# Netconf 2024 topics

Jason Xing
<kernelxing@tencent.com>

# Extending SO_TIMESTAMPING Feature

# 1. History

What are the major milestones more than one decade?

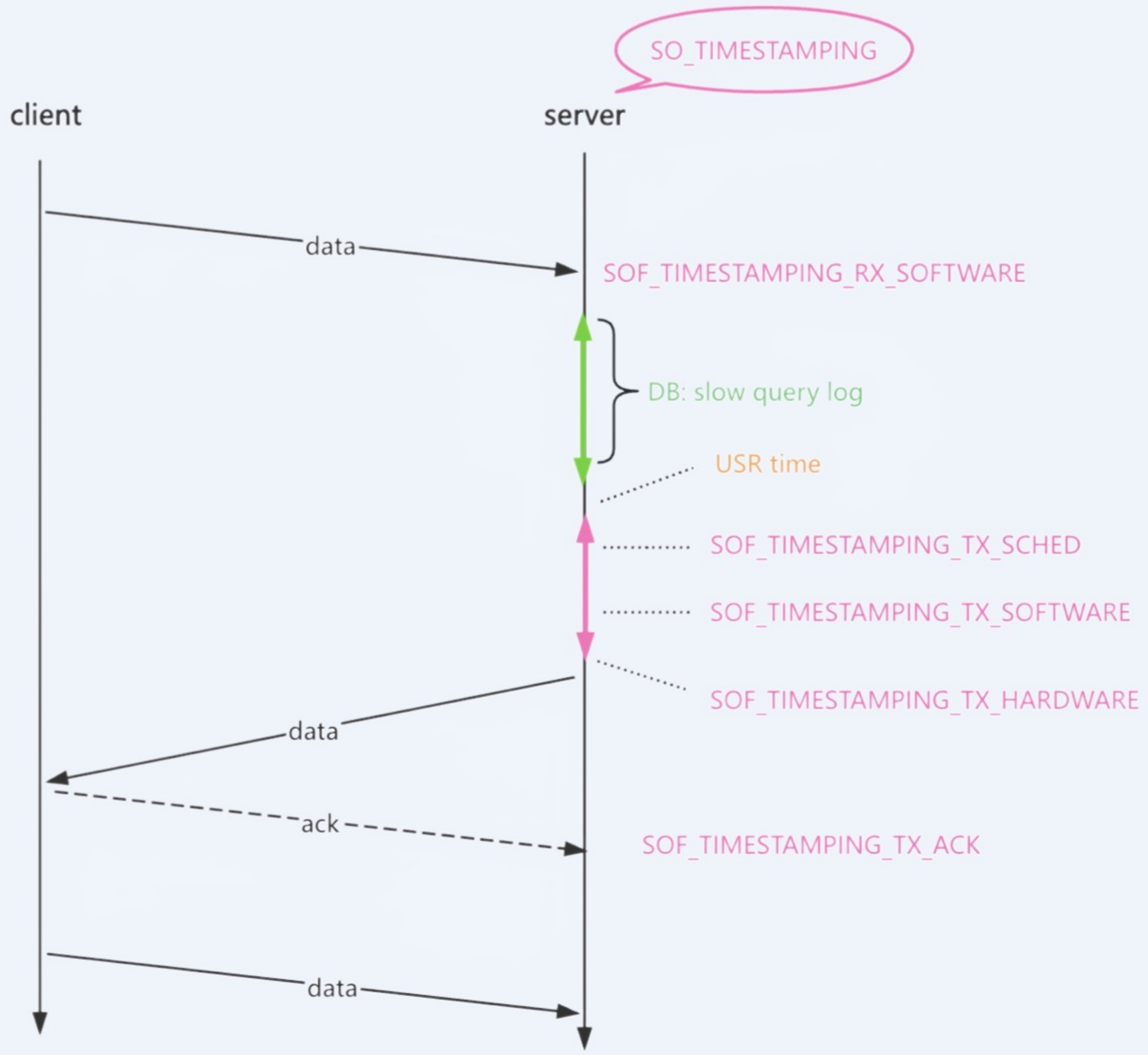
- It was first introduced in 2009 by Patrick Ohly with SOFTWARE and HARDWARE timestamp features only.
- In 2014 and in the next few years, Willem de Bruijn and other developers implemented a few important generation and control flags. Then, users have the abilities to trace the historical behaviors well to know what exactly happened.
- Google showed best practices (like Fathom & Dapper) in production in 2022 and 2023.

# 2. Use Cases

What can we do with this feature enabled?

- For users, SO_TIMESTAMPING mostly serves the RPC applications and serves well
- For admins, SO_TIMESTAMPING can be used as a detector to know what the situations in qdisc/driver/hardware/network are

