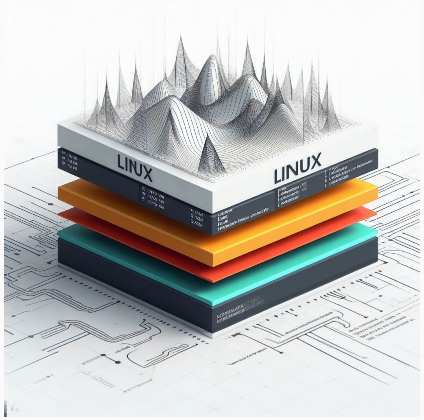
Exploring Linux Kernel Analysis through Graph Theory: A Network Stack Perspective

Hagen Paul Pfeifer <hagen@jauu.net>

Everything Linux



The Linux network stack forms the backbone of the Internet

- Nearly all cloud or edge services are based on Linux (AWS, Azure, Google)
- A large part of all embedded and IoT devices runs on Linux
- Core switching and routing products are based on Linux (with help of offloading capabilities, Cisco IOS, Arista, ...)
- 5G/6G - SDN, OpenRAN

Linux has the network stack with the most extensive protocol support - RFC standard compliant in most areas and very efficient

If you are doing something with networking - you are all in Linux

Understand Linux

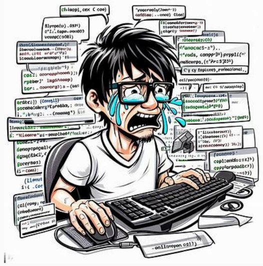
Demystify the Network Stack

To add new network or protocol functionality, to extend the kernel, or to understand protocol behavior, an extensive network stack understanding is required

- e.g. adding another tunnel protocol to the kernel

Comprehensive understanding of the "dirty details" is the prerequisite for many things and the key to success!

BUT



Linux kernel is complex and highly optimized for nearly every instruction - repeat: kernel code is difficult and hard to understand

- Code execution end in "dead ends", caused by deferred execution of code by mechanisms like RCU, NAPI or SoftIRQ contexts for network processing
- The kernel works with many function pointers - simple reading code often does not help (i.e. struct file_operations)

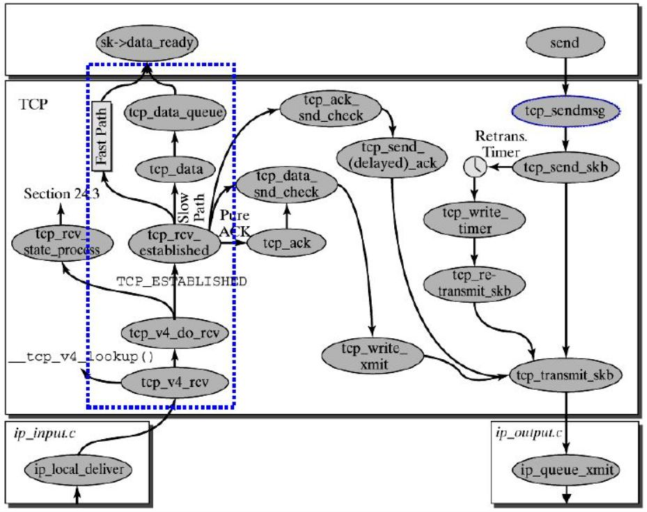
Understand the Big Picture

Demystify the Network Stack

Complex interrelationships, especially sequences, can be illustrated visually in a very comprehensible way

Ideal is a "runtime" feeling what functions is actually called, in the right order

- What are the main branches - which are called most frequently? What are error branches and which function is never called?
- The entire, actual sequence of calls helps to understand the big picture.
- It is no longer necessary to understand every single line in order to piece together manually the big picture



Behind the Scenes - Ftrace

Demystify the Network Stack

Callgraph data capturing is done by ftrace function tracer

- In kernel, low overhead capturing engine done by Steve Rostedt & RT friends
- Commanding by reading/writing to tracefs - a pseudo filesystem

ftrace-callgrapher utilize this functionality and abstract away complicated things like dealing with ring-buffer sizes and co

