bd016
    Counter: 0
    v Packet (encrypted)
        Ciphertext: ad6bb0478e6afd812becc5946c2d5cbd4bc19f0938796f0d...
        Auth Tag: 77eb5c70e898254f83591ad30d32fba8
        (authentication tag verified)
> Internet Protocol Version 4, Src: 10.0.5.1 (10.0.5.1), Dst: 10.0.5.2 (10.0.5.2)
> Internet Control Message Protocol
```

type $:= 0x4$ (1 byte)	reserved $:= 0^3$ (3 bytes)
receiver $:= I_{m'}$ (4 bytes)	
counter (8 bytes)	
packet ($\widehat{\lVert P \rVert}$ bytes)	

Image source: https://www.wireguard.com/papers/wireguard.pdf

Unless this can be offloaded at some point in future, this is here to stay. But, what about optimizing receive side?

Receive Path, enter L2 GRO:



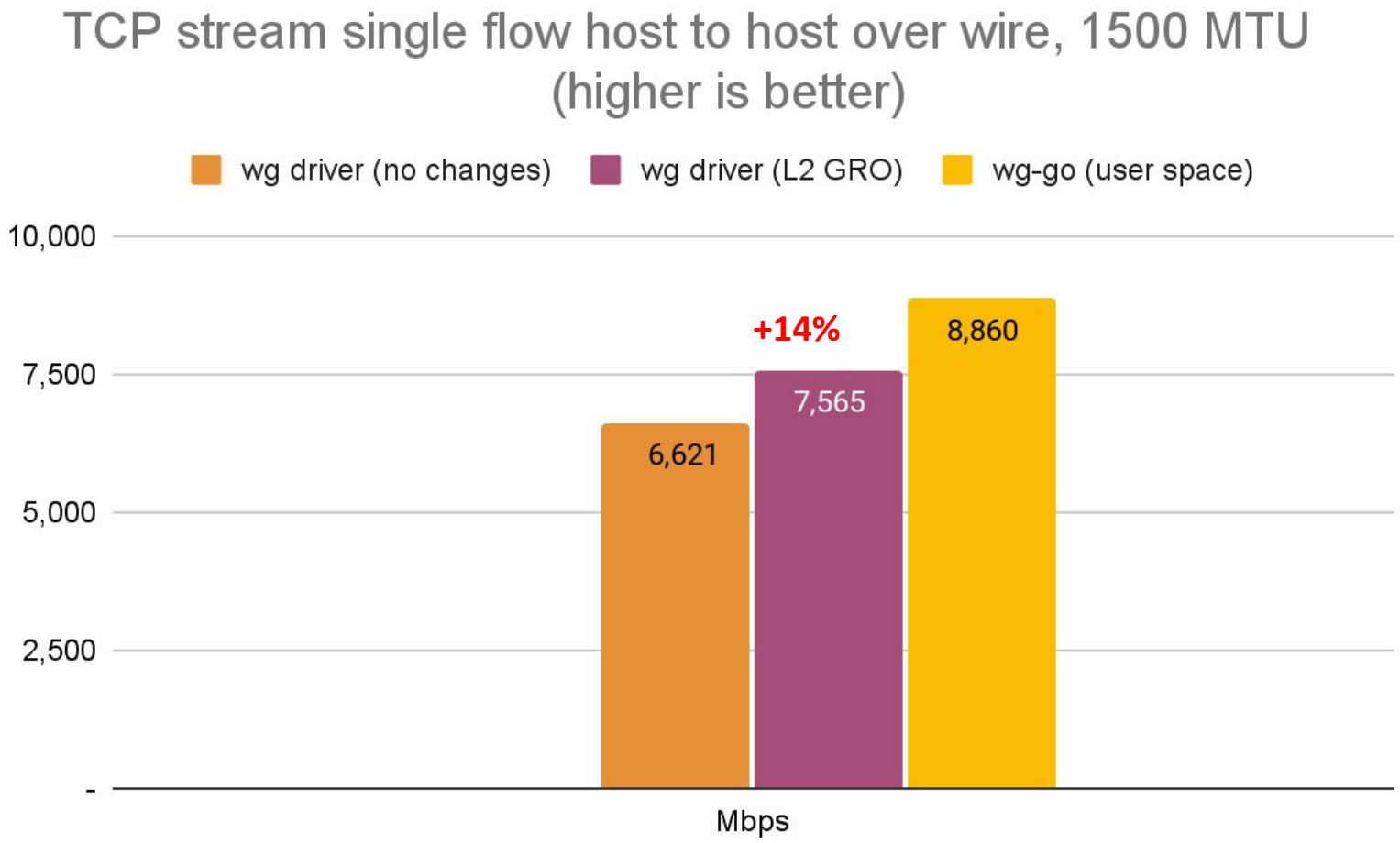
```
static size_t wg_gro_candidate(struct sk_buff *skb)
{
        if (unlikely(skb->len < sizeof(struct message_header)))
                return false;
        if (SKB_TYPE_LE32(skb) == cpu_to_le32(MESSAGE_DATA) &&
            skb->len >= MESSAGE_MINIMUM_LENGTH)
                return true;
        return false;
}

struct sk_buff *wg_gro_receive(struct sock *sk,
                               struct list_head *head,
                               struct sk_buff *skb)
{
        struct wg_device *wg = sk->sk_user_data;
        int offset = skb_gro_offset(skb);

        if (!pskb_pull(skb, offset))
                return NULL;
        if (!wg_gro_candidate(skb))
                goto out;
        skb_mark_not_on_list(skb);
        PACKET_CB(skb)->ds = ip_tunnel_get_dsfield(ip_hdr(skb), skb);
        wg_packet_consume_data(wg, skb);
        return ERR_PTR(-EINPROGRESS);
out:
        skb_push(skb, offset);
        NAPI_GRO_CB(skb)->same_flow = 0;
        NAPI_GRO_CB(skb)->flush = 1;
        return NULL;
}
```

```
int wg_socket_init(struct wg_device *wg, u16 port)
{
        struct net *net;
        int ret;
        struct udp_tunnel_sock_cfg cfg = {
                .sk_user_data   = wg,
                .encap_type     = 1,
                .encap_rcv      = wg_receive,
                .gro_receive    = wg_gro_receive,
        };
...
```

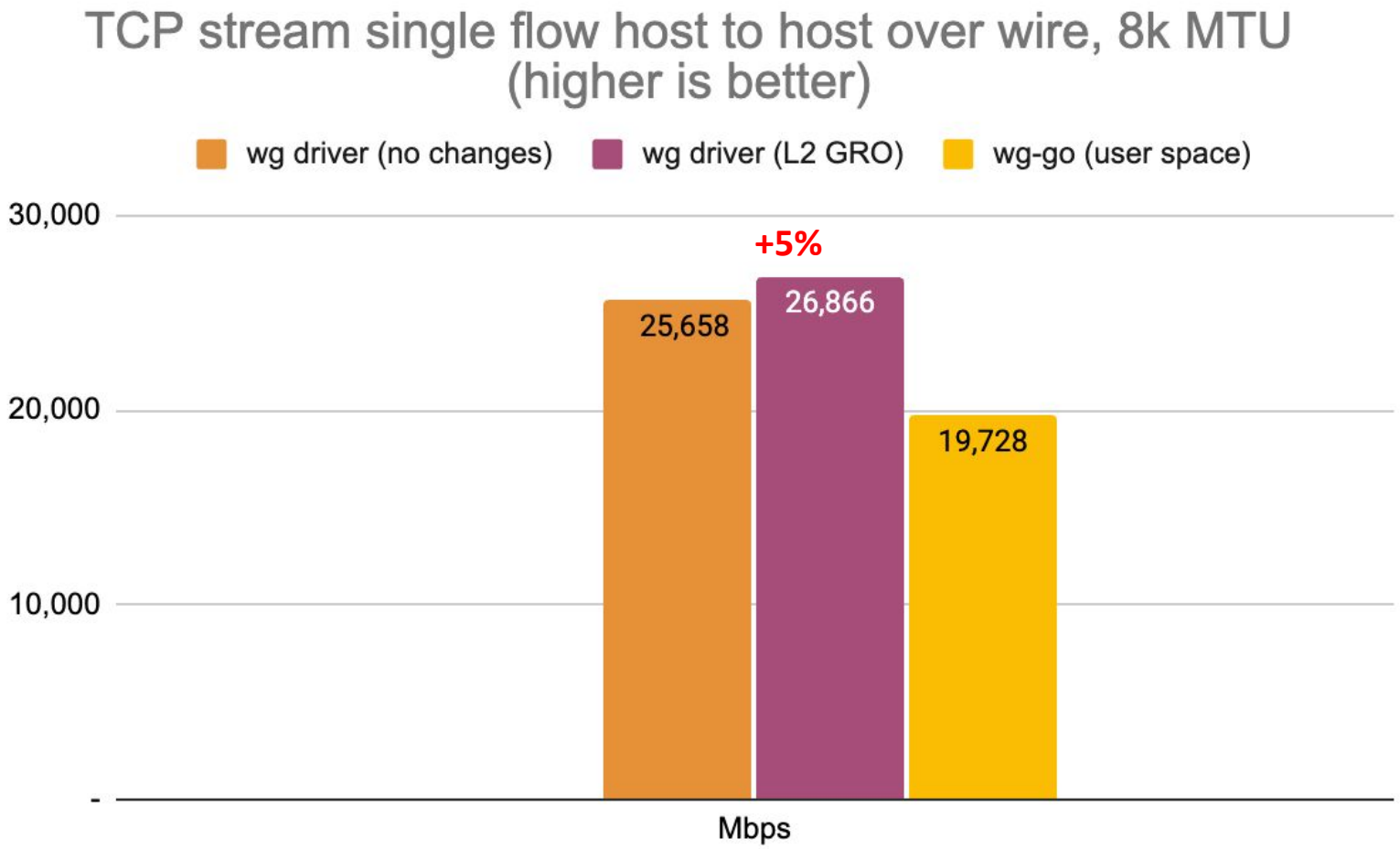
UDP socket registers GRO handler

GRO handler pushes data packets directly for decryption when GRO engine is invoked from physical dev

IPSec ESP GRO (INET_ESP_OFFLOAD) operates similarly

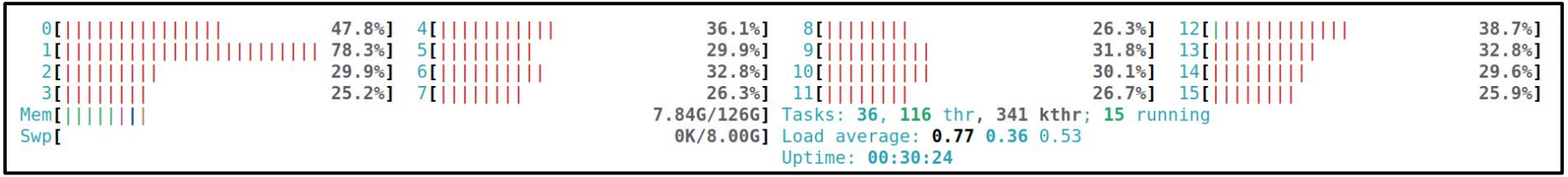

TCP stream single flow host to host over wire, 1500 MTU
(higher is better)
wg driver (no changes)
wg driver (L2 GRO)
wg-go (user space)
10,000
7,500
5,000
2,500
-
+14%
6,621
7,565
8,860
Mbps

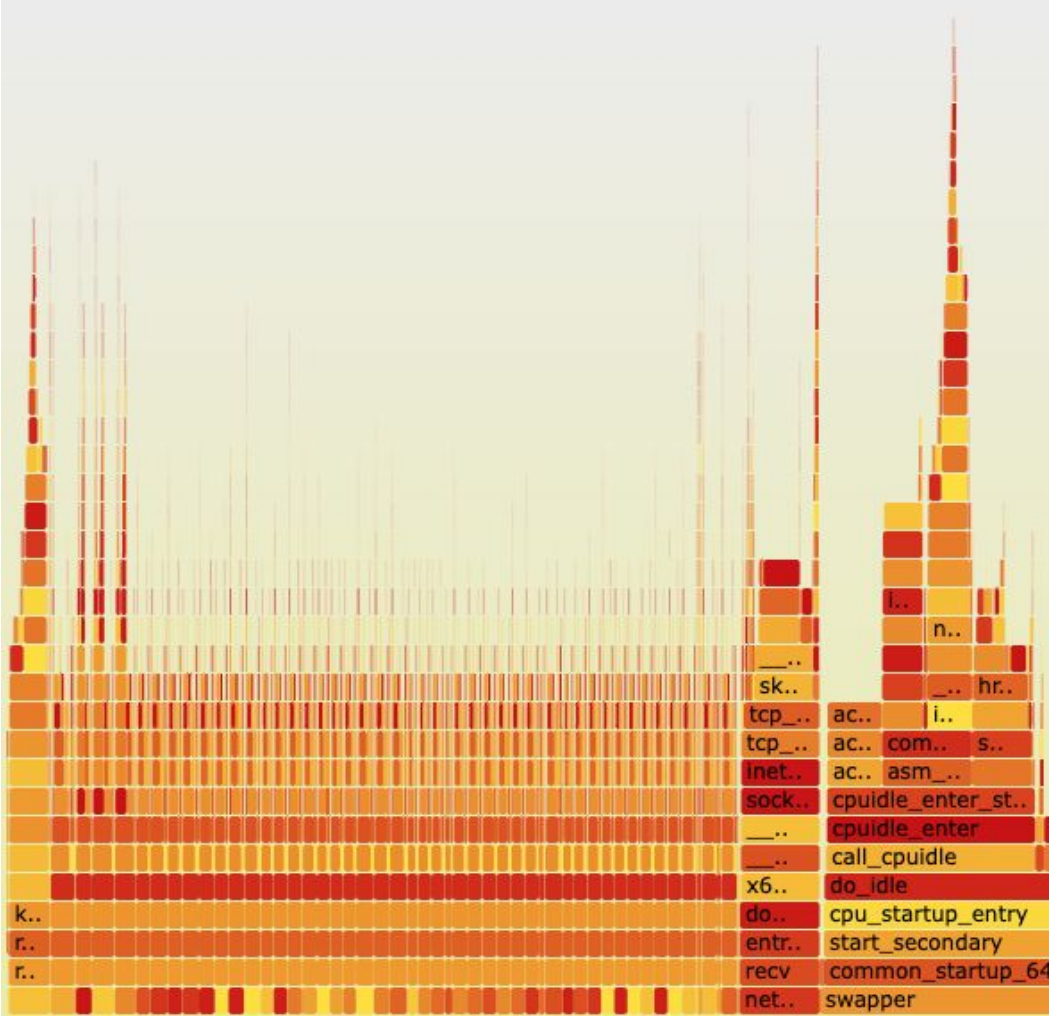
The 14% improvement becomes less visible with 8k MTU since packets are batched already, and most time is spent in decryption.


