bpf tracing multi-link

Menglong Dong
Kernel engineer
Net & BPF for 6 years
China Telecom

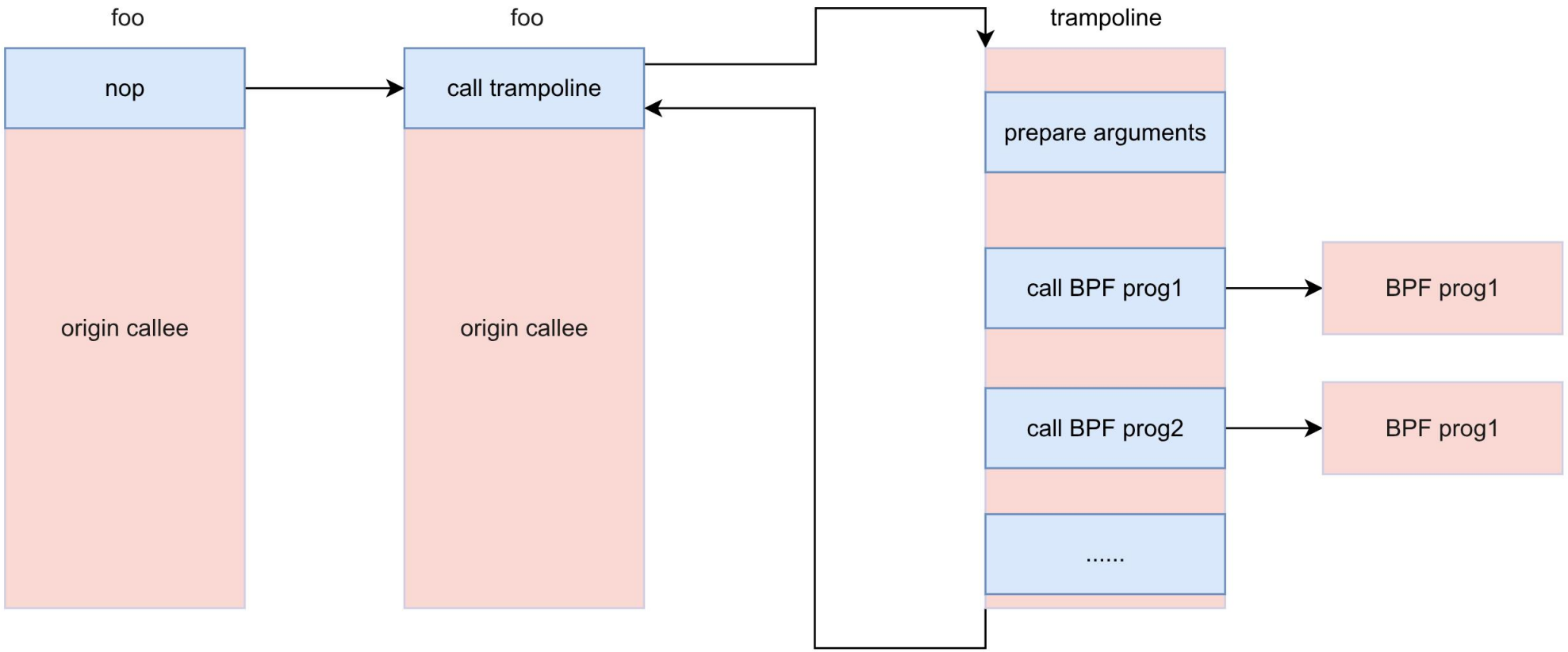
Jiri Olsa
Isovalent at cisco

1. **bpf trampoline**
2. single direct ftrace_ops
3. tracing multi-link
4. verifier & attchment
5. performance