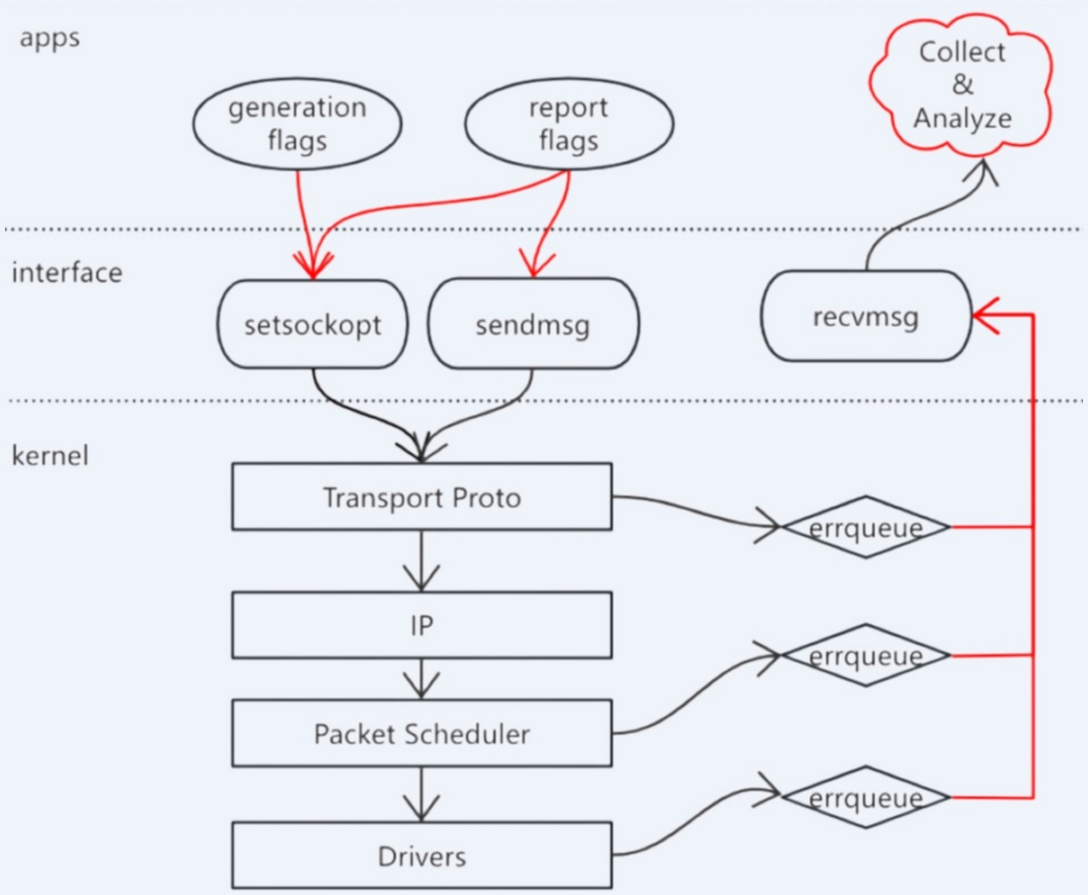
# 3. Goals


SO_TIMESTAMPING
client
server
data
SOF_TIMESTAMPING_RX_SOFTWARE
DB: slow query log
USR time
SOF_TIMESTAMPING_TX_SCHED
SOF_TIMESTAMPING_TX_SOFTWARE
SOF_TIMESTAMPING_TX_HARDWARE
data
ack
SOF_TIMESTAMPING_TX_ACK
data

Is it enough for DB or RPC applications currently?

- In the short run, I am looking for a good way of deploying in production **quickly and transparently**.
- In the long run, I am trying to equip the feature with **more effective information** and efficient means for users/admins

# 4. Current State



## Details

SO_TIMESTAMPING is really GOOD, which can help users detect the latency in the rx/tx path, but....

We are facing three kinds of dilemmas, see the red lines:

- Applications need changes
  - Modify the params in setsockopt/sendmsg/recvmsg
  - for(...) loops to parse the cmsgs
- Additional overhead per recv call
  - Syscalls: 0-2 times setsockopt() + X times recvmsg()
    (X is the number of tx generation flags)
  - Extra analysis in apps
- Uapi limits the future
  - If we have flaws in the original design
  - If we have more useful data to output

# 5. setsockopt

Make the first move to set option without any modification in apps

- Let it pass the check in sol_socket_sockopt(), which is implemented in the kernel module, and then set the existing/new sockets with related flags.
- In the future, we could implement bpf_setsockopt to help us.
  - We are going to implement more new trace functions soon, or else it is meaningless

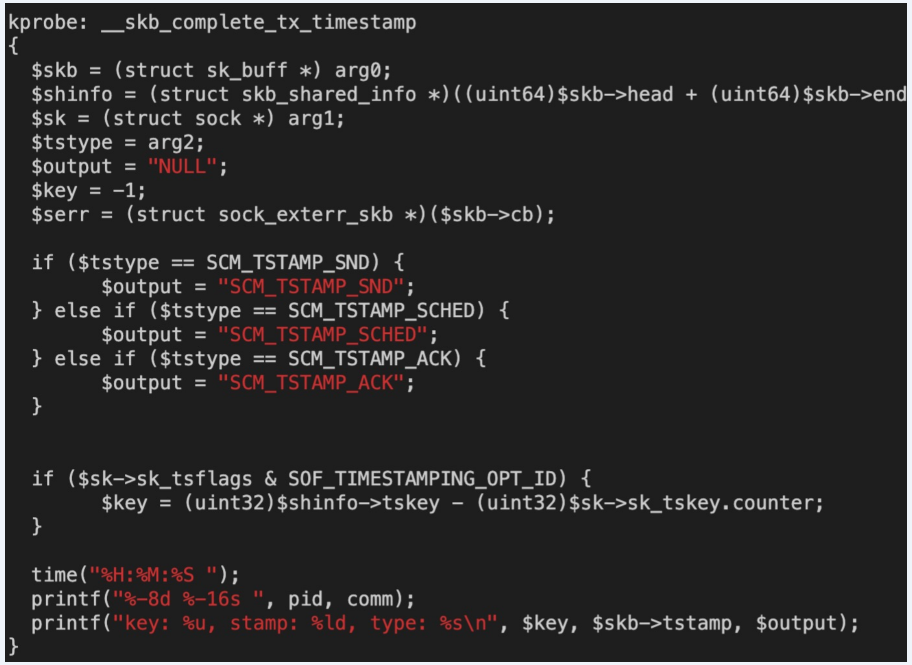
# 6.1 uprobe

If we try uprobe, what will happen?

- “Single digit, two microseconds” . Not acceptable :(
- Willem once talked about this one in netdevconf 0x17

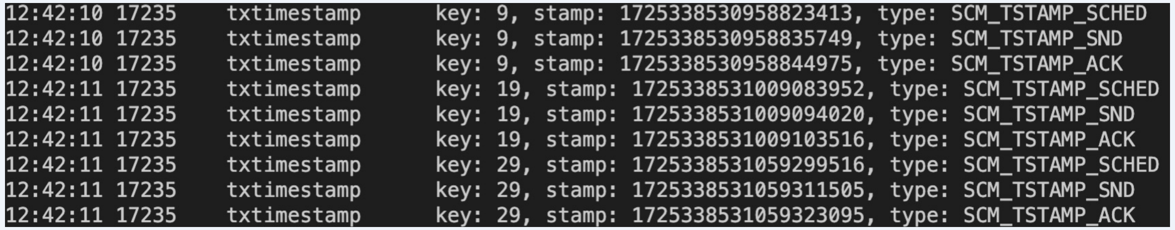
# 6.2 kprobe (tx path)