TCP stream single flow host to host over wire, 8k MTU
(higher is better)
wg driver (no changes)
wg driver (L2 GRO)
wg-go (user space)
30,000
20,000
10,000
-
+5%
25,658
26,866
19,728
Mbps

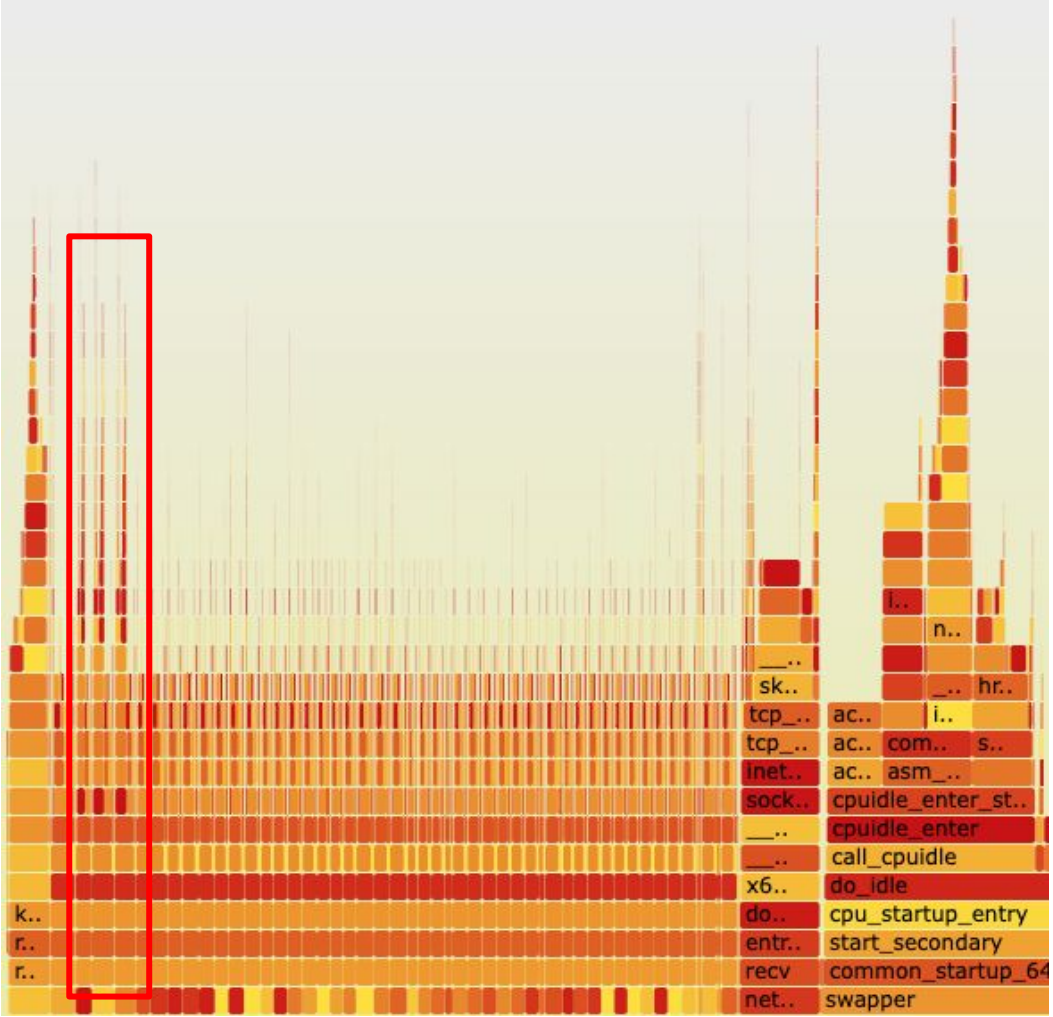
For a single stream, wg spreads the encryption and decryption across all CPUs through kernel worker threads.

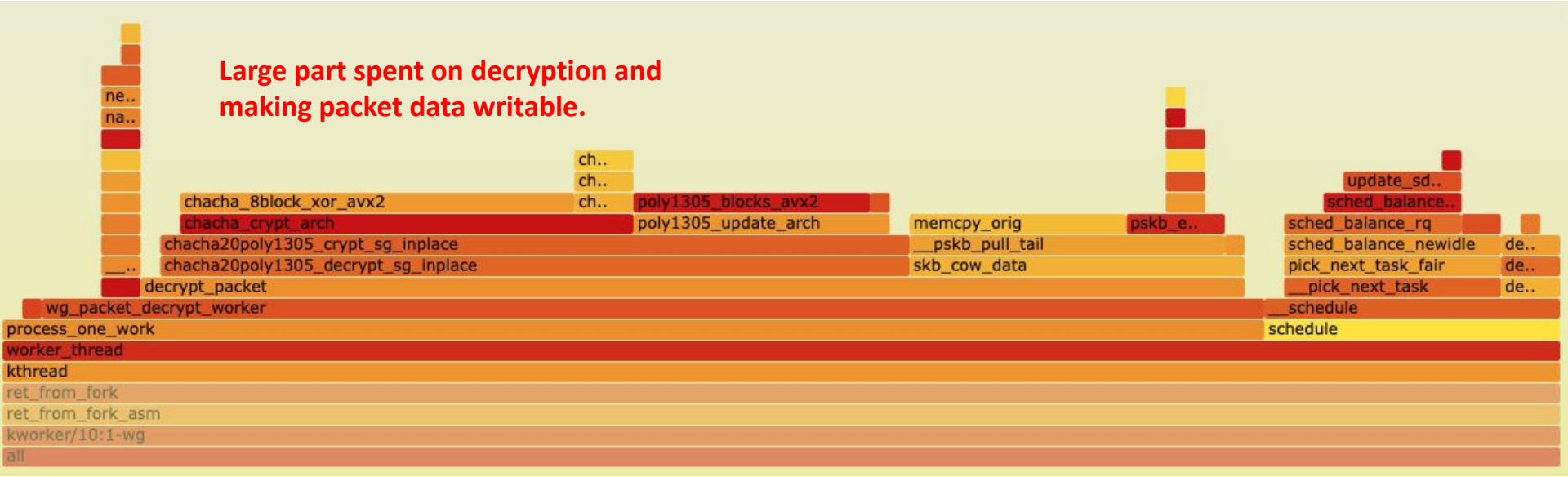
For a single stream, wg spreads the encryption and decryption across all CPUs through kernel worker threads.

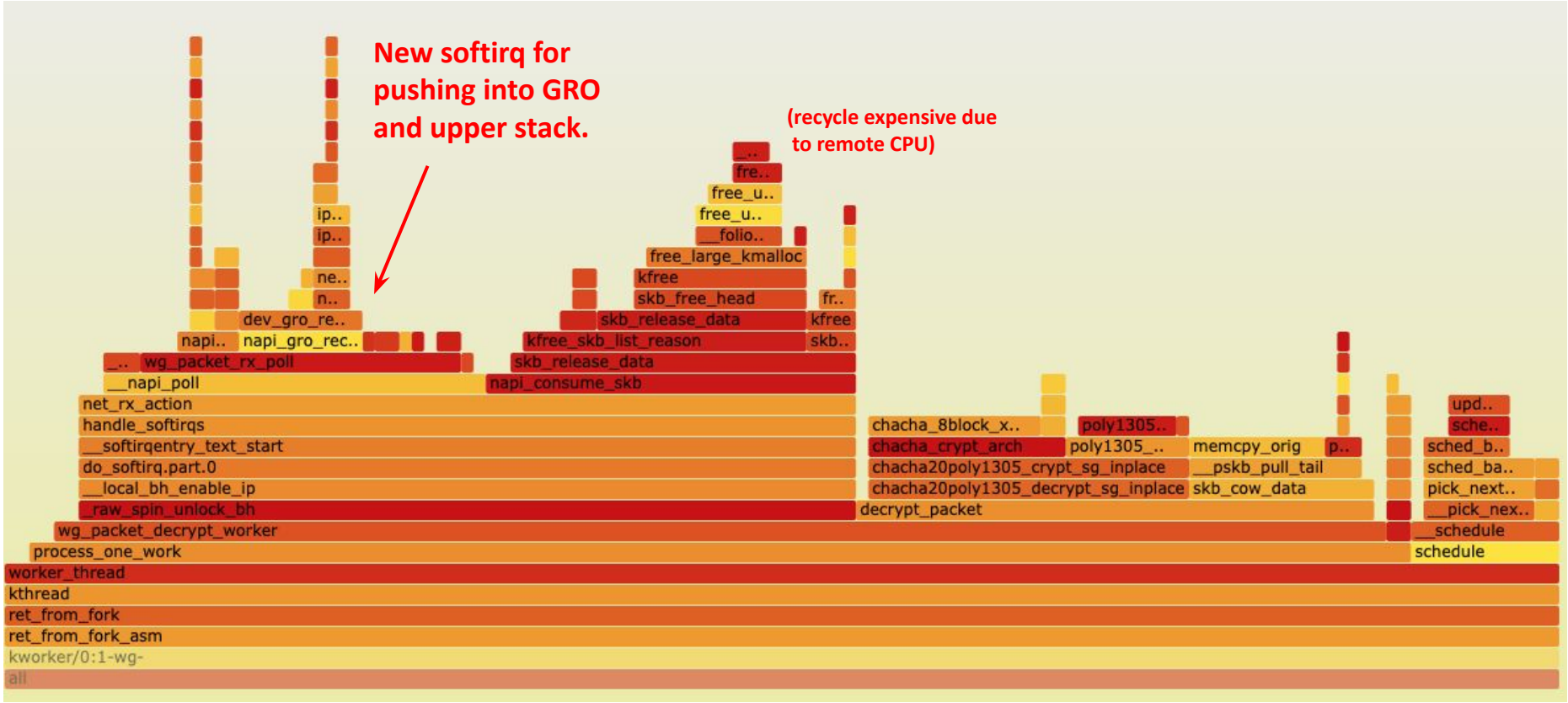



i..
n..
