BPF/netdev implementation upgrade?

Martin Lau

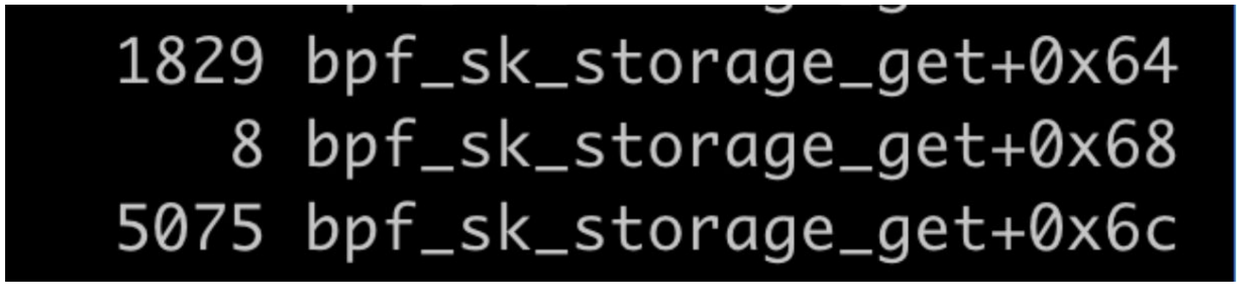
Meta

Agenda

01 bpf_sk_storage_get performance

02 What should we do about the cgroup's bpf_sock_ops?

bpf_sk_storage_get()



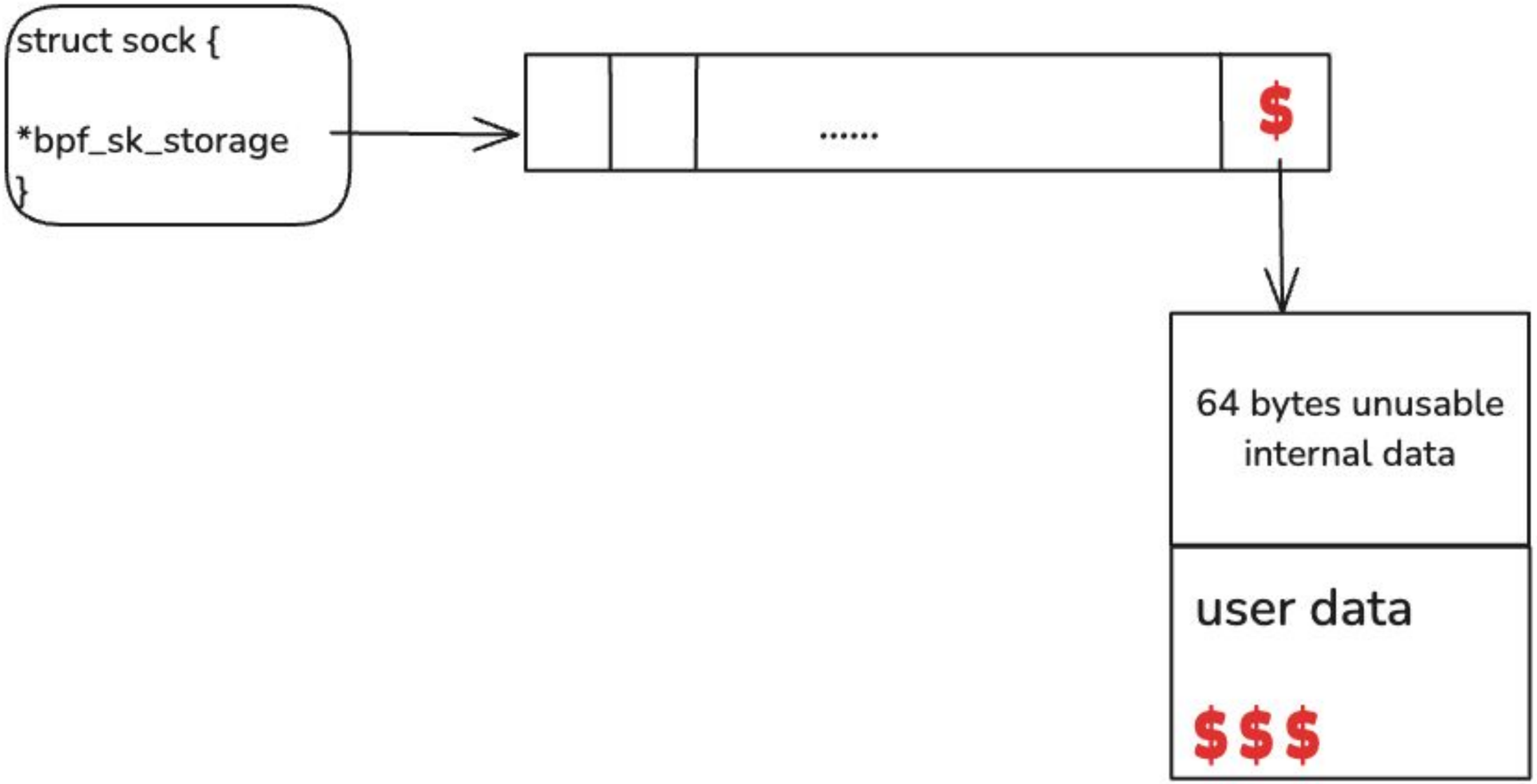
Two instructions are hot from the perf profile