Hook when generating tx timestamp



```
kprobe: __skb_complete_tx_timestamp
{
  $skb = (struct sk_buff *) arg0;
  $shinfo = (struct skb_shared_info *)((uint64)$skb->head + (uint64)$skb->end
  $sk = (struct sock *) arg1;
  $tstype = arg2;
  $output = "NULL";
  $key = -1;
  $serr = (struct sock_exterr_skb *)($skb->cb);

  if ($tstype == SCM_TSTAMP_SND) {
        $output = "SCM_TSTAMP_SND";
  } else if ($tstype == SCM_TSTAMP_SCHED) {
        $output = "SCM_TSTAMP_SCHED";
  } else if ($tstype == SCM_TSTAMP_ACK) {
        $output = "SCM_TSTAMP_ACK";
  }


  if ($sk->sk_tsflags & SOF_TIMESTAMPING_OPT_ID) {
        $key = (uint32)$shinfo->tskey - (uint32)$sk->sk_tskey.counter;
  }

  time("%H:%M:%S ");
  printf("%-8d %-16s ", pid, comm);
  printf("key: %u, stamp: %ld, type: %s\n", $key, $skb->tstamp, $output);
}
```

Test with “ ./txtimestamp -4 -C -L 127.0.0.1 ”



```
12:42:10 17235    txtimestamp      key: 9, stamp: 1725338530958823413, type: SCM_TSTAMP_SCHED
12:42:10 17235    txtimestamp      key: 9, stamp: 1725338530958835749, type: SCM_TSTAMP_SND
12:42:10 17235    txtimestamp      key: 9, stamp: 1725338530958844975, type: SCM_TSTAMP_ACK
12:42:11 17235    txtimestamp      key: 19, stamp: 1725338531009083952, type: SCM_TSTAMP_SCHED
12:42:11 17235    txtimestamp      key: 19, stamp: 1725338531009094020, type: SCM_TSTAMP_SND
12:42:11 17235    txtimestamp      key: 19, stamp: 1725338531009103516, type: SCM_TSTAMP_ACK
12:42:11 17235    txtimestamp      key: 29, stamp: 1725338531059299516, type: SCM_TSTAMP_SCHED
12:42:11 17235    txtimestamp      key: 29, stamp: 1725338531059311505, type: SCM_TSTAMP_SND
12:42:11 17235    txtimestamp      key: 29, stamp: 1725338531059323095, type: SCM_TSTAMP_ACK
```

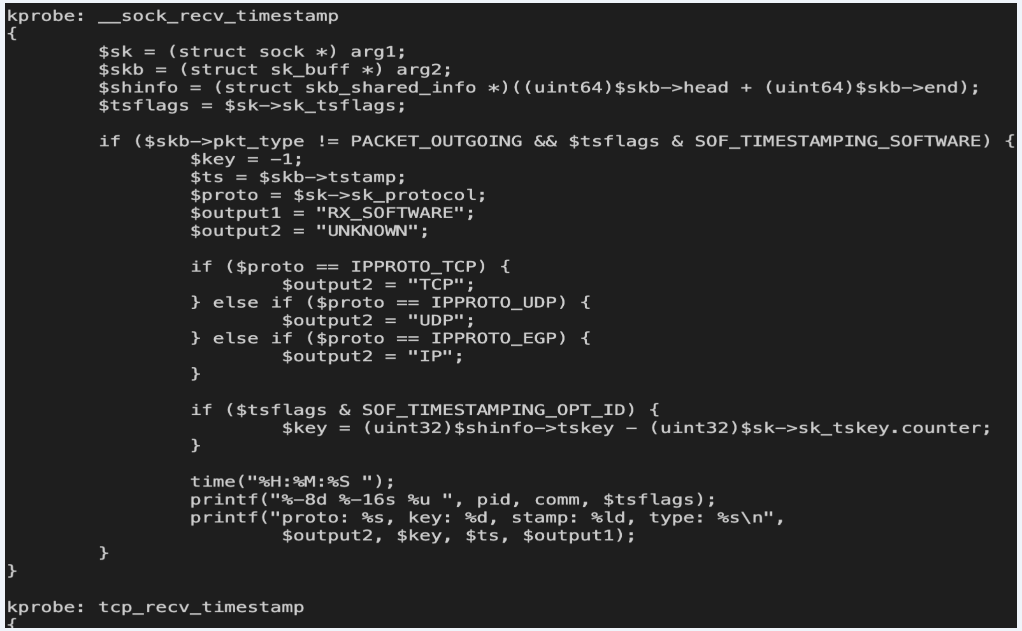
## Can we try kprobe in the egress path?

- Try to hook during the generating timestamp phase
  - SCM_TSTAMP_(SND/SCHED/ACK)
    - Track __skb_complete_tx_timestamp() √
    - Not consider the report flag
  - OPT_STATS
    - Re-implement tcp_get_timestamping_opt_stats() √

- Try to hook during the reporting timestamp phase
  - SCM_TSTAMP_(SND/SCHED/ACK)
    - Rely on extra recvmsg() syscalls ×
    - Re-implement __sock_recv_timestamp() √
  - OPT_STATS
    - Hard to parse skb->data in different kernels ×

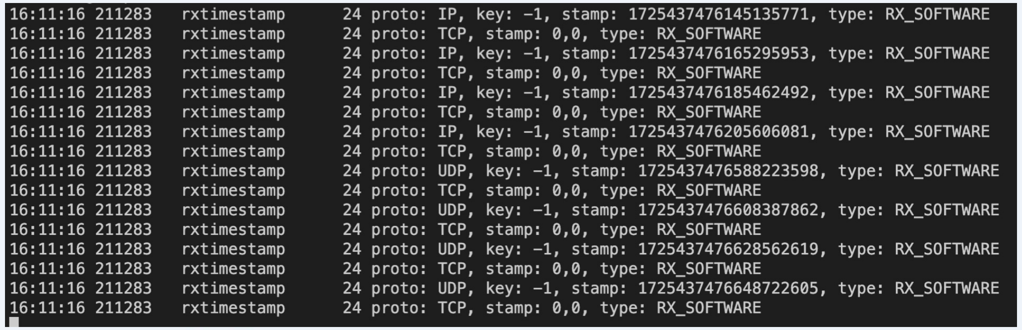
# 6.3 kprobe (rx path)