__..
sk..
_..
hr..
tcp_..
ac..
i..
tcp_..
ac..
com..
s..
inet..
ac..
asm_..
sock..
cpuidle_enter_st..
__..
cpuidle_enter
__..
call_cpuidle
x6..
do_idle
k..
do..
cpu_startup_entry
r..
entr..
start_secondary
r..
recv
common_startup_64
net..
swapper

Taking a look at the individual kthreads ...




Large part spent on decryption and making packet data writable.
ne..
na..
ch..
ch..
ch..
chacha_8block_xor_avx2
chacha_crypt_arch
poly1305_blocks_avx2
poly1305_update_arch
memcpy_orig
pskb_e..
update_sd..
sched_balance..
sched_balance_rq
chacha20poly1305_crypt_sg_inplace
__pskb_pull_tail
sched_balance_newidle
de..
__..
chacha20poly1305_decrypt_sg_inplace
skb_cow_data
pick_next_task_fair
de..
decrypt_packet
__pick_next_task
de..
wg_packet_decrypt_worker
__schedule
process_one_work
schedule
worker_thread
kthread
ret_from_fork
ret_from_fork_asm
kworker/10:1-wg
all


New softirq for pushing into GRO and upper stack.
(recycle expensive due to remote CPU)
_..
fre..
free_u..
free_u..
__folio..
free_large_kmalloc
kfree
skb_free_head
fr..
skb_release_data
kfree
kfree_skb_list_reason
skb..
skb_release_data
napi_consume_skb
ip..
ip..
ne..
n..
dev_gro_re..
napi..
napi_gro_rec..
_..
wg_packet_rx_poll
__napi_poll
net_rx_action
handle_softirqs
__softirqentry_text_start
do_softirq.part.0
__local_bh_enable_ip
_raw_spin_unlock_bh
wg_packet_decrypt_worker
process_one_work
worker_thread
kthread
ret_from_fork
ret_from_fork_asm
kworker/0:1-wg-
all
chacha_8block_x..
poly1305..
chacha_crypt_arch
poly1305_..
memcpy_orig
p..
chacha20poly1305_crypt_sg_inplace
__pskb_pull_tail
chacha20poly1305_decrypt_sg_inplace
skb_cow_data
decrypt_packet
upd..
sche..
sched_b..
sched_ba..
pick_next..
__pick_nex..
__schedule
schedule

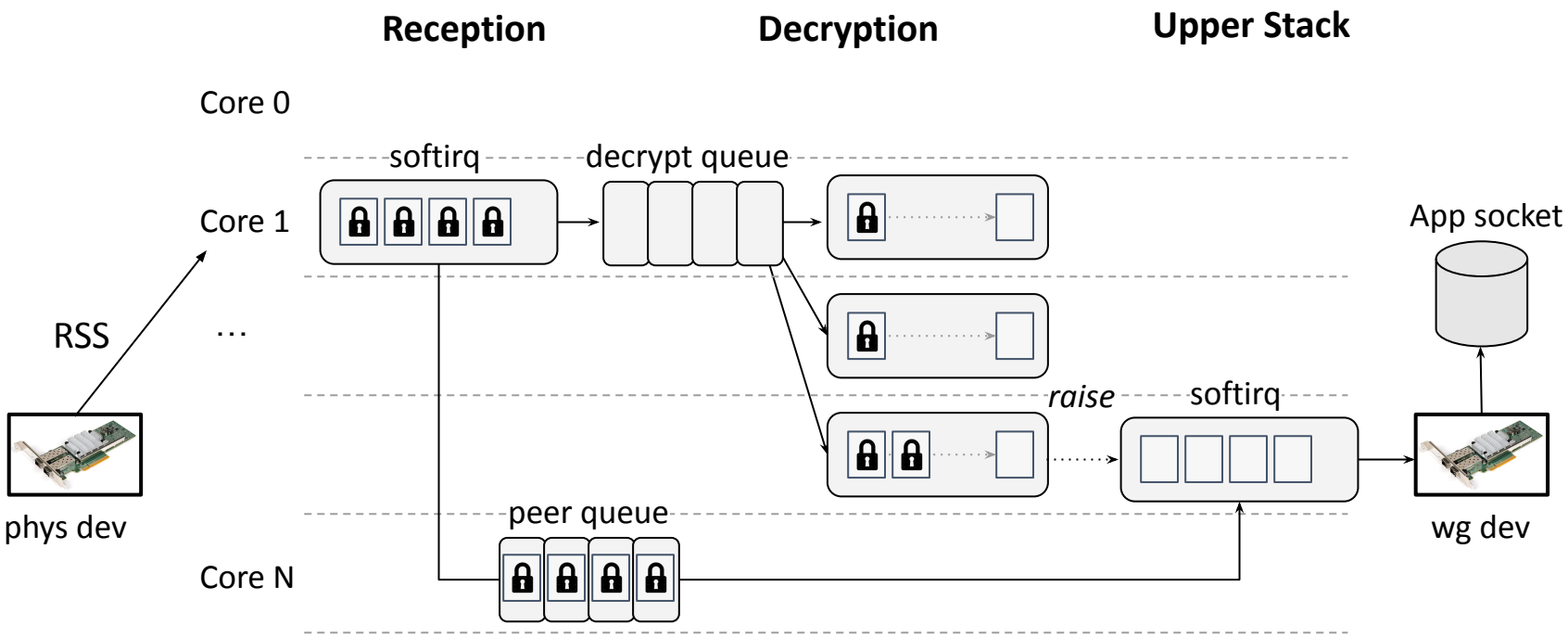


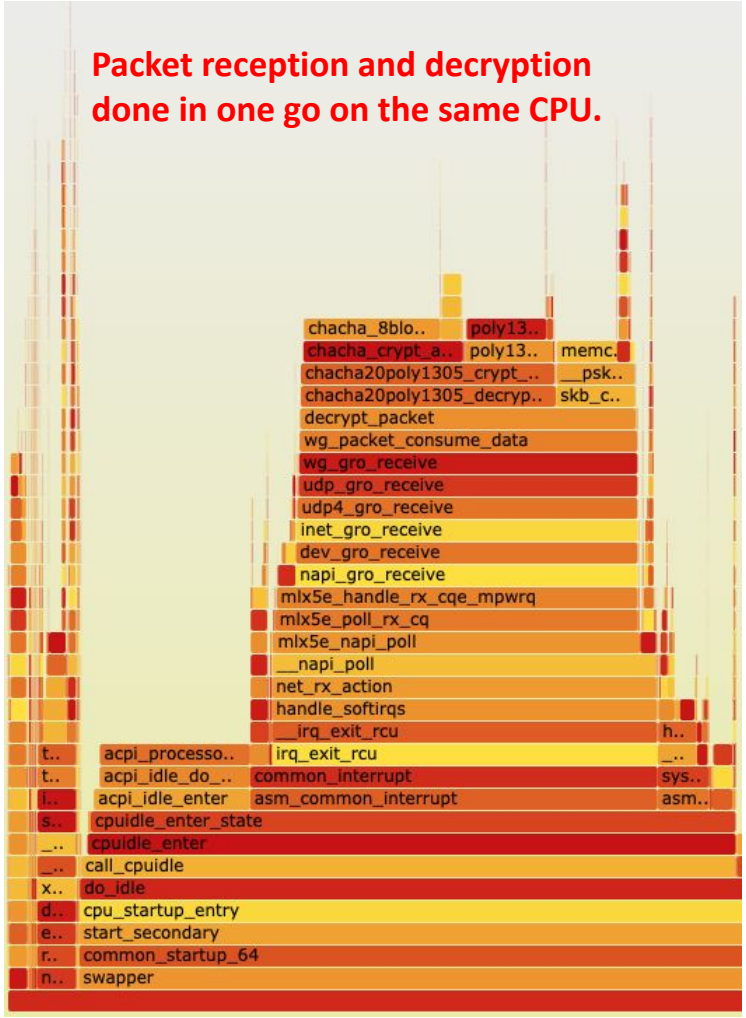
Image source: https://netdevconf.org/0x18/docs/netdev-0x18-paper23-talk-paper.pdf

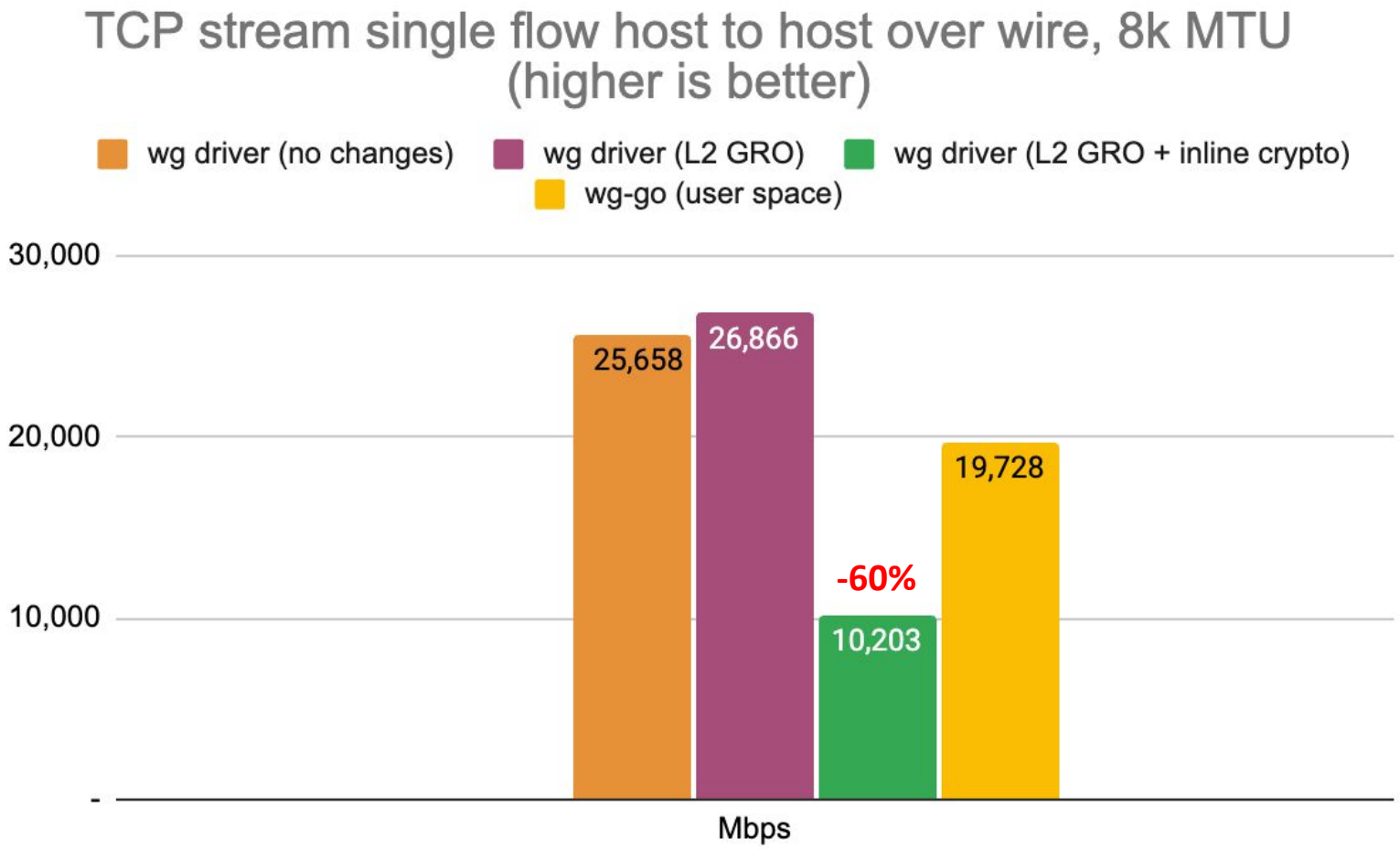
Spreading across all CPUs helps bulk workloads, but comes at the cost of thrashing cache-locality. What about inline crypto?