bpf_sk_storage_get()

```
static inline struct bpf_local_storage_data *
bpf_local_storage_lookup(struct bpf_local_storage *local_storage,
                         struct bpf_local_storage_map *smap,
                         bool cacheit_lockit)
{
        /* 1st cache-miss ($) */
        sdata = rcu_dereference_check(local_storage->cache[smap->cache_idx],
                                      bpf_rcu_lock_held());
        /* 2nd cache-miss ($$$) */
        if (sdata && rcu_access_pointer(sdata->smap) == smap)
                return sdata;

        /* … */
}
```

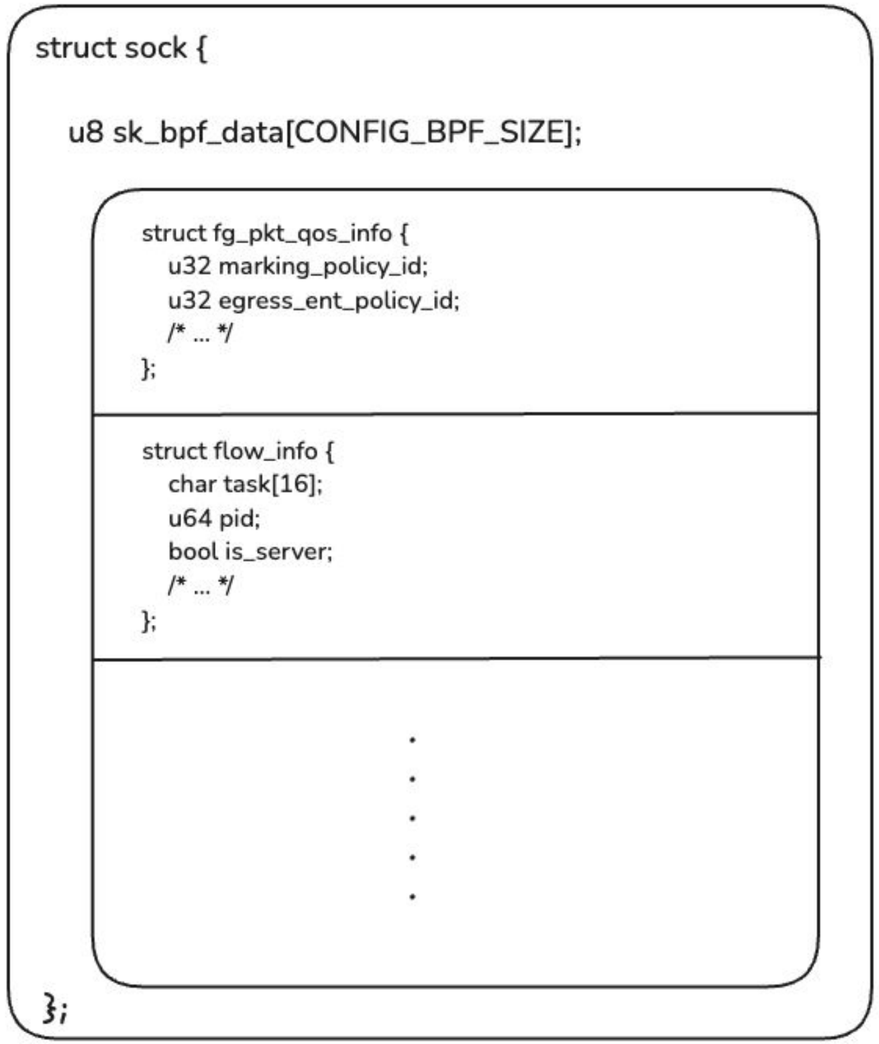
bpf_sk_storage_get()