Hook when reporting tx timestamp



```
kprobe: __sock_recv_timestamp
{
        $sk = (struct sock *) arg1;
        $skb = (struct sk_buff *) arg2;
        $shinfo = (struct skb_shared_info *)((uint64)$skb->head + (uint64)$skb->end);
        $tsflags = $sk->sk_tsflags;

        if ($skb->pkt_type != PACKET_OUTGOING && $tsflags & SOF_TIMESTAMPING_SOFTWARE) {
                $key = -1;
                $ts = $skb->tstamp;
                $proto = $sk->sk_protocol;
                $output1 = "RX_SOFTWARE";
                $output2 = "UNKNOWN";

                if ($proto == IPPROTO_TCP) {
                        $output2 = "TCP";
                } else if ($proto == IPPROTO_UDP) {
                        $output2 = "UDP";
                } else if ($proto == IPPROTO_EGP) {
                        $output2 = "IP";
                }

                if ($tsflags & SOF_TIMESTAMPING_OPT_ID) {
                        $key = (uint32)$shinfo->tskey - (uint32)$sk->sk_tskey.counter;
                }

                time("%H:%M:%S ");
                printf("%-8d %-16s %u ", pid, comm, $tsflags);
                printf("proto: %s, key: %d, stamp: %ld, type: %s\n",
                        $output2, $key, $ts, $output1);
        }
}

kprobe: tcp_recv_timestamp
{
```

Test with “./rxtimestamp”



```
16:11:16 211283   rxtimestamp      24 proto: IP, key: -1, stamp: 1725437476145135771, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: IP, key: -1, stamp: 1725437476165295953, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: IP, key: -1, stamp: 1725437476185462492, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: IP, key: -1, stamp: 1725437476205606081, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: UDP, key: -1, stamp: 1725437476588223598, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: UDP, key: -1, stamp: 1725437476608387862, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: UDP, key: -1, stamp: 1725437476628562619, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: UDP, key: -1, stamp: 1725437476648722605, type: RX_SOFTWARE
16:11:16 211283   rxtimestamp      24 proto: TCP, stamp: 0,0, type: RX_SOFTWARE
```

Can we try kprobe in the ingress path?

- Try to hook during the generating timestamp phase
  - RX_SOFTWARE
    - performance degradation ×
      - Over 15% on the loopback

- Try to hook during the reporting timestamp phase
  - RX_SOFTWARE
    - Re-implement the same logic √
      - TCP: tcp_recv_timestamp()
      - Others: __sock_recv_timestamp()

# 6.4 kprobe (conclusion)

Is the kprobe method practical?

- Hook everywhere ☹
- Take care of so many conditions ☹
- Partially deprecate the existing flags ☹
- Cumulative skbs residing in the errqueue, which consumes sk_rmem_alloc ☹
- …… ☹
- Require less or even no modifications in apps, say, retrieving cmsgs in the loop ☺
- Admins get benefits ☺
- Easily extend OPT_STAT ☺

# 6.5 sk_rmem_alloc issue

```
@@ -5517,6 +5517,10 @@ void __skb_tstamp_tx(struct sk_buff *orig_skb,
        if (!skb_may_tx_timestamp(sk, tsonly))
                return;

+       /* Match the specific flow and then return eariler */
+       if (TUPLE_MATCH(sk))
+               return;
+
        if (tsonly) {
 #ifdef CONFIG_INET
                if ((tsflags & SOF_TIMESTAMPING_OPT_STATS) &&
```

## How to avoid “cumulative skbs residing in the errqueue” ?

- It only happens on the tx generation phase
- Hook __skb_tstamp_tx()
  - sysctl –w net.core.tstamp_allow_data =0
  - not enable SOF_TIMESTAMPING_OPT_TSONLY
  - remove CAP_NET_RAW from socket file
  - without super root privilege
- Actually I wrote a kernel module to hack and bypass this part instead of the above workaround.

# 7.1 tracepoint (basic theory)

Since it is inevitable to modify the kernel, how about…?

- Insert the tracepoint into the specific position where it is going to report to the user space, so that we can easily parse the useful data.

- No need to re-consider so many if-else and tsflags conditions

# 7.2 tracepoint (code snippets v1)

TencentOS

```
@@ -976,6 +977,11 @@ void __sock_recv_timestamp(struct msghdr *msg, struct sock *sk,
                else
                        put_cmsg_scm_timestamping(msg, &tss);

+               if (!skb_is_err_queue(skb))
+                       trace_ingress_ts(sk, &tss);
+               else
+                       trace_egress_ts(sk, &tss, &(SKB_EXT_ERR(skb)->ee));
+


@@ -2293,6 +2294,8 @@ void tcp_recv_timestamp(struct msghdr *msg, const struct sock *sk,
                        put_cmsg_scm_timestamping64(msg, tss);
                else
                        put_cmsg_scm_timestamping(msg, tss);
+
+               trace_ingress_ts(sk, tss);
        }
 }
```

A few notes:

- Insert into the report function
- Print them when we are ready.
- Simple really but…
- Still rely on extra syscalls (recvmsg(MSG_ERRQUEUE))

# 7.3 tracepoint (code snippets v2)

```
@@ -2293,6 +2294,8 @@ void tcp_recv_timestamp(struct msghdr *msg, const struct sock *sk,
                        put_cmsg_scm_timestamping64(msg, tss);
                else
                        put_cmsg_scm_timestamping(msg, tss);
+
+               trace_ingress_ts(sk, tss);

@@ -976,6 +977,9 @@ void __sock_recv_timestamp(struct msghdr *msg, struct sock *sk,
                else
                        put_cmsg_scm_timestamping(msg, &tss);

+               if (!skb_is_err_queue(skb))
+                       trace_ingress_ts(sk, &tss);

@@ -5422,6 +5424,10 @@ static void __skb_complete_tx_timestamp(struct sk_buff *skb,

        if (err)
                kfree_skb(skb);
+       else if (tsflags & SOF_TIMESTAMPING_SOFTWARE ||
+                (tsflags & SOF_TIMESTAMPING_RAW_HARDWARE &&
+                skb_hwtstamps(skb)->hwtstamp))
+               trace_egress_ts(sk, skb, &serr->ee)
```