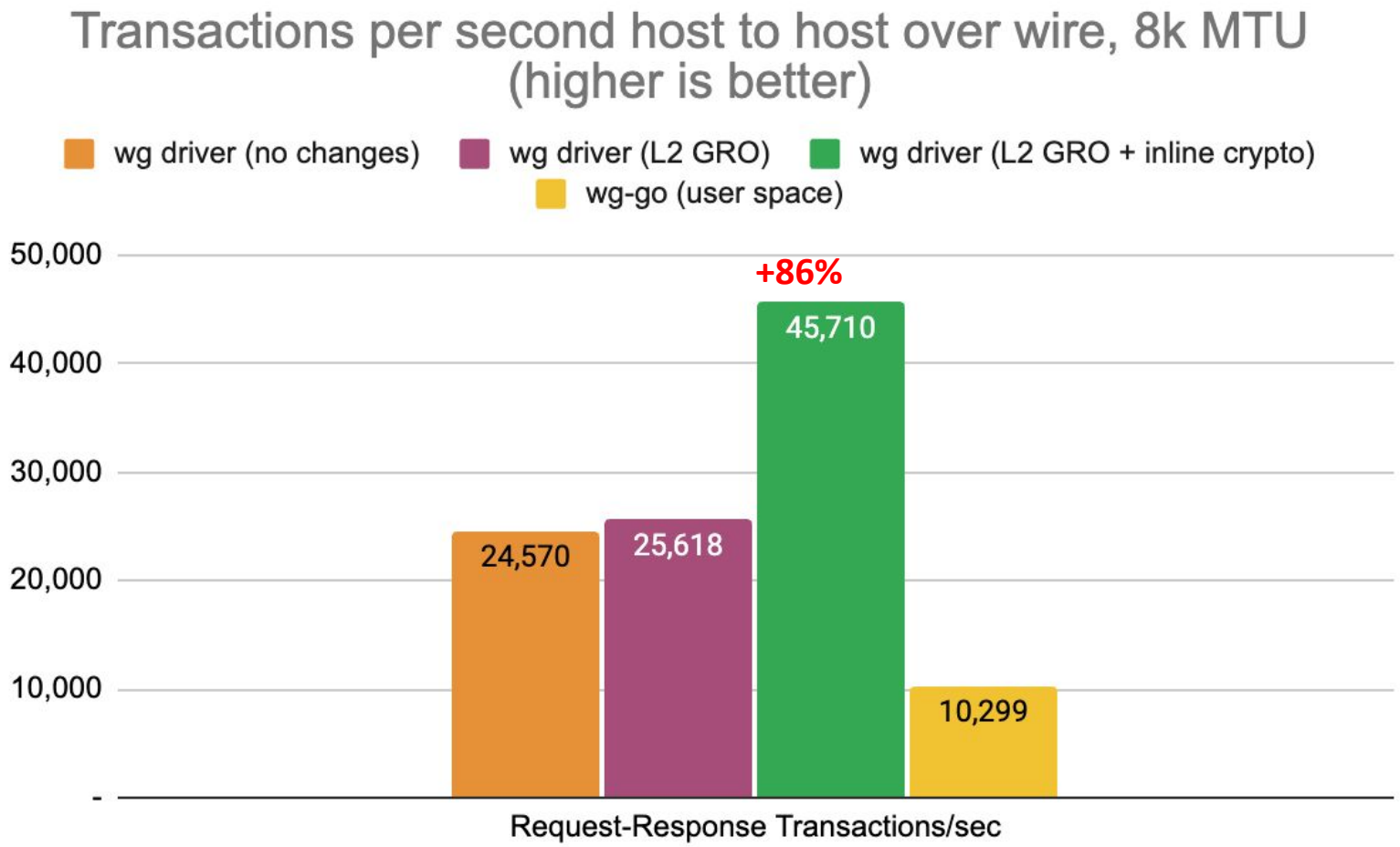
Spreading across all CPUs helps bulk workloads, but comes at the cost of thrashing cache-locality. What about inline crypto?

```
  0[||                        5.3%]  4[                        0.0%]   8[                        0.0%]  12[                        0.0%]
  1[                          0.0%]  5[                        0.0%]   9[||                      1.3%]  13[||                      6.0%]
  2[                          0.0%]  6[                        0.0%]  10[                        0.0%]  14[|||||||||||||||||||||||||96.1%]
  3[                          0.0%]  7[                        0.0%]  11[                        0.0%]  15[                        0.0%]
Mem[||||                                                 1.08G/126G] Tasks: 36, 119 thr, 322 kthr; 1 running
Swp[                                                       0K/8.00G] Load average: 0.09 0.05 0.01
                                                                     Uptime: 01:38:21
```

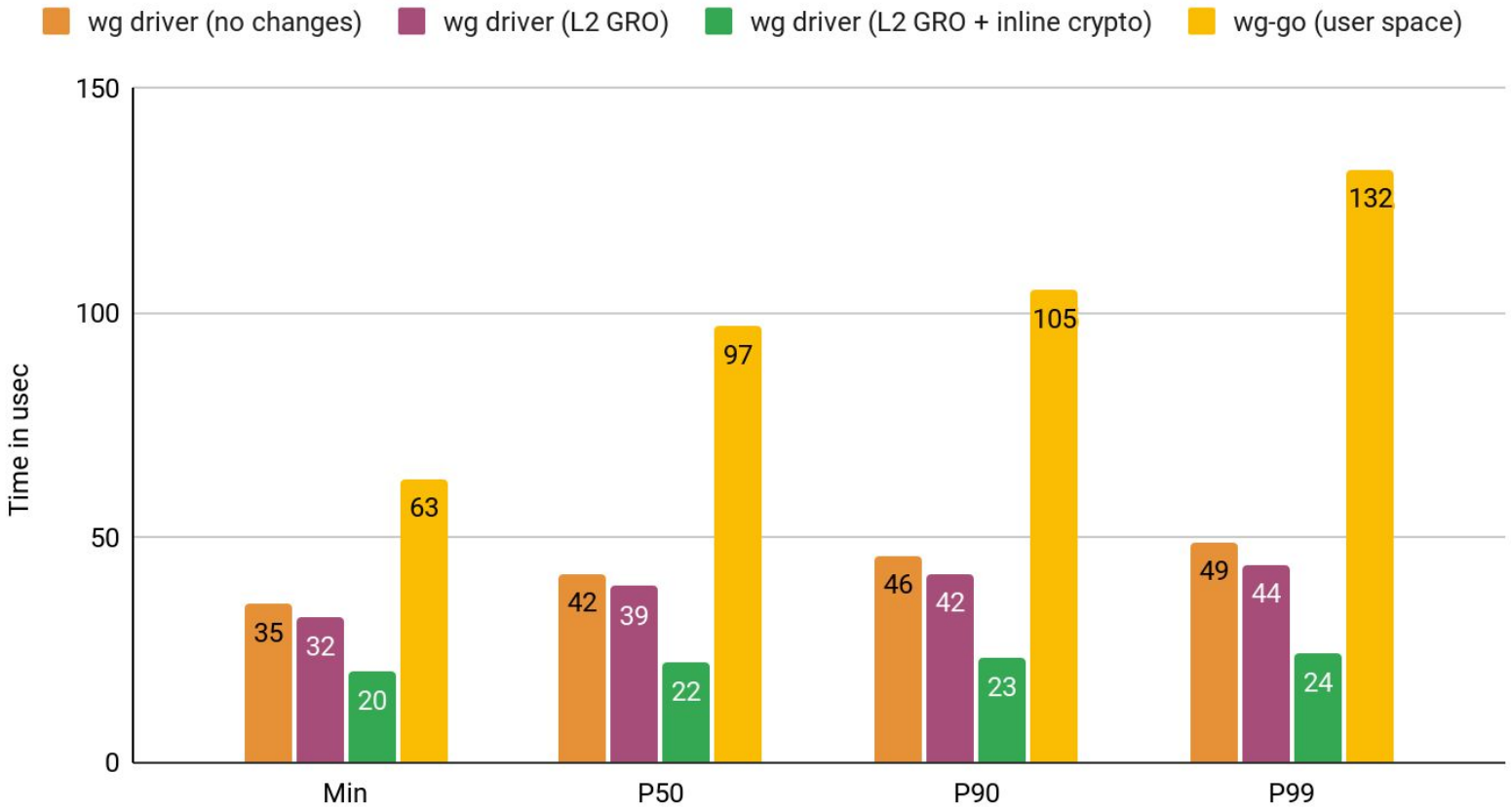
Packet reception and decryption done in one go on the same CPU.




TCP stream single flow host to host over wire, 8k MTU
(higher is better)
wg driver (no changes)
wg driver (L2 GRO)
wg driver (L2 GRO + inline crypto)
wg-go (user space)
30,000
20,000
10,000
-
25,658
26,866
10,203
19,728
-60%
Mbps


Transactions per second host to host over wire, 8k MTU
(higher is better)
wg driver (no changes)
wg driver (L2 GRO)
wg driver (L2 GRO + inline crypto)
wg-go (user space)
50,000
40,000
30,000
20,000
10,000
-
+86%
45,710
24,570
25,618
10,299
Request-Response Transactions/sec

Single flow latency host to host over wire (lower is better)



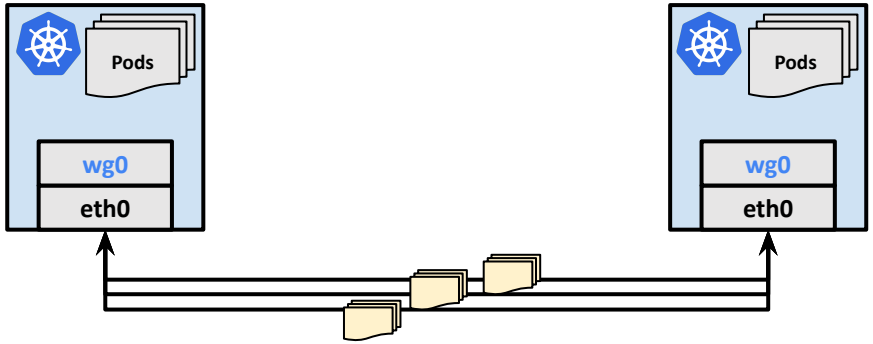
req/resp-type workloads
stream-type workloads
WIREGUARD

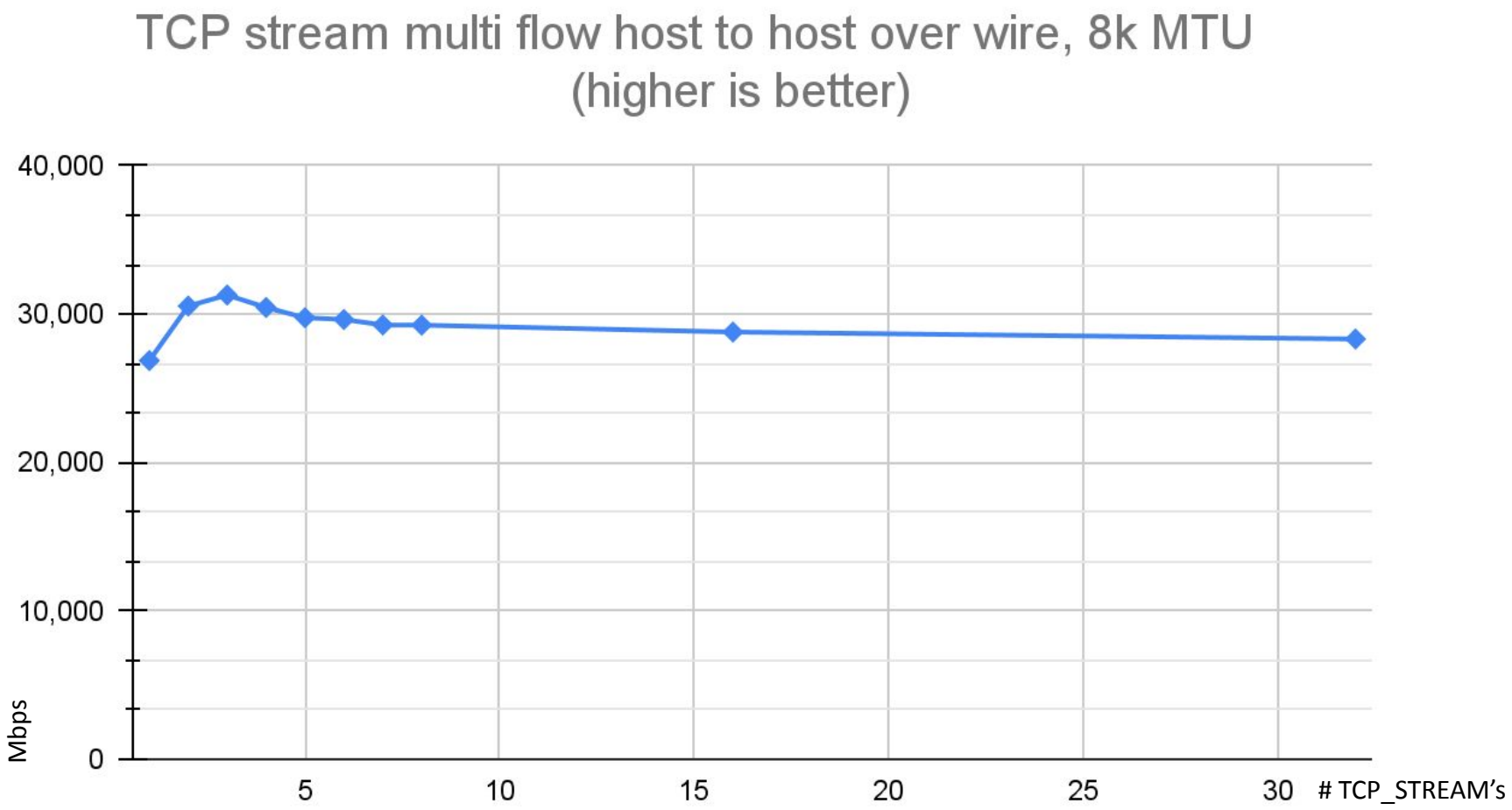
For single stream workloads, while some aspects such as L2 GRO help to optimize, there is generally no free lunch.

Trading off either bulk or request-response type workloads depending on the use-case.

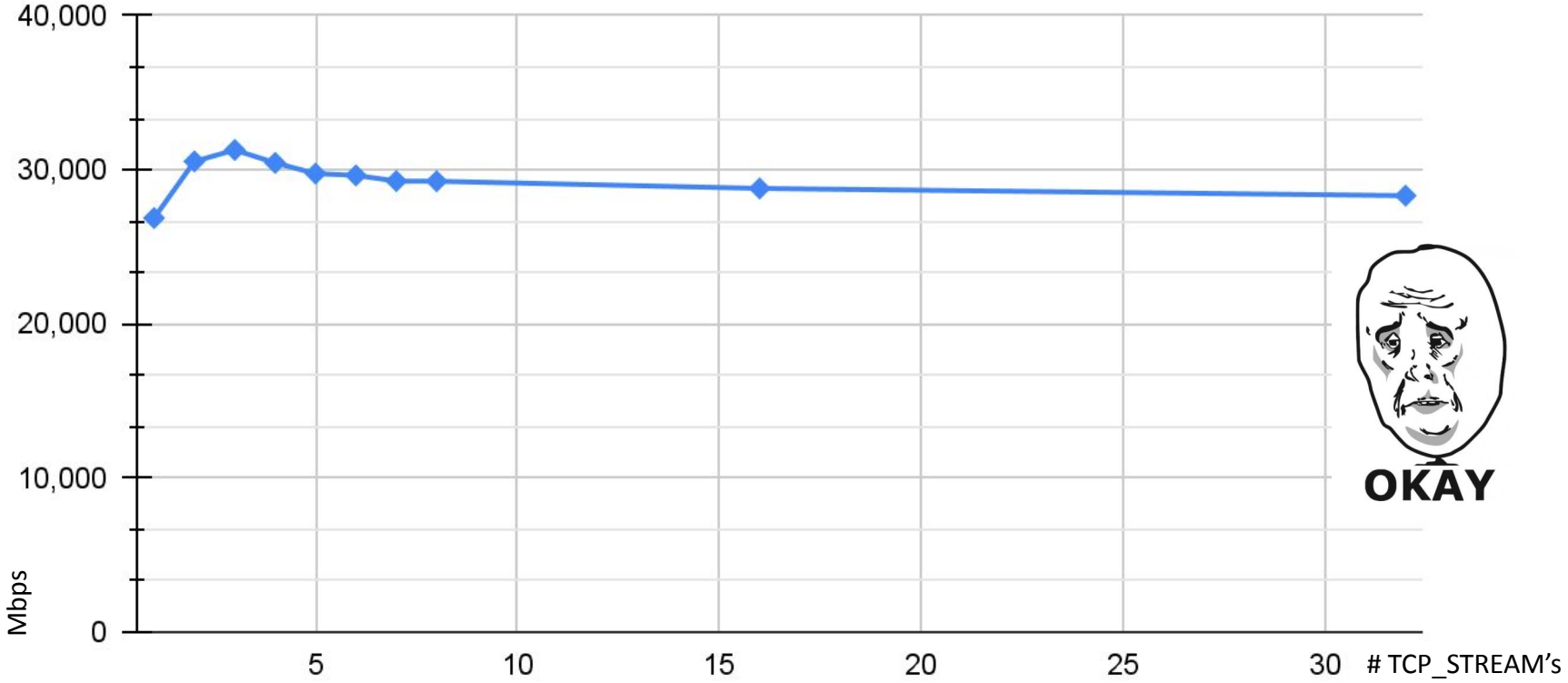
Looking beyond single flow might also provide a better overall picture ...

Step 3: Can we do better for multiple streams?



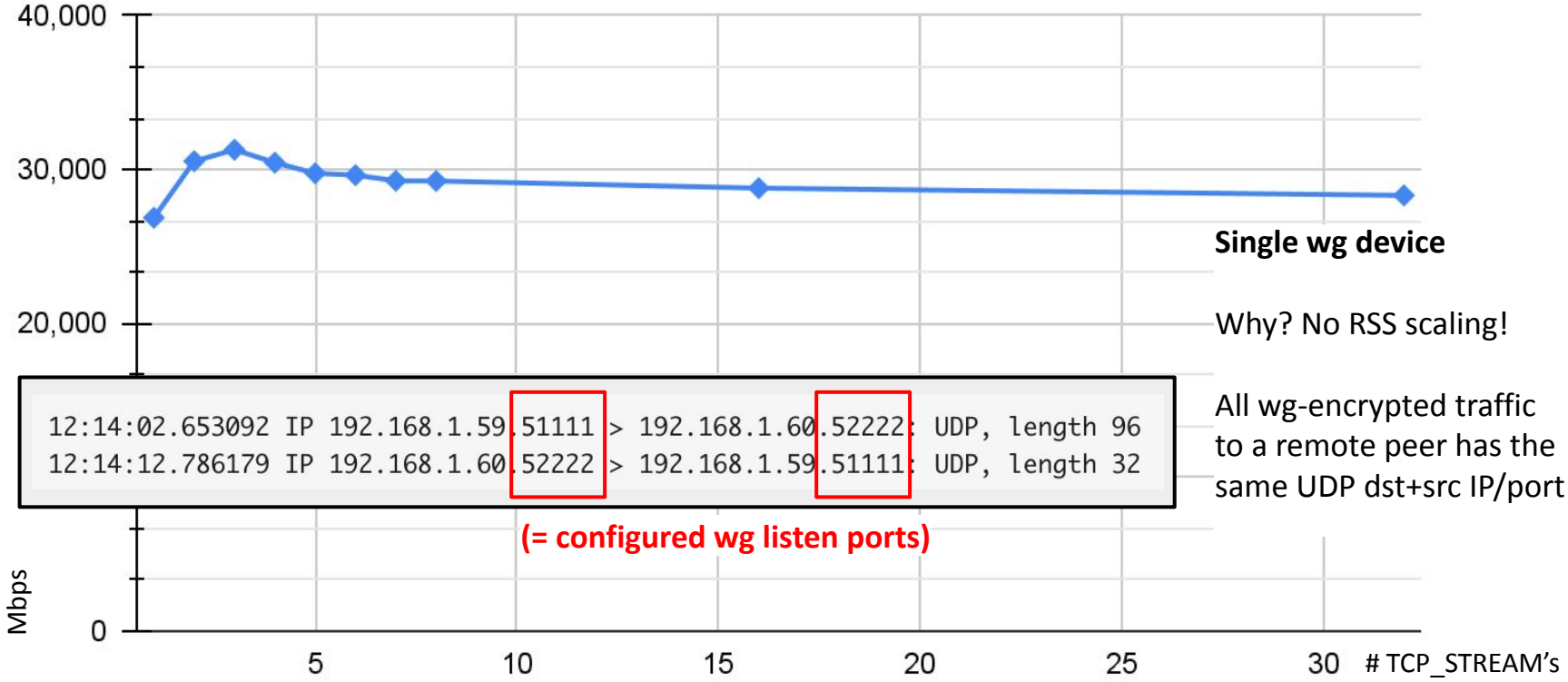

TCP stream multi flow host to host over wire, 8k MTU
(higher is better)
40,000
30,000
20,000
10,000
0
Mbps
5
10
15
20
25
30
TCP_STREAM's

TCP stream multi flow host to host over wire, 8k MTU (higher is better)



How come that the E/W multi-flow scenario does not scale?

TCP stream multi flow host to host over wire, 8k MTU (higher is better)




```
12:14:02.653092 IP 192.168.1.59.51111 > 192.168.1.60.52222: UDP, length 96
12:14:12.786179 IP 192.168.1.60.52222 > 192.168.1.59.51111: UDP, length 32
```

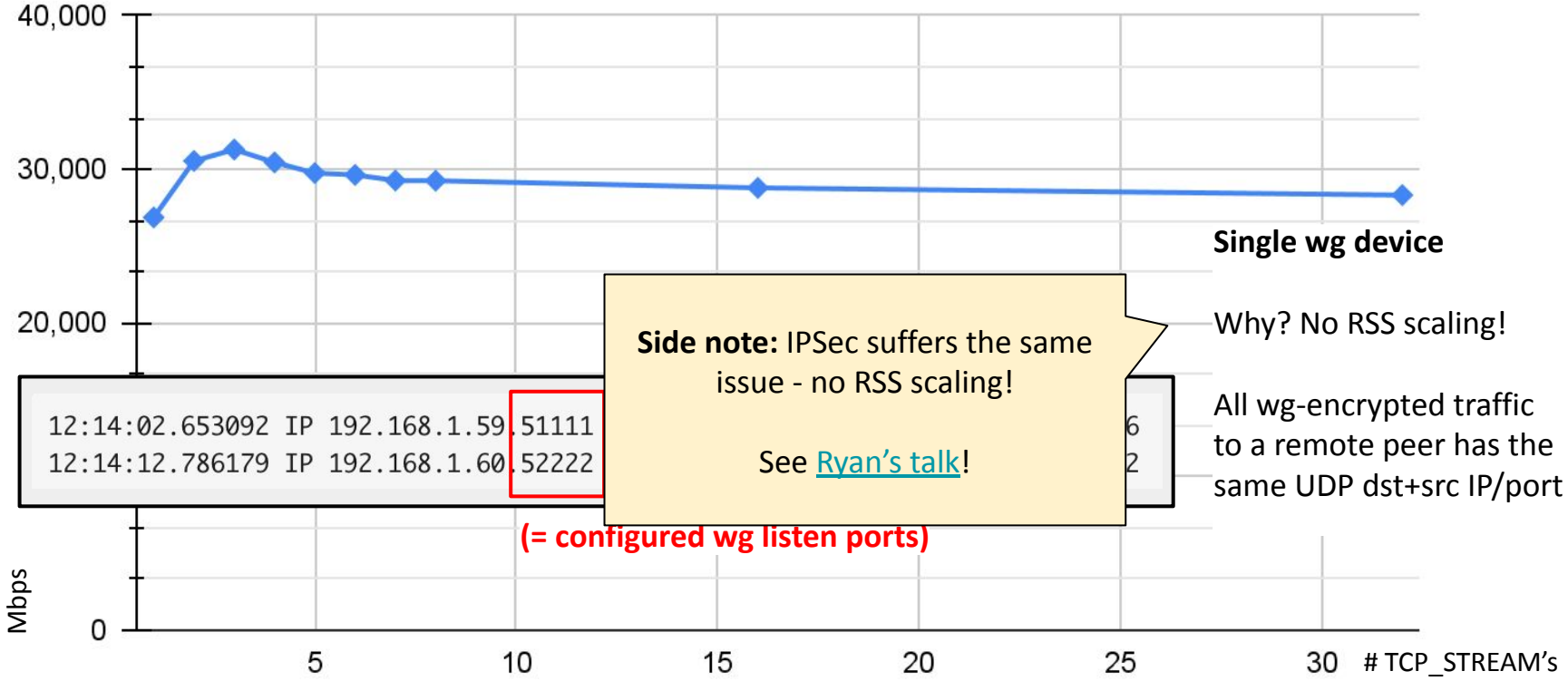
(= configured wg listen ports)

Single wg device

Why? No RSS scaling!

All wg-encrypted traffic to a remote peer has the same UDP dst+src IP/port

TCP stream multi flow host to host over wire, 8k MTU (higher is better)




```
12:14:02.653092 IP 192.168.1.59.51111
12:14:12.786179 IP 192.168.1.60.52222
```

(= configured wg listen ports)

Side note: IPSec suffers the same issue - no RSS scaling!

See Ryan's talk!

Single wg device

Why? No RSS scaling!

All wg-encrypted traffic to a remote peer has the same UDP dst+src IP/port

Naive approach: What about multiple wg devices with different listener ports and load balancing traffic among them?

Options 1/2:

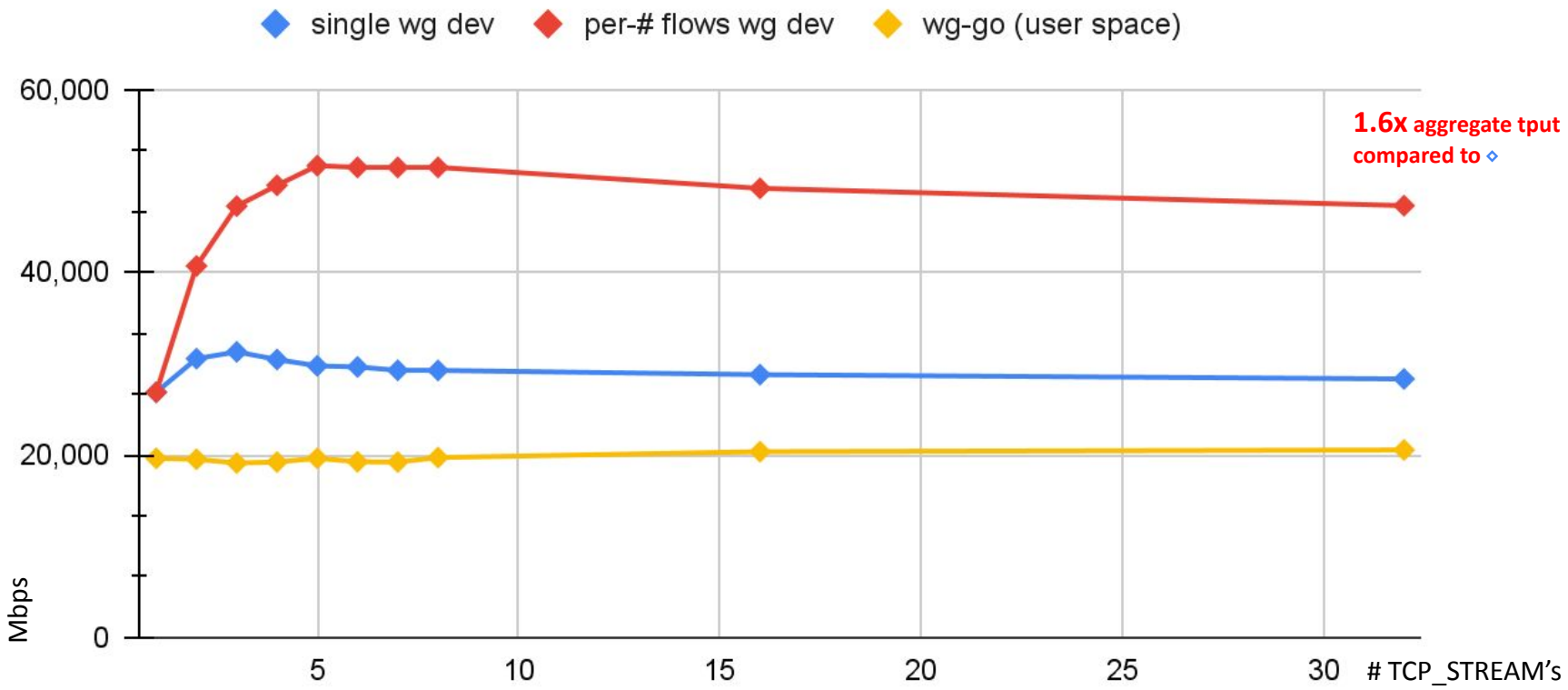
- Multiple WireGuard devices under a bond? L3 vs L2 device, currently not possible
- Multiple WireGuard devices and load-balance via multipath next hops?

```
# ip r
default via 192.168.1.1 dev enp5s0 proto dhcp src 192.168.1.119 metric 100
10.0.0.0/24 dev enp10s0f0np0 proto kernel scope link src 10.0.0.2
10.1.0.1
        nexthop dev wg0 weight 1
        nexthop dev wg1 weight 1
```

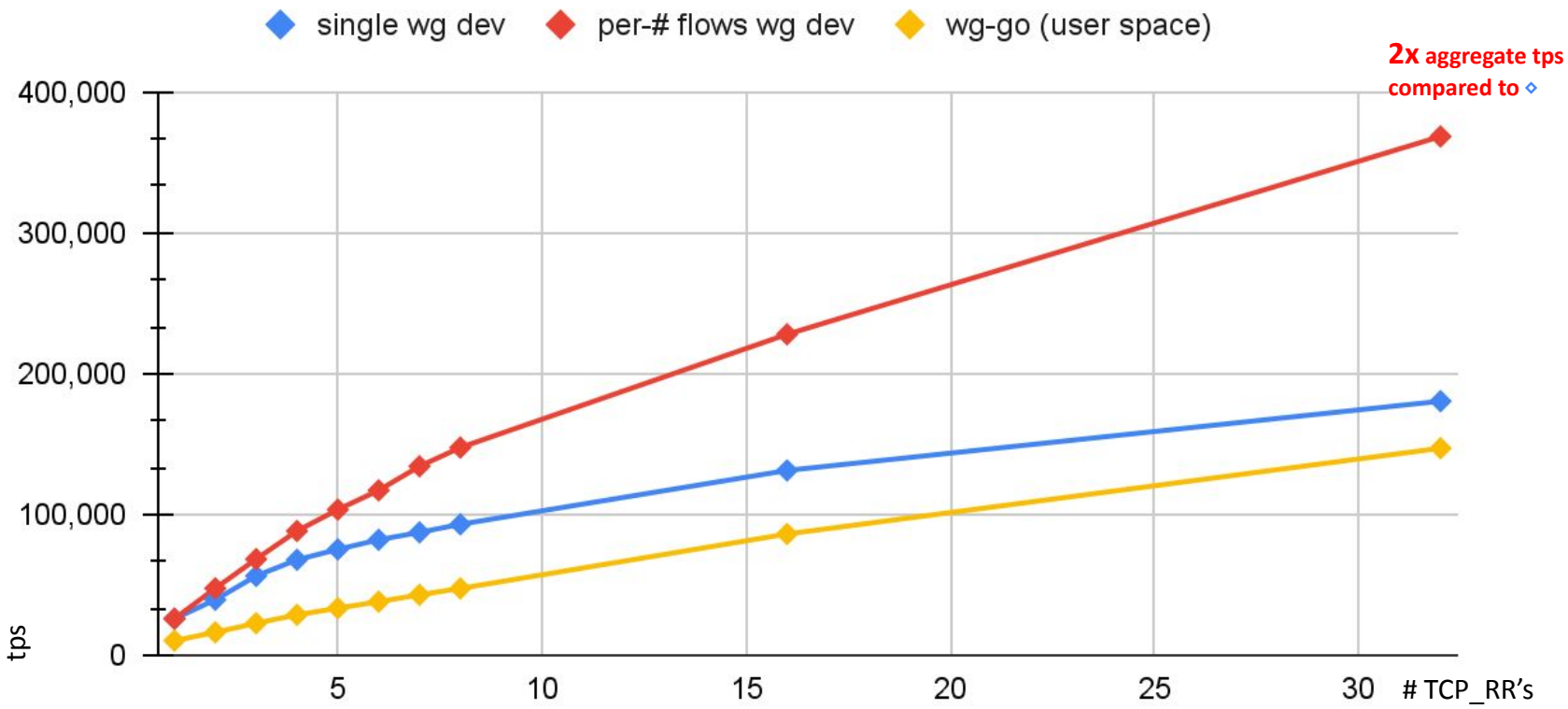
Options 2/2:

- Different listen-port but otherwise same peer key/endpoint/allowed-ip settings?
  - Not possible: WireGuard devices override each others settings
- Different listen-port and key-pairs, but same endpoint/allowed-ip settings?
  - Not possible: same as above
- Different “everything”?
  - Yes. But multipath next hops are not an option anymore.

TCP stream multi flow host to host over wire, 8k MTU (higher is better)



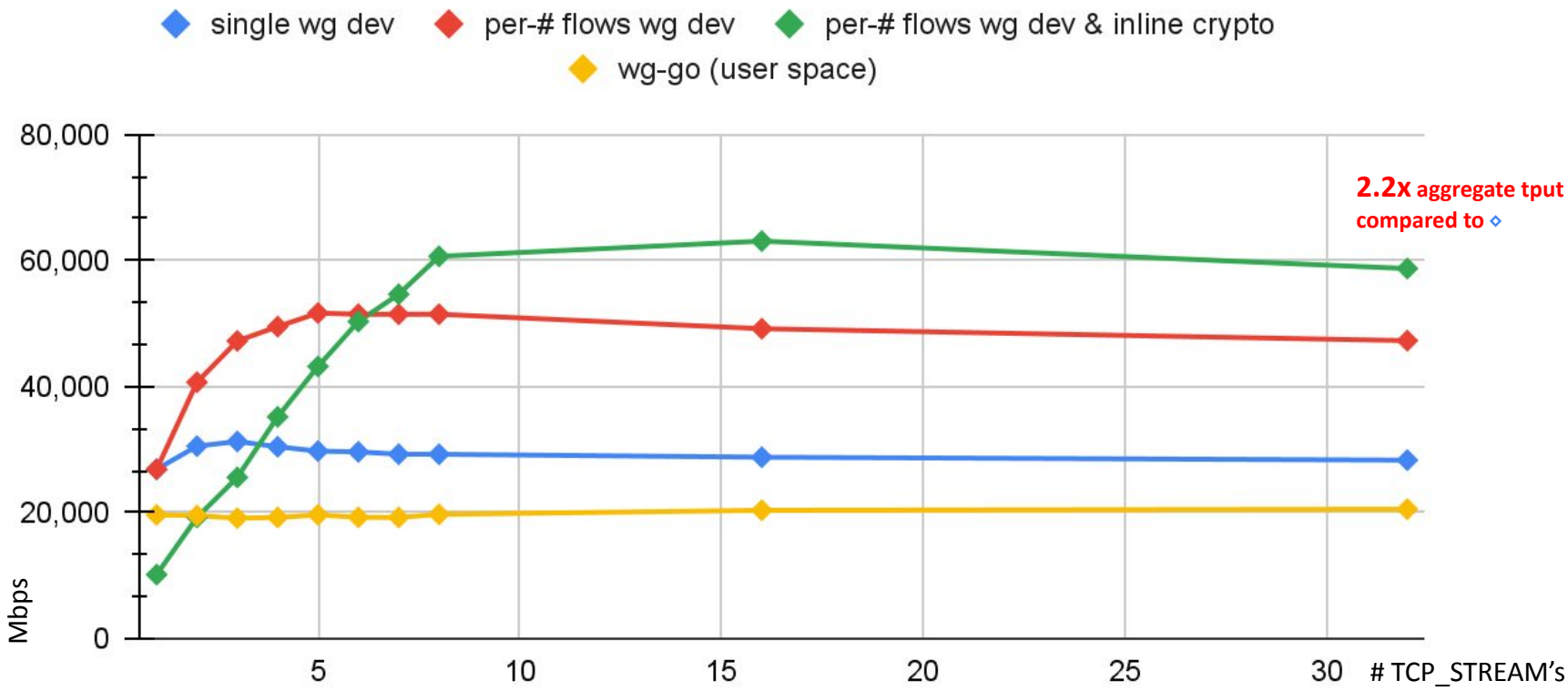
Transactions per second host to host over wire, 8k MTU (higher is better)



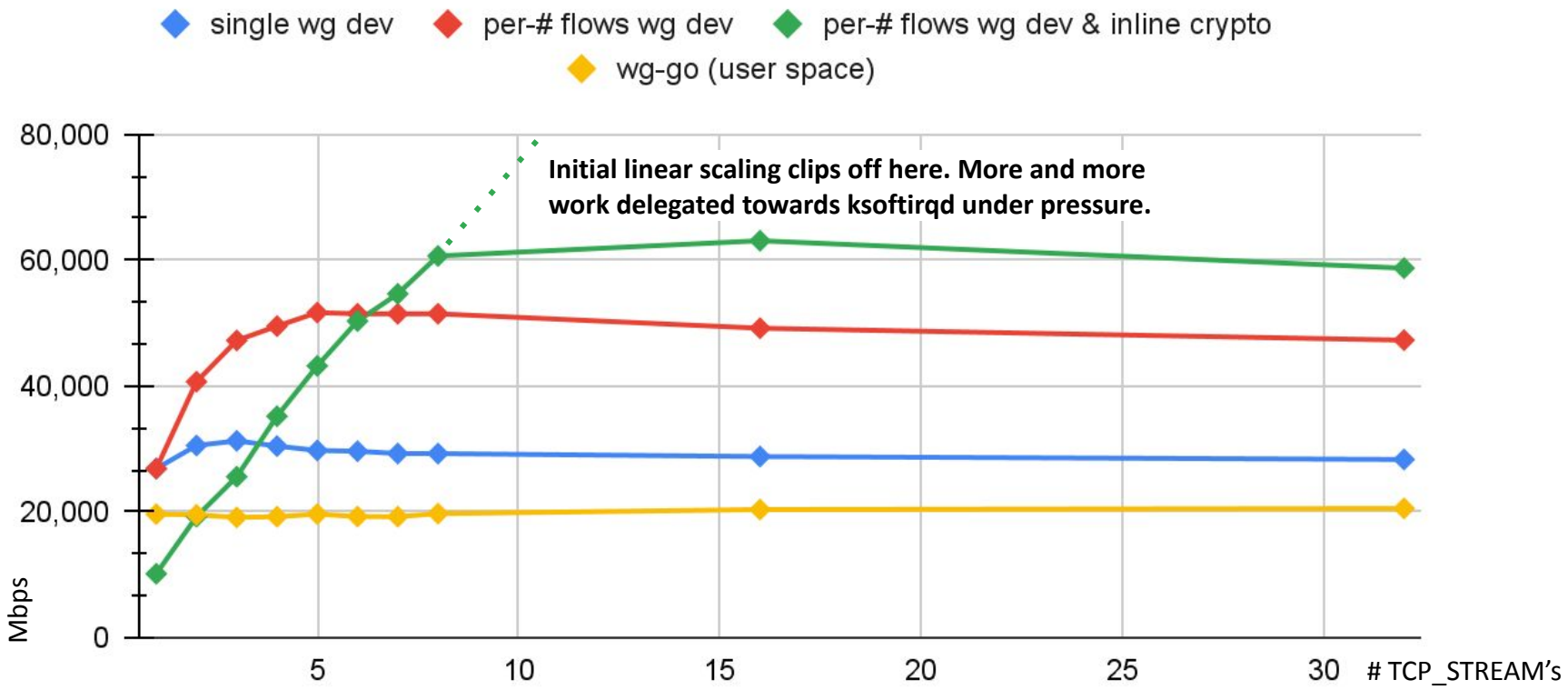
Thanks to RSS, scalability improves, but now the limitation shifts towards the non-local per-worker processing.

How does the CPU-local, ‘inline’ encryption and decryption compare with increasing number of parallel flows?

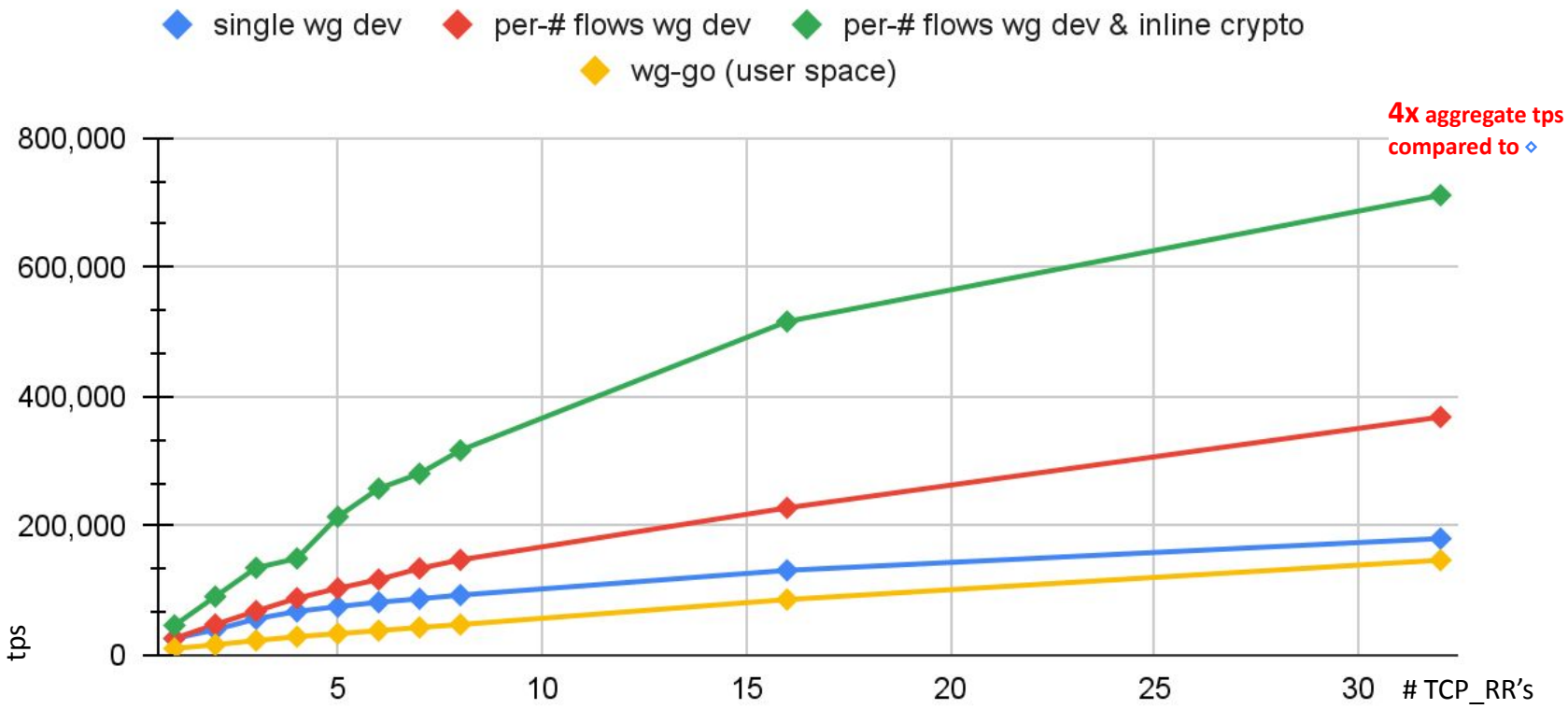
TCP stream multi flow host to host over wire, 8k MTU (higher is better)



TCP stream multi flow host to host over wire, 8k MTU (higher is better)



Transactions per second host to host over wire, 8k MTU (higher is better)



Instead of multiple devices, could we hash the inner packet into the outer wg header UDP source port for achieving RSS?

Source Port Hashing:

- On egress before encryption, packet hash is calculated and stored in control block
- After encryption, this is passed to udp_tunnel_xmit_skb as outer source port
- Concept is very similar to vxlan: Fixed global listener port, dynamic source port

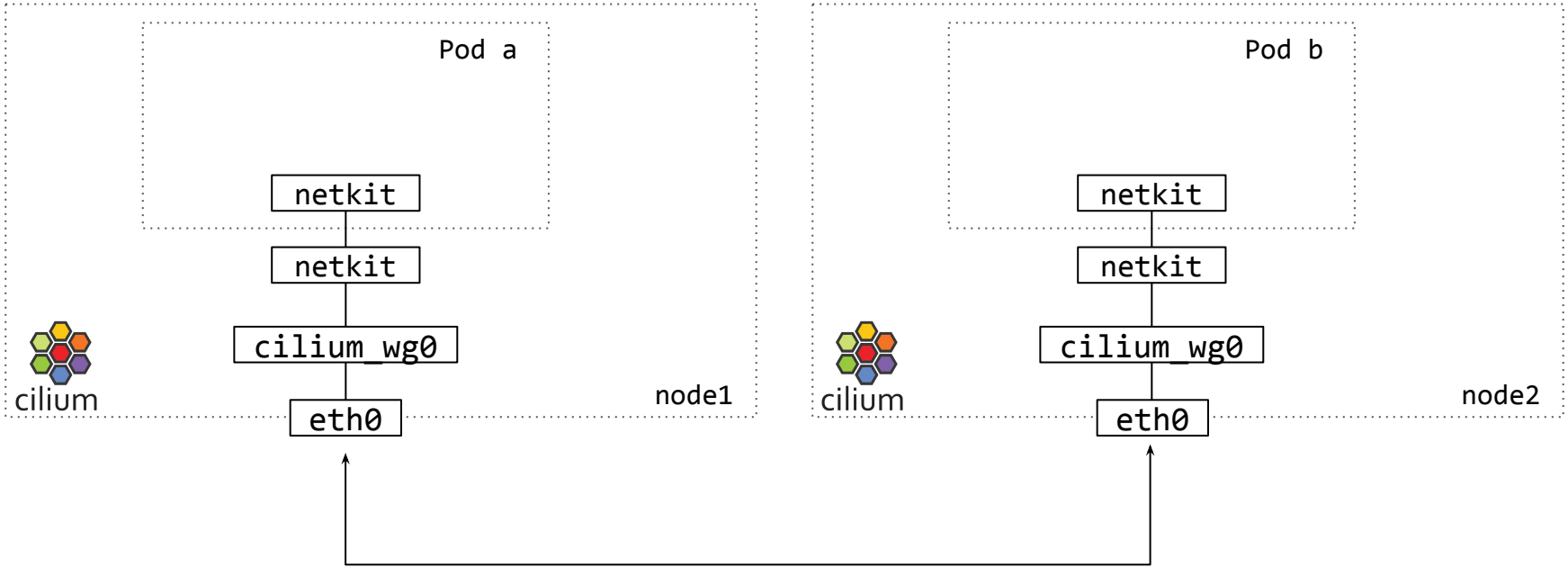
wg src port hashing

target node wg listen port

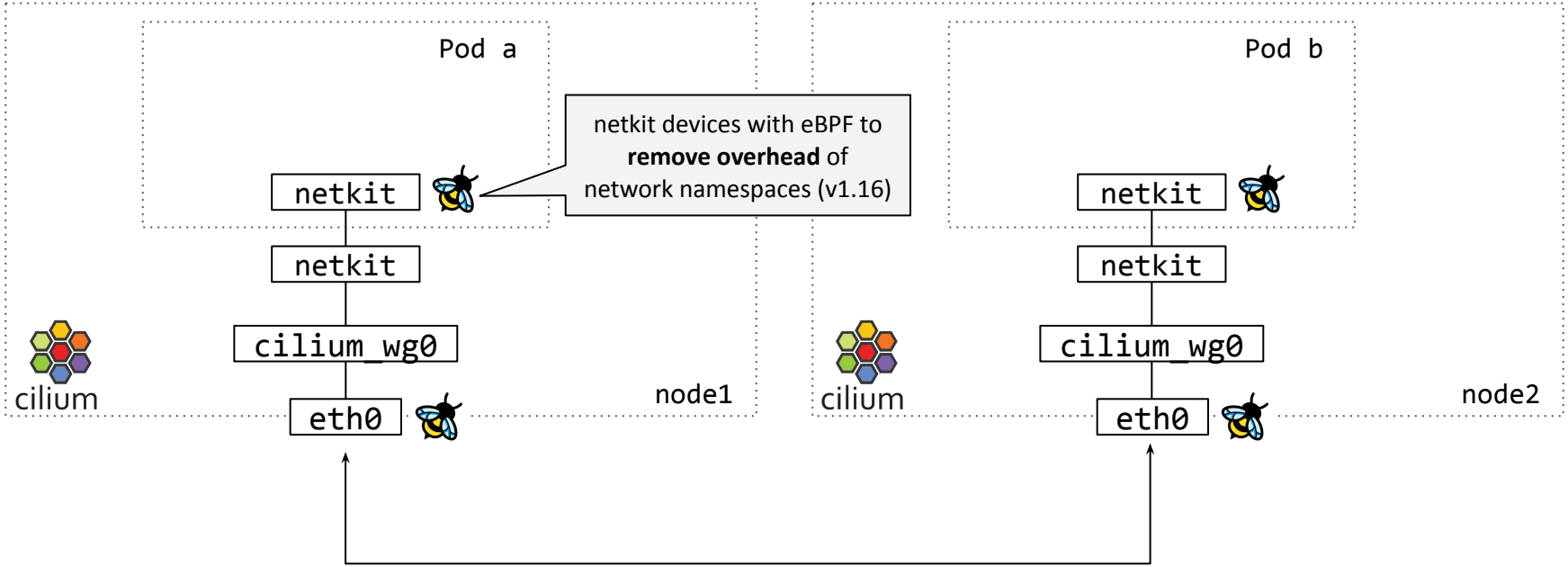
```
21:56:32.342774 enp10s0f0np0 Out IP 10.0.0.2.61051 > 10.0.0.1.9001: UDP, length 96
21:56:32.342776 enp10s0f0np0 Out IP 10.0.0.2.20735 > 10.0.0.1.9001: UDP, length 96
21:56:32.342778 enp10s0f0np0 Out IP 10.0.0.2.64010 > 10.0.0.1.9001: UDP, length 96
21:56:32.342802 enp10s0f0np0 Out IP 10.0.0.2.61051 > 10.0.0.1.9001: UDP, length 96
21:56:32.342810 enp10s0f0np0 Out IP 10.0.0.2.20735 > 10.0.0.1.9001: UDP, length 96
21:56:32.342812 enp10s0f0np0 Out IP 10.0.0.2.20507 > 10.0.0.1.9001: UDP, length 96
```

Step 4: Putting it all together - Cilium, eBPF & (optimized) WireGuard

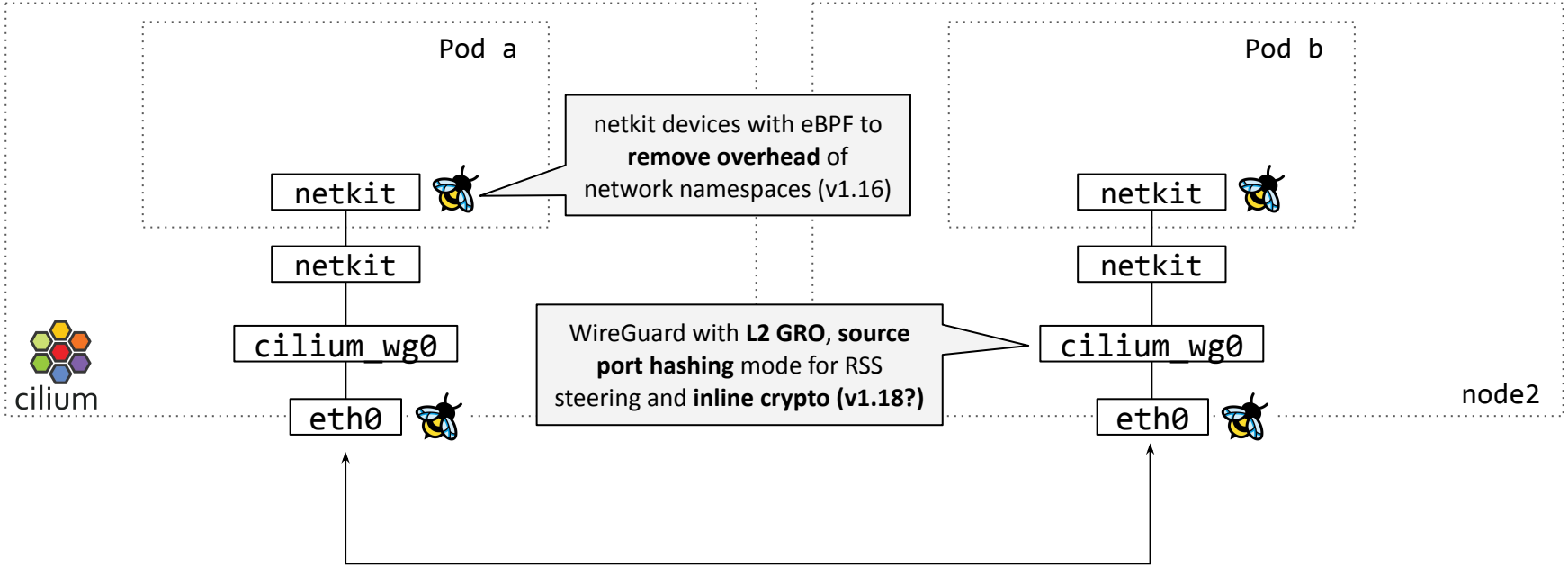
Architecture in a nutshell (outlook)



Architecture in a nutshell (outlook)



Architecture in a nutshell (outlook)



Recap & Conclusions

- Cilium v1.10+ provides “turn-key” integration of WireGuard:

```
cilium install --set encryption.enabled=true
               --set encryption.type=wireguard
```

- Cilium v1.10+ provides “turn-key” integration of WireGuard:

```
cilium install --set encryption.enabled=true
               --set encryption.type=wireguard
```

- While the crypto software cost remains, current WireGuard implementation can still be made significantly more scalable

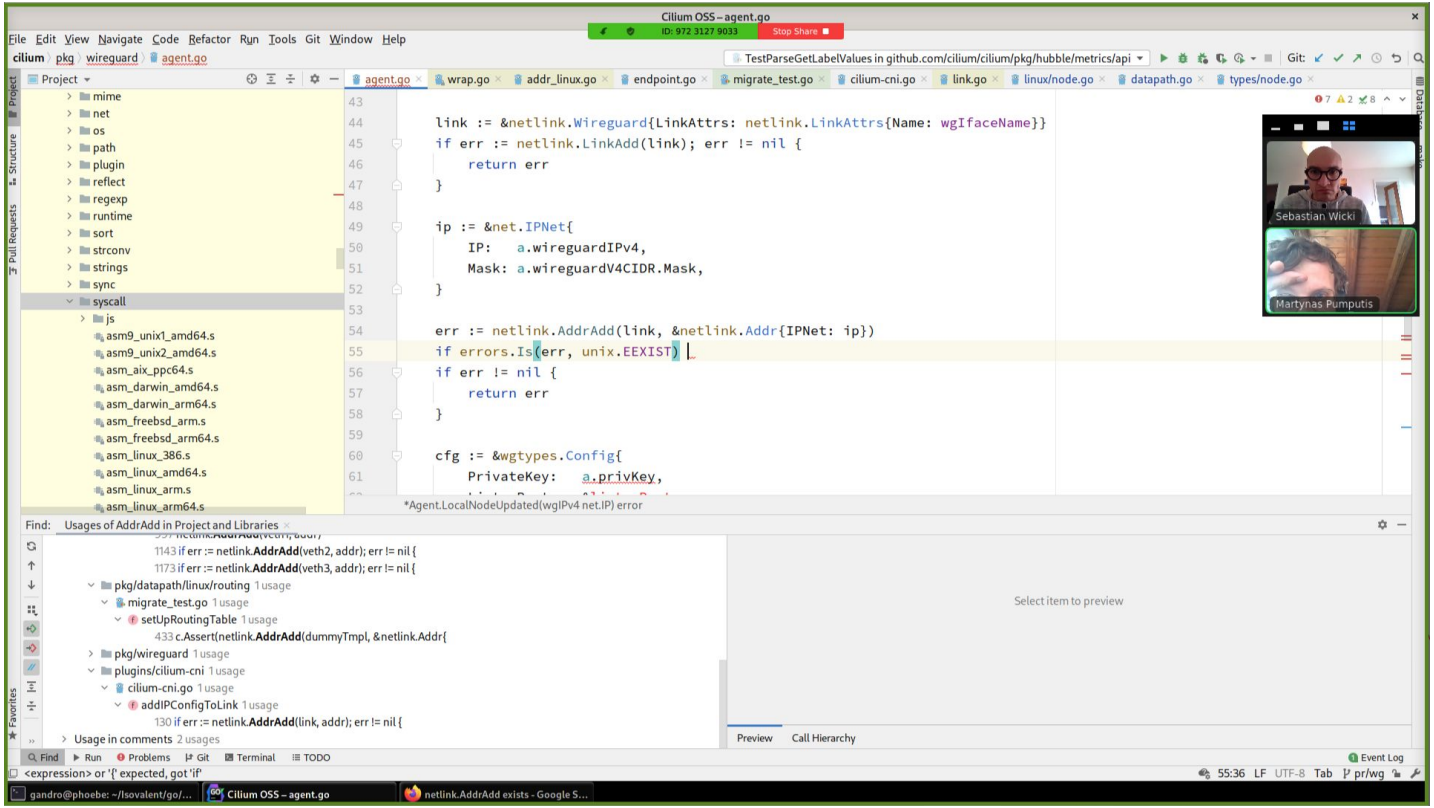
- Cilium v1.10+ provides “turn-key” integration of WireGuard:

```
cilium install --set encryption.enabled=true
               --set encryption.type=wireguard
```

- While the crypto software cost remains, current WireGuard implementation can still be made significantly more scalable

- Improvements mentioned here will be submitted for merging into the Linux kernel as well as Cilium for the broader benefit

Acknowledgements



Acknowledgements

Jason A. Donenfeld (WireGuard* creator)

Jordan Whited & James Tucker (WireGuard-go improvements)

Davide Miola, Mirco Barone et al. (WireGuard inline crypto)

Martynas Pumputis & Sebastian Wicki (initial Cilium integration)

Cilium, netdev & BPF communities

* “WireGuard” is a registered trademark of Jason A. Donenfeld

ISOVALENT
now part of CISCO

Thank you! Questions?

github.com/cilium/cilium

cilium.io

ebpf.io

wireguard.com

Cilium & WireGuard Instruqt Lab

Cilium & WireGuard Documentation

Appendix: Docs, Tutorials, Labs

- WireGuard Whitepaper
- Cilium Transparent Encryption with IPsec and WireGuard
- Cilium Online Learning Labs
- Cilium Docs on Transparent Encryption
- The original presentation on WireGuard in Cilium

- Performance related: IPSec transition to local CPU encryption and decryption, another talk on WireGuard with similar findings