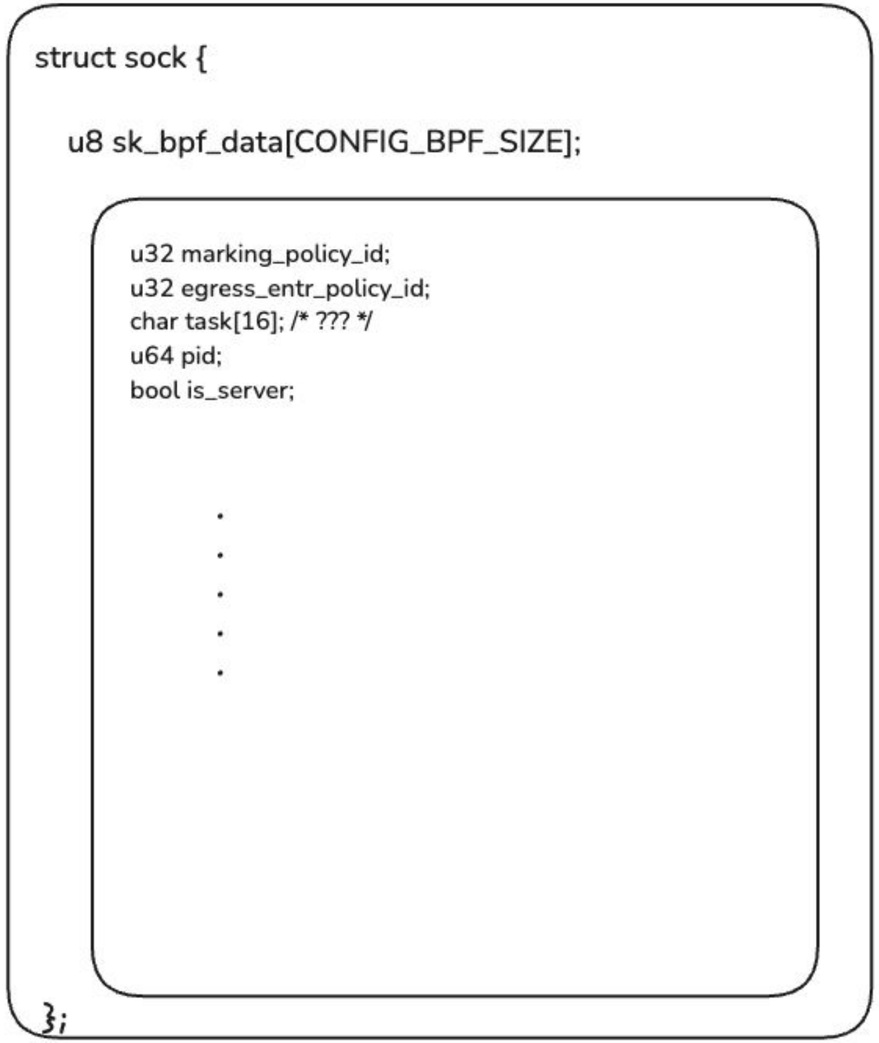


Production usage observations

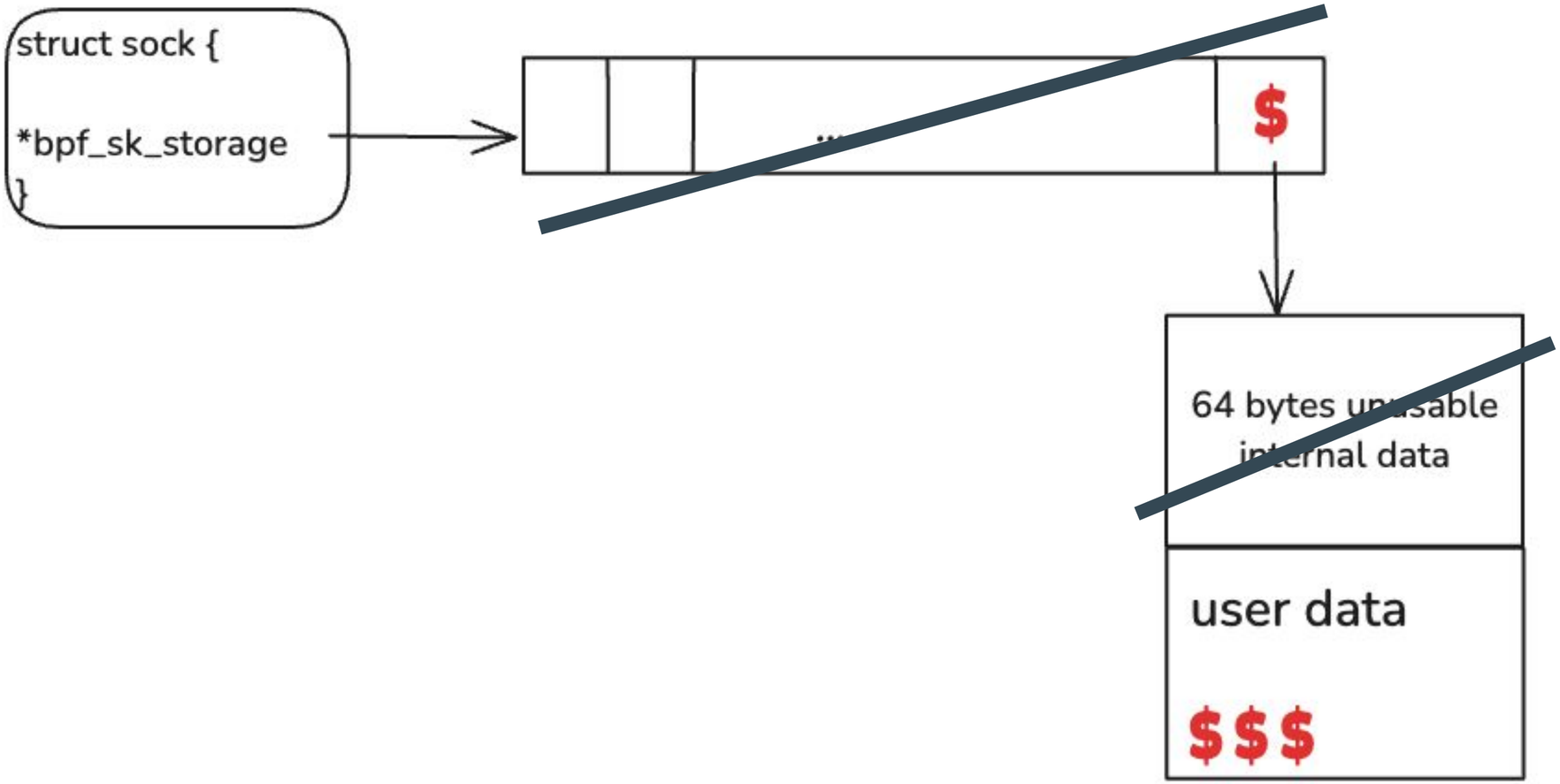
- The map_value’s struct hardly changed
- In one of our maps, it had only one change since 2021
- The storage is needed for all sk(s) in a machine

What if...


struct sock {
u8 sk_bpf_data[CONFIG_BPF_SIZE];
struct fg_pkt_qos_info {
u32 marking_policy_id;
u32 egress_ent_policy_id;
/* ... */
};
struct flow_info {
char task[16];
u64 pid;
bool is_server;
/* ... */
};
};


struct sock {
u8 sk_bpf_data[CONFIG_BPF_SIZE];
u32 marking_policy_id;
u32 egress_entr_policy_id;
char task[16]; /* ??? */
u64 pid;
bool is_server;
};

What if...


struct sock {
*bpf_sk_storage
}
$
64 bytes unusable
internal data
user data
$$$

Ideas...