## A few notes:

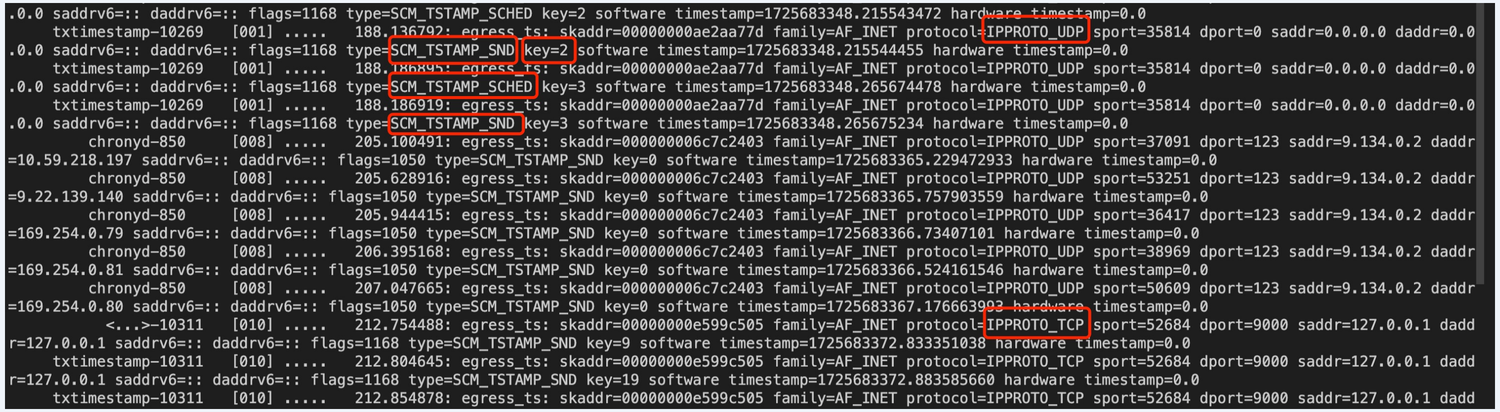
- Insert into the generating function
- We only handle errqueue in tx path
- We have to add report flags check during the generation phase
- One minor problem is that we mix the generation and report logic

# 7.4 tracepoint (output)

TencentOS

## TX monitor

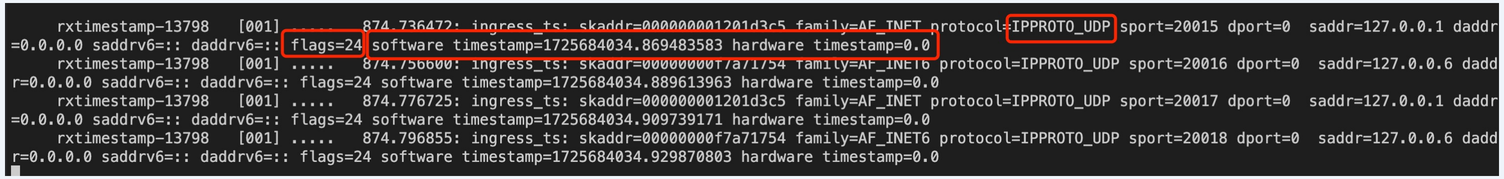
echo 1 > events/timestamp/egress_ts/enable
./txtimestamp -4 -L -C 127.0.0.1



```
.0.0 saddrv6=:: daddrv6=:: flags=1168 type=SCM_TSTAMP_SCHED key=2 software timestamp=1725683348.215543472 hardware timestamp=0.0
     txtimestamp-10269   [001] .....   188.136792: egress_ts: skaddr=00000000ae2aa77d family=AF_INET protocol=IPPROTO_UDP sport=35814 dport=0 saddr=0.0.0.0 daddr=0.0
.0.0 saddrv6=:: daddrv6=:: flags=1168 type=SCM_TSTAMP_SND key=2 software timestamp=1725683348.215544455 hardware timestamp=0.0
     txtimestamp-10269   [001] .....   188.186895: egress_ts: skaddr=00000000ae2aa77d family=AF_INET protocol=IPPROTO_UDP sport=35814 dport=0 saddr=0.0.0.0 daddr=0.0
.0.0 saddrv6=:: daddrv6=:: flags=1168 type=SCM_TSTAMP_SCHED key=3 software timestamp=1725683348.265674478 hardware timestamp=0.0
     txtimestamp-10269   [001] .....   188.186919: egress_ts: skaddr=00000000ae2aa77d family=AF_INET protocol=IPPROTO_UDP sport=35814 dport=0 saddr=0.0.0.0 daddr=0.0
.0.0 saddrv6=:: daddrv6=:: flags=1168 type=SCM_TSTAMP_SND key=3 software timestamp=1725683348.265675234 hardware timestamp=0.0
         chronyd-850     [008] .....   205.100491: egress_ts: skaddr=000000006c7c2403 family=AF_INET protocol=IPPROTO_UDP sport=37091 dport=123 saddr=9.134.0.2 daddr
=10.59.218.197 saddrv6=:: daddrv6=:: flags=1050 type=SCM_TSTAMP_SND key=0 software timestamp=1725683365.229472933 hardware timestamp=0.0
         chronyd-850     [008] .....   205.628916: egress_ts: skaddr=000000006c7c2403 family=AF_INET protocol=IPPROTO_UDP sport=53251 dport=123 saddr=9.134.0.2 daddr
=9.22.139.140 saddrv6=:: daddrv6=:: flags=1050 type=SCM_TSTAMP_SND key=0 software timestamp=1725683365.757903559 hardware timestamp=0.0
         chronyd-850     [008] .....   205.944415: egress_ts: skaddr=000000006c7c2403 family=AF_INET protocol=IPPROTO_UDP sport=36417 dport=123 saddr=9.134.0.2 daddr
=169.254.0.79 saddrv6=:: daddrv6=:: flags=1050 type=SCM_TSTAMP_SND key=0 software timestamp=1725683366.73407101 hardware timestamp=0.0
         chronyd-850     [008] .....   206.395168: egress_ts: skaddr=000000006c7c2403 family=AF_INET protocol=IPPROTO_UDP sport=38969 dport=123 saddr=9.134.0.2 daddr
=169.254.0.81 saddrv6=:: daddrv6=:: flags=1050 type=SCM_TSTAMP_SND key=0 software timestamp=1725683366.524161546 hardware timestamp=0.0
         chronyd-850     [008] .....   207.047665: egress_ts: skaddr=000000006c7c2403 family=AF_INET protocol=IPPROTO_UDP sport=50609 dport=123 saddr=9.134.0.2 daddr
=169.254.0.80 saddrv6=:: daddrv6=:: flags=1050 type=SCM_TSTAMP_SND key=0 software timestamp=1725683367.176663993 hardware timestamp=0.0
           <...>-10311   [010] .....   212.754488: egress_ts: skaddr=00000000e599c505 family=AF_INET protocol=IPPROTO_TCP sport=52684 dport=9000 saddr=127.0.0.1 dadd
r=127.0.0.1 saddrv6=:: daddrv6=:: flags=1168 type=SCM_TSTAMP_SND key=9 software timestamp=1725683372.833351038 hardware timestamp=0.0
     txtimestamp-10311   [010] .....   212.804645: egress_ts: skaddr=00000000e599c505 family=AF_INET protocol=IPPROTO_TCP sport=52684 dport=9000 saddr=127.0.0.1 dadd
r=127.0.0.1 saddrv6=:: daddrv6=:: flags=1168 type=SCM_TSTAMP_SND key=19 software timestamp=1725683372.883585660 hardware timestamp=0.0
     txtimestamp-10311   [010] .....   212.854878: egress_ts: skaddr=00000000e599c505 family=AF_INET protocol=IPPROTO_TCP sport=52684 dport=9000 saddr=127.0.0.1 dadd
```