Background: Kernel compiled with CONFIG_DYNAMIC_FTRACE

- instruct kernel to compile with gcc option -pg and -mfentry
- Every traceable function gets a special mcount/mfence call
- During kernel boot all calls are replaced with NOPs
- NOPs: no measurable runtime overhead
- If measurement is enabled: particular NOPs are re-patched again

```
<__x64_sys_getsockopt>:
       e8 eb 3c 8d ff          call   ffffffff81071220 <__fentry__>
       48 8b 4f 38             mov    0x38(%rdi),%rcx
       4c 8b 47 48             mov    0x48(%rdi),%r8
       8b 57 60                mov    0x60(%rdi),%edx
       8b 77 68                mov    0x68(%rdi),%esi
       8b 7f 70                mov    0x70(%rdi),%edi
       e8 05 fe ff ff          call   ffffffff8179d350 <__sys_getsockopt>
       48 98                   cltq
       e9 6e 46 66 00          jmp    ffffffff81e01bc0 <__x86_return_thunk>
       66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
       00 00 00 00
       0f 1f 00                nopl   (%rax)
```

ftrace-callgrapher - Hello World

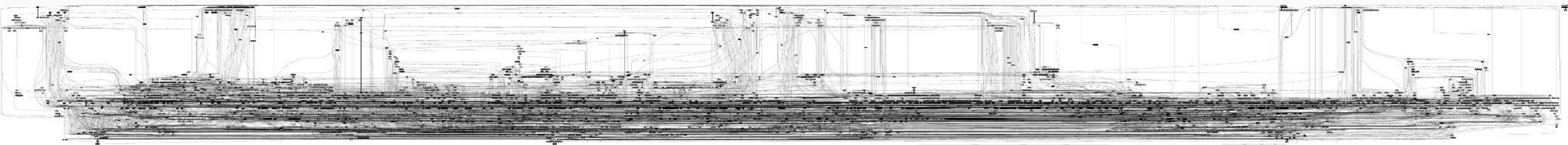
A Ftracer Frontend

Step 1: capture data

```
$ ftrace-callgrapher.py record --record-time 100 --cpumask 1
Record mode - now starting recording traces for 10.0 seconds
Limit recording to CPU mask 1
wrote data to ftrace-callgrapher.data
Recorded filesize: 199.38 MiB
```

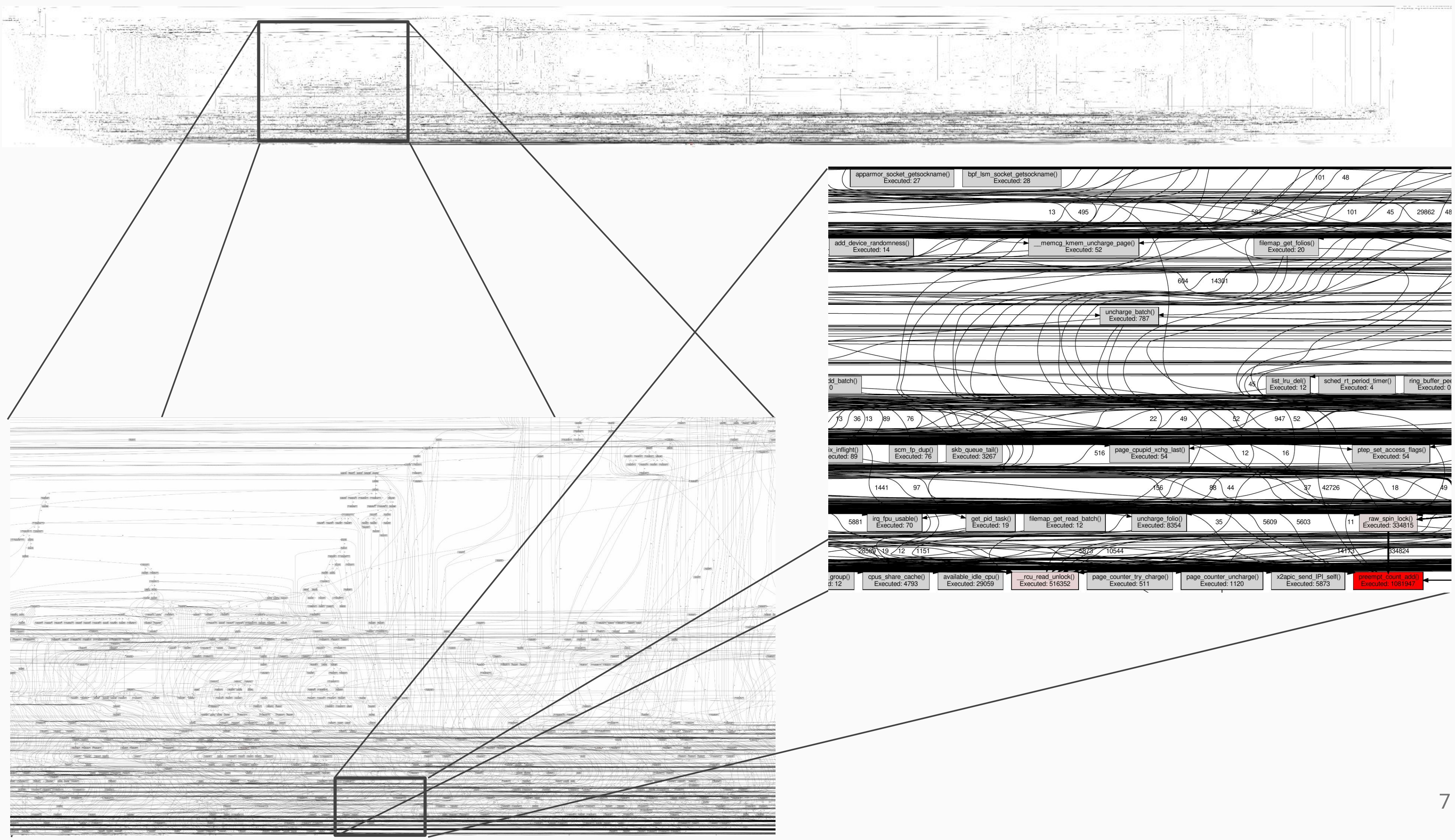
Step 2: visualize

```
$ ftrace-callgrapher.py visualize
Visualization mode - now generating visualization...
parsing completed, found 2316184 events
581015 events missed during capturing process (20.05%)
function-calls.png generated
Warning: graph has 3227 nodes...layout may take a long time.
ftrace-callgrapher.pdf generated
```