- When CONFIG_BPF_SIZE is 0, goes back to existing slower path
- cgroup memory accounting is loss.

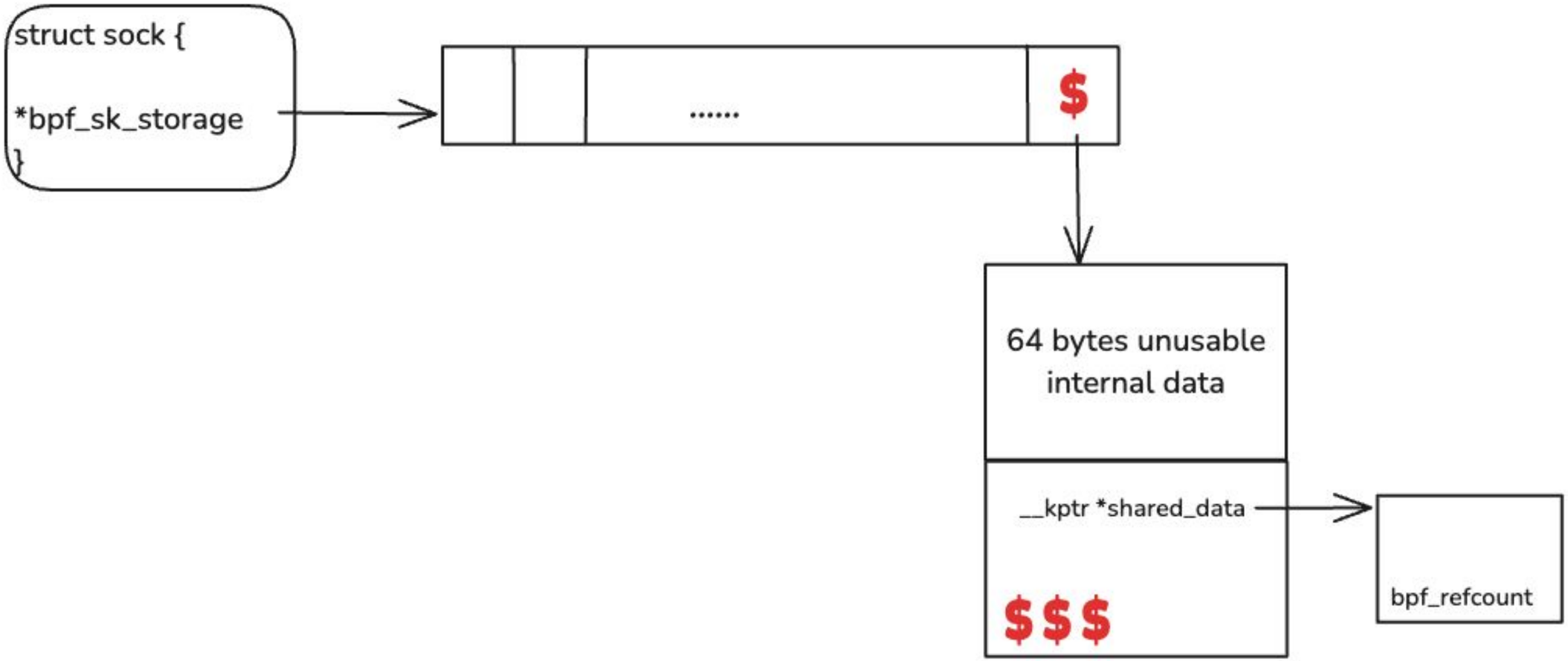
Ideas...

- How to reserve space in the sk_bpf_data?
  - First come first reserve.
  - Once reserved, the data will always be there until the reboot.
- The first 64 bytes internal data is gone.
  - The storage will stay with the sock(s) forever

Sharing sk storage data

- Half a million sockets having individual sk storages.
- The storages at most have 500 different values.
- Only 500 copies of the values are needed instead of half a million values.
- There is a bpf_refcount and bpf_obj_new(), so it should help?