## RX monitor

echo 1 > events/timestamp/ingress_ts/enable
./rxtimestamp --udp



```
     rxtimestamp-13798   [001] .....   874.736472: ingress_ts: skaddr=000000001201d3c5 family=AF_INET protocol=IPPROTO_UDP sport=20015 dport=0  saddr=127.0.0.1 daddr
=0.0.0.0 saddrv6=:: daddrv6=:: flags=24 software timestamp=1725684034.869483583 hardware timestamp=0.0
     rxtimestamp-13798   [001] .....   874.756600: ingress_ts: skaddr=00000000f7a71754 family=AF_INET6 protocol=IPPROTO_UDP sport=20016 dport=0  saddr=127.0.0.6 dadd
r=0.0.0.0 saddrv6=:: daddrv6=:: flags=24 software timestamp=1725684034.889613963 hardware timestamp=0.0
     rxtimestamp-13798   [001] .....   874.776725: ingress_ts: skaddr=000000001201d3c5 family=AF_INET protocol=IPPROTO_UDP sport=20017 dport=0  saddr=127.0.0.1 daddr
=0.0.0.0 saddrv6=:: daddrv6=:: flags=24 software timestamp=1725684034.909739171 hardware timestamp=0.0
     rxtimestamp-13798   [001] .....   874.796855: ingress_ts: skaddr=00000000f7a71754 family=AF_INET6 protocol=IPPROTO_UDP sport=20018 dport=0  saddr=127.0.0.6 dadd
r=0.0.0.0 saddrv6=:: daddrv6=:: flags=24 software timestamp=1725684034.929870803 hardware timestamp=0.0
```

# 7.5 tracepoint (OPT_STATS)

TencentOS

```
// generation phase
@@ -3983,6 +3983,8 @@ struct sk_buff *tcp_get_timestamping_opt_stats(const struct sock *sk,
(……)
+       trace_egress_ts_with_opt_stat(sk, orig_skb); // sk is the key
+

// report phase
@@ -983,9 +983,11 @@ void __sock_recv_timestamp(struct msghdr *msg, struct sock *sk,
                    trace_egress_ts(sk, &tss, &(SKB_EXT_ERR(skb)->ee));

             if (skb_is_err_queue(skb) && skb->len &&
-                SKB_EXT_ERR(skb)->opt_stats)
+                SKB_EXT_ERR(skb)->opt_stats) {
                    put_cmsg(msg, SOL_SOCKET, SCM_TIMESTAMPING_OPT_STATS,
                            skb->len, skb->data);
+                   trace_egress_ts_with_opt_stat(sk, skb); // skb->data is the key
+             }
```

## A few notes:

- We need to re-implement tcp_get_timestamping_opt_stats() during the generation phase or the report phase. The latter is not very easy for we have to parse skb->data..
- It can be replaced by hooking tcp_get_timestamping_opt_stats() with bpf tools
- This tracepoint is not that necessary, which is only useful for the users who expect to see the output from trace_pipe directly

# 7.6 tracepoint (conclusion)

Is the tracepoint method practical?

- Too much noise, like chronyd ☹
- Limited size of trace_pipe ☹
- Possible cumulative skbs issue 😐
- A fine-grained solution ☺
- Very easy to use ☺
- Good news is that tracepoint is not uAPI ☺
- like kprobe
  - Require _no_ modifications in apps except setsockopt ☺
  - Admins get benefits ☺
  - Easily extend OPT_STAT ☺

# 7.7 tracepoint+OPT_OUTPUT (theory)

- We can take advantage of the setsockopt with tracepoint here, then let the users decide if we expect to see the data through printing to trace_pipe.

- If socket does not have new option flag, it will behave exactly as before, which means printing cmsgs in apps.

- setsockopt is per socket controlled while trace_pipe is controlled by admins. Is it reasonable?

# 7.8 tracepoint+OPT_OUTPUT (code snippets)

```
// kernel
@@ -32,8 +32,9 @@ enum {
        SOF_TIMESTAMPING_OPT_TX_SWHW = (1<<14),
        SOF_TIMESTAMPING_BIND_PHC = (1 << 15),
        SOF_TIMESTAMPING_OPT_ID_TCP = (1 << 16),
+        SOF_TIMESTAMPING_OPT_OUTPUT = (1 << 17),

@@ -2295,7 +2295,8 @@ void tcp_recv_timestamp(struct msghdr *msg, const struct sock *sk,
                else
                        put_cmsg_scm_timestamping(msg, tss);

-               trace_ingress_ts(sk, tss);
+                if (READ_ONCE(sk->sk_tsflags) & SOF_TIMESTAMPING_OPT_OUTPUT)
+                        trace_ingress_ts(sk, tss);


@@ -5424,9 +5424,10 @@ static void __skb_complete_tx_timestamp(struct sk_buff *skb,

        if (err)
                kfree_skb(skb);
-        else if (tsflags & SOF_TIMESTAMPING_SOFTWARE ||
-                 (tsflags & SOF_TIMESTAMPING_RAW_HARDWARE &&
-                 skb_hwtstamps(skb)->hwtstamp))
+        else if ((READ_ONCE(sk->sk_tsflags) & SOF_TIMESTAMPING_OPT_OUTPUT) &&
+                  (tsflags & SOF_TIMESTAMPING_SOFTWARE ||
+                   (tsflags & SOF_TIMESTAMPING_RAW_HARDWARE &&
+                    skb_hwtstamps(skb)->hwtstamp)))
```

```
// kernel
@@ -977,7 +977,7 @@ void __sock_recv_timestamp(struct msghdr
*msg, struct sock *sk,
                else
                        put_cmsg_scm_timestamping(msg, &tss);

-               if (!skb_is_err_queue(skb))
+               if (tsflags & SOF_TIMESTAMPING_OPT_OUTPUT
&& !skb_is_err_queue(skb))
                        trace_ingress_ts(sk, &tss);
```

```
==================================================
|| // txtimestamp.c
|| @@ -563,7 +563,8 @@ static void do_test(int family, unsigned int
|| report_opt)
||
||        sock_opt = SOF_TIMESTAMPING_SOFTWARE |
||                   SOF_TIMESTAMPING_OPT_CMSG |
||-                  SOF_TIMESTAMPING_OPT_ID;
||+                  SOF_TIMESTAMPING_OPT_ID |
||+                  SOF_TIMESTAMPING_OPT_OUTPUT;
==================================================
```

# 7.9 tracepoint+OPT_OUTPUT (conclusion)

Is the tracepoint method practical?

- Possible cumulative skbs issue 😐
- Avoid too much noise (best effort) ☺
- like tracepoint
    - A fine-grained solution ☺
    - Very easy to use ☺
    - Good news is that tracepoint is not uAPI
    - Require no modifications in apps except setsockopt ☺
    - Admins get benefits ☺
    - Easily extend OPT_STAT ☺

# 7.10 tracepoint+OPT_OUTPUT_ONLY (theory)

- Based on the OPT_OUTPUT, we can go further and disable the generation feature thoroughly
- _Only_ allow reading timestamp from trace_pipe by admins.
- Avoid previous generation failure due to shortage of memory
- In order to save more cycles because no more skb allocation and so on

# 7.11 tracepoint+OPT_OUTPUT_ONLY (pseudo code)

```
@@ -33,8 +33,9 @@ enum {
        SOF_TIMESTAMPING_BIND_PHC = (1 << 15),
        SOF_TIMESTAMPING_OPT_ID_TCP = (1 << 16),
        SOF_TIMESTAMPING_OPT_OUTPUT = (1 << 17),
+       SOF_TIMESTAMPING_OPT_OUTPUT_ONLY = (1 << 18),

+static void tstamp_report_direct(struct sock *sk, int type
+                              struct skb_shared_hwtstamps *hwtstamps)
+{
+
+       if (tsonly && (tsflags & SOF_TIMESTAMPING_OPT_STATS) && sk_is_tcp(sk)) {
+               trace_egress_ts_stat(...);
+
+       if (hwtstamps)
+               *skb_trace_egress_ts_only(...,hwtstamps,...);
+       else
+               *skb_trace_egress_ts_only(...,ktime_get_real(),...);
+}

@@ -5489,6 +5505,11 @@ void __skb_tstamp_tx(struct sk_buff *orig_skb,
        if (!skb_may_tx_timestamp(sk, tsonly))
                return;

+       if (READ_ONCE(sk->sk_tsflags) & SOF_TIMESTAMPING_OPT_OUTPUT) {
+               tstamp_report_direct(sk, tstype, hwtstamps);
+               return;
+       }
```

A few notes:

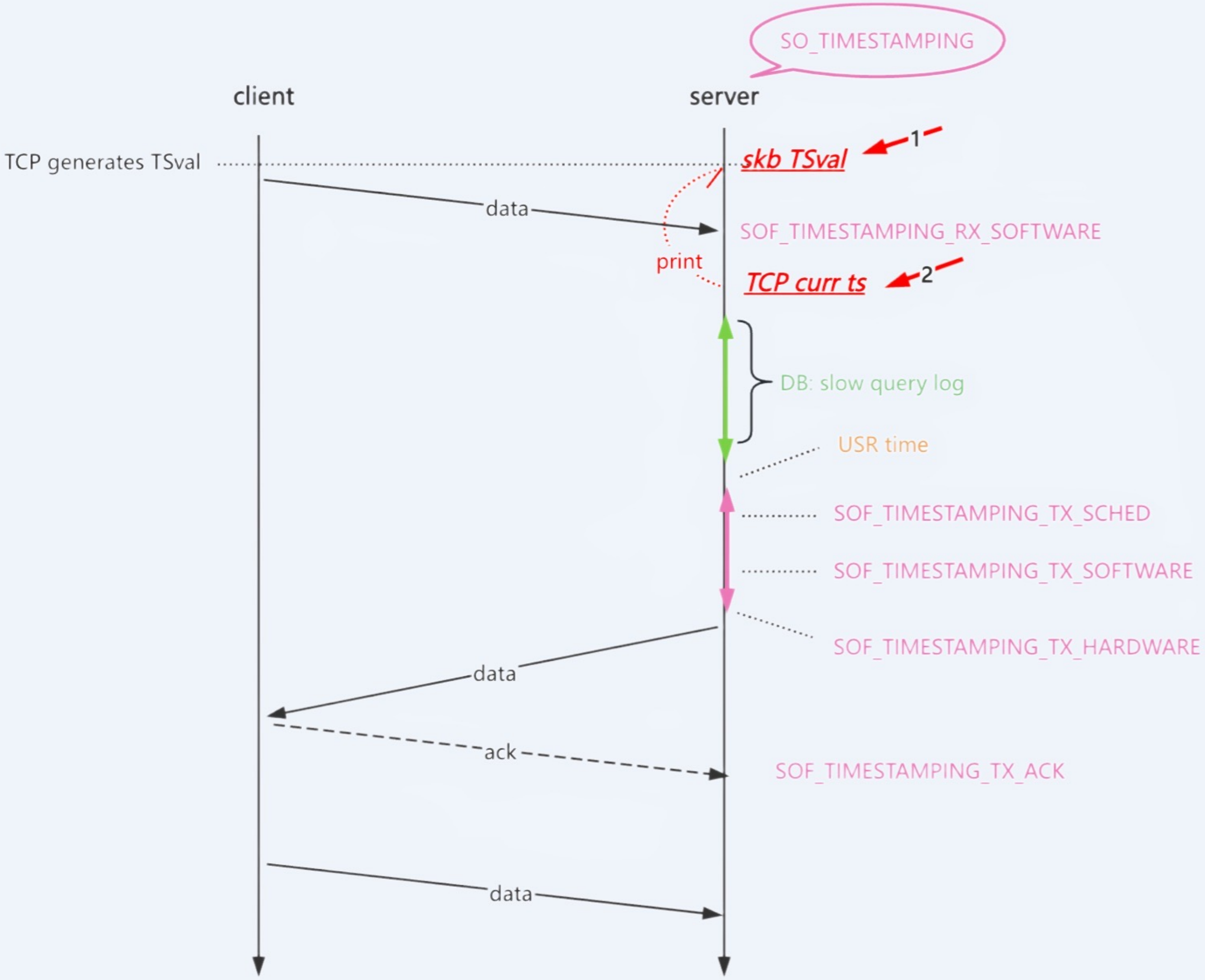
- We need to avoid skb related operations
  - Memory check
  - Allocation
  - Add it into erruque
- Similar to __skb_tstamp_tx()
- We only need to handle egress path

# 7.12 tracepoint+OPT_OUTPUT_ONLY (conclusion)

Is the tracepoint+setsockopt method practical?

- Break the previous design of timestamp feature due to deprecating errqueue 😐
- Save more cycles when generating timestamp skbs ☺
- Solve the cumulative skbs issue ☺
- like tracepoint + OPT_OUTPUT
  - Avoid too much noise (best effort) ☺
  - A fine-grained solution ☺
  - Very easy to use ☺
  - Good news is that tracepoint is not uAPI
  - Require no modifications in apps except setsockopt ☺
  - Admins get benefits ☺
  - Easily extend OPT_STAT ☺

# 8.1 Adding recv tracepoints in TCP



How to get more useful info in rx path?

- Like red arrow 1, can we make a best guess about the other side?
  - NO? At least very complicated, because every ts includes a tsoffset (see tcp_v4_init_ts_off())
  - By mapping the initial tsval in SYN and the one in SYN ACK which are recorded, so when the data arrives we parse the tsval option and then get the relative time in the local machine.
  - Then, we roughly infer the elapsed time.

- Like red arrow 2, we can print more current ts in the key functions
  - It could be implemented easily based on prior tracepoint* method.
  - so that we can know the time consumed between RX_SOFTWARE and TCP stack.

# 8.2 Adding xmit tracepoints in TCP

Is creating the TCP snapshot too late?

- Timestamp is recorded in __dev_queue_xmit(), which means we probably miss the best chance to do the TCP snapshot, because the final skb traverses through TCP IP layer already.

- How about creating the snapshot when TCP stops sending skb in tcp_write_xmit(), say, due to nagle policy?

# 8.3 Adding more fine-grained tracepoints

Further more, can we hook more functions?

In order to narrow down the scope. If we have already learned that the high latency happens somewhere like from userspace->dev xmit, what could we do next?

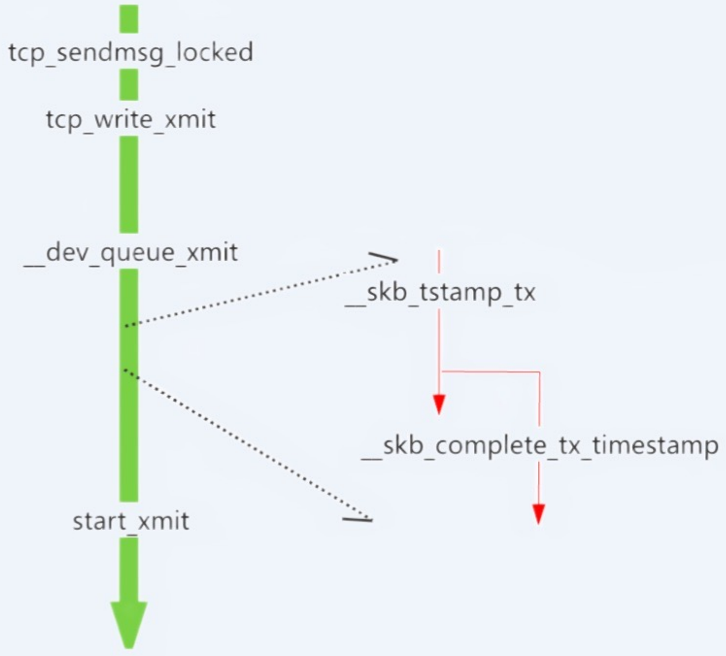
Here are a few possible high latency chances:

- Userspace timestamp> SCHED timestamp
  - tcp_write_xmit() delays due to nagle or internal pacing
  - netfilter delays due to possible external hook functions

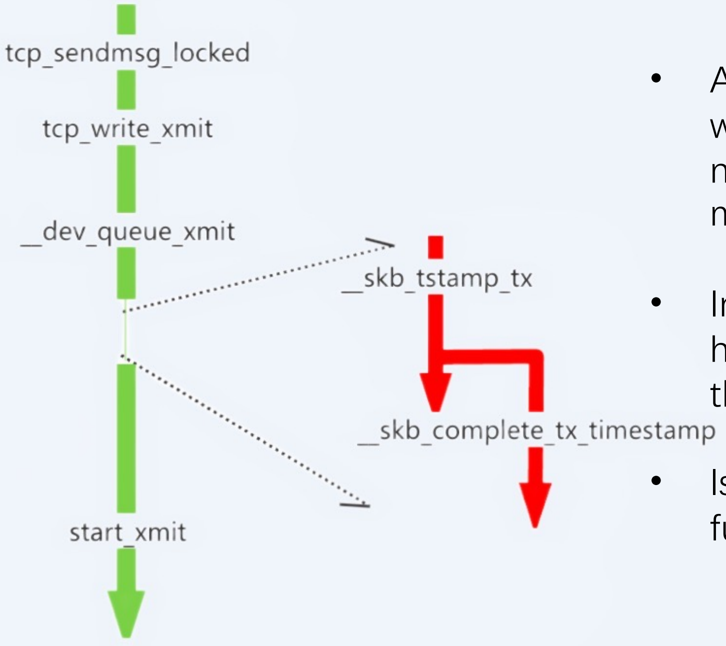
- RX SOFTWARE timestamp -> ?
  - process backlog delays due to ksoftirqd
  - receive stack runs slowly due to many ptype_all
  - tcp backlog delays due to the socket lock held by users

- Writing the bpf related programs to trace every functions in the hot path seems clumsy, but we have no other alternatives. Is it possible in the future with setting SO_TIMESTAMPING?

# 9. Future?



Nowadays



Future?

## If all the apps rely on the feature?

- At present, we only make sure the sockets with SO_TIMESTAMPING feature enabled do not interfere with others, which outperforms many bpf-based track tools.

- In the future, if all the sockets needs it, will it have a huge side effect then? We are facing the issue actually...

- Is it possible that we can trace most of key functions without impacting performance.

# Thank you :) !!!