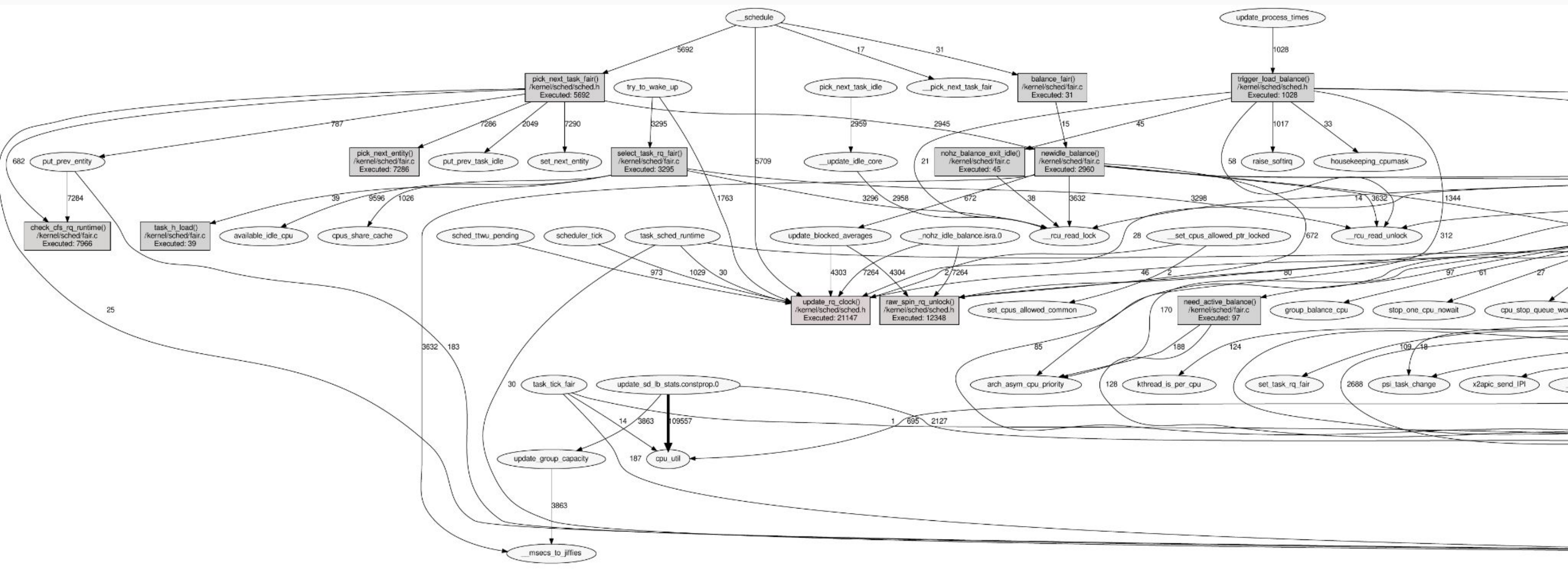
Big Data - Big Problems

CPU_0 Kernel Call Chains - Capture Period 10sec on an Idle System

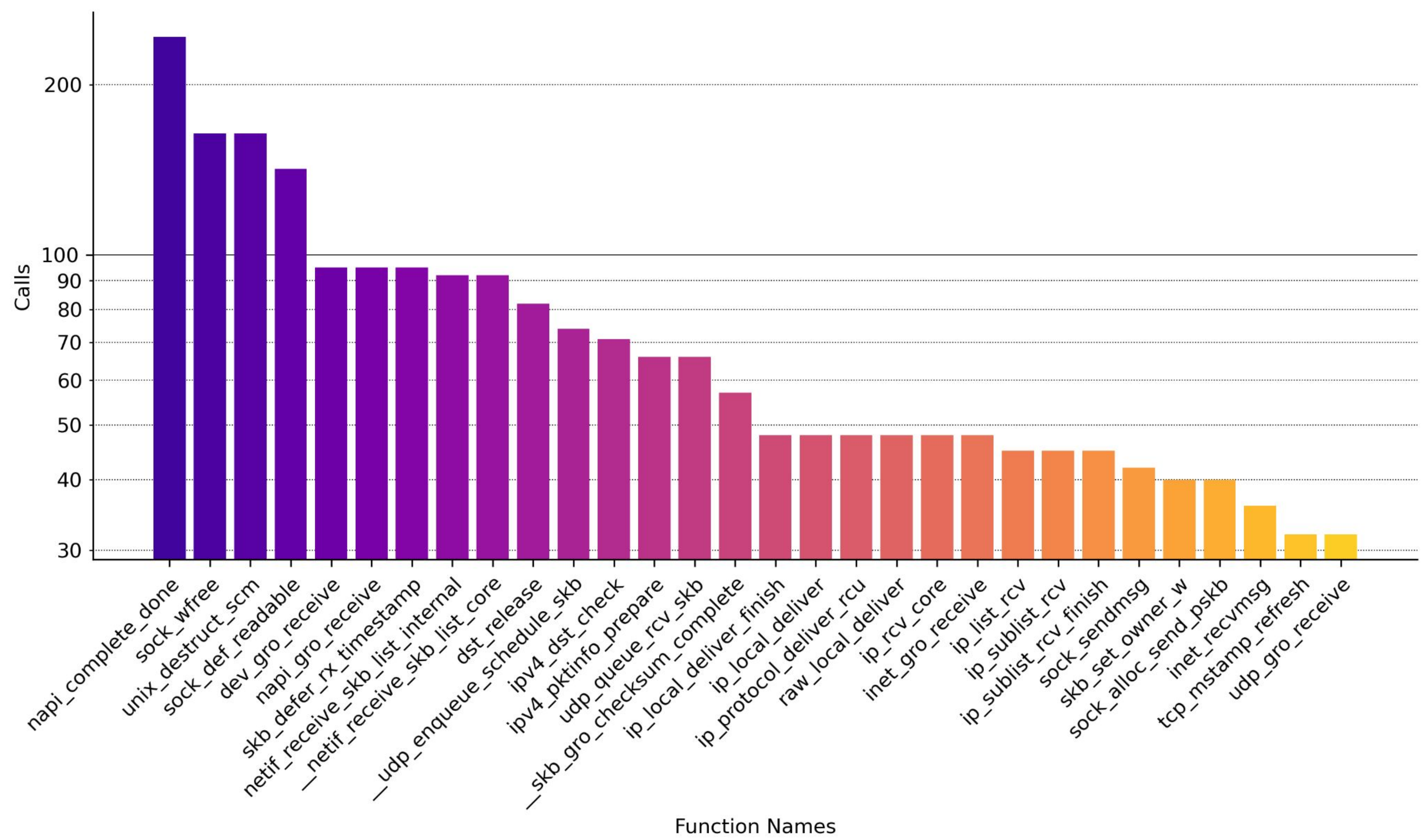


Symbol Filtering

```
$ apt-get install linux-image-(uname -r)-dbg
$ ftrace-callgrapher.py generate-symbol-map -k /usr/lib/debug/boot/vmlinux-$(uname -r)
$ ftrace-callgrapher.py visualize --filter-filepath kernel/sched/fair.c,/kernel/sched/sched.h
# ftrace-callgrapher.py visualize --filter-filepath net
```



Bonus



Calls
200
100
90
80
70
60
50
40
30
napi_complete_done
sock_wfree
unix_destruct_scm
sock_def_readable
dev_gro_receive
napi_gro_receive
skb_defer_rx_timestamp
netif_receive_skb_list_internal
__netif_receive_skb_list_core
dst_release
__udp_enqueue_schedule_skb
ipv4_dst_check
ipv4_pktinfo_prepare
udp_queue_rcv_skb
__skb_gro_checksum_complete
ip_local_deliver_finish
ip_local_deliver
ip_protocol_deliver_rcu
raw_local_deliver
ip_rcv_core
inet_gro_receive
ip_list_rcv
ip_sublist_rcv
ip_sublist_rcv_finish
sock_sendmsg
skb_set_owner_w
sock_alloc_send_pskb
inet_recvmsg
tcp_mstamp_refresh
udp_gro_receive
Function Names

Thank You!

Questions?

Source Code and Project Home:

https://github.com/hgn/ftrace-callgrapher