Sharing sk storage data



What should we do with bpf_sock_ops?

```
enum {
        BPF_SOCK_OPS_VOID,
        BPF_SOCK_OPS_TIMEOUT_INIT,
        BPF_SOCK_OPS_RWND_INIT,
        BPF_SOCK_OPS_TCP_CONNECT_CB,
        BPF_SOCK_OPS_ACTIVE_ESTABLISHED_CB,
        BPF_SOCK_OPS_PASSIVE_ESTABLISHED_CB,
        BPF_SOCK_OPS_NEEDS_ECN,
        BPF_SOCK_OPS_BASE_RTT,
        BPF_SOCK_OPS_RTO_CB,
        BPF_SOCK_OPS_RETRANS_CB,
        BPF_SOCK_OPS_STATE_CB,
        BPF_SOCK_OPS_TCP_LISTEN_CB,
        BPF_SOCK_OPS_RTT_CB,
        BPF_SOCK_OPS_PARSE_HDR_OPT_CB,
        BPF_SOCK_OPS_HDR_OPT_LEN_CB,
        BPF_SOCK_OPS_WRITE_HDR_OPT_CB,
        BPF_SOCK_OPS_TSTAMP_SCHED_CB,
        BPF_SOCK_OPS_TSTAMP_SND_SW_CB,
        BPF_SOCK_OPS_TSTAMP_SND_HW_CB,
        BPF_SOCK_OPS_TSTAMP_ACK_CB,
        BPF_SOCK_OPS_TSTAMP_SENDMSG_CB,
};
```

```
struct bpf_tcp_ops  {
        u32 timeout_init(struct tcp_sock *sk);
        u32 rwnd_init(struct tcp_sock *sk);
        /* … */
        u32 rtt_cb(struct tcp_sock *tp);
        u32 parse_hdr(struct tcp_sock *tp, struct sk_buff *skb);
        /* … */
        U32 write_hdr_opt(struct tcp_sock *tp, struct sk_buff *skb);
};

struct bpf_sock_ops {
        void tx_tstamp(int tstatmp_type, struct sk_buff *skb);
        void rx_tstamp(int tstatmp_type, struct sk_buff *skb);
};
```

struct bpf_sock_ops_kern

```
struct bpf_sock_ops_kern {
        struct  sock *sk;
        union {
                u32 args[4];
                u32 reply;
                u32 replylong[4];
        };
        struct sk_buff  *syn_skb;
        struct sk_buff  *skb;
        void    *skb_data_end;
        u8      op;
        u8      is_fullsock;
        u8      is_locked_tcp_sock;
        u8      remaining_opt_len;
        u64     temp;                   /* temp and everything after is not
                                         * initialized to 0 before calling
                                         * the BPF program. New fields that
                                         * should be initialized to 0 should
                                         * be inserted before temp.
                                         * temp is scratch storage used by
                                         * sock_ops_convert_ctx_access
                                         * as temporary storage of a register.
                                         */
};
```

Recent timestamp additions

	enum bpf_sock_ops	**struct_ops**
helpers/kfuncs filtering	runtime by skops->op. The op enum is a runtime value.	Verification time filtering by individual “ops” bpf prog.
kernel ptr access (e.g. sk)	is_valid_access() not useful because op enum is a runtime value. => BPF_INSN rewrite.	BTF. ops is known at the load time.
passing data to bpf prog	A kitchen sink in bpf_sock_ops_kern for all ops	Pass what is needed to the function’s arguments.
return values	Most ops ignore the CGRP_OK (1) and CGRP_EPERM (0). u32 replylong[4]. Not intuitive for bpf prog to use. union with the input args[4]. technically broken when >1 cgrp program running.	The function’s return value. Intuitive to bpf prog.

Discussions...

Q: Is the cgroup specific return value useful?

- The CGRP_OK (1) and CGRP_EPERM (0) are not very useful in sock_ops.
- Most of the ops ignore them. Equivalent to a struct_ops program returning void.
- Whatever done (e.g. bpf_setsockopt, writer_header...) is done. CGRP_OK/EPERM does not matter.

Q: Does it still need ordering?

- Should it be per-cgroup?
- or a global struct_ops ordering infra works for other sub-systems?

Q: Worth moving to struct_ops?


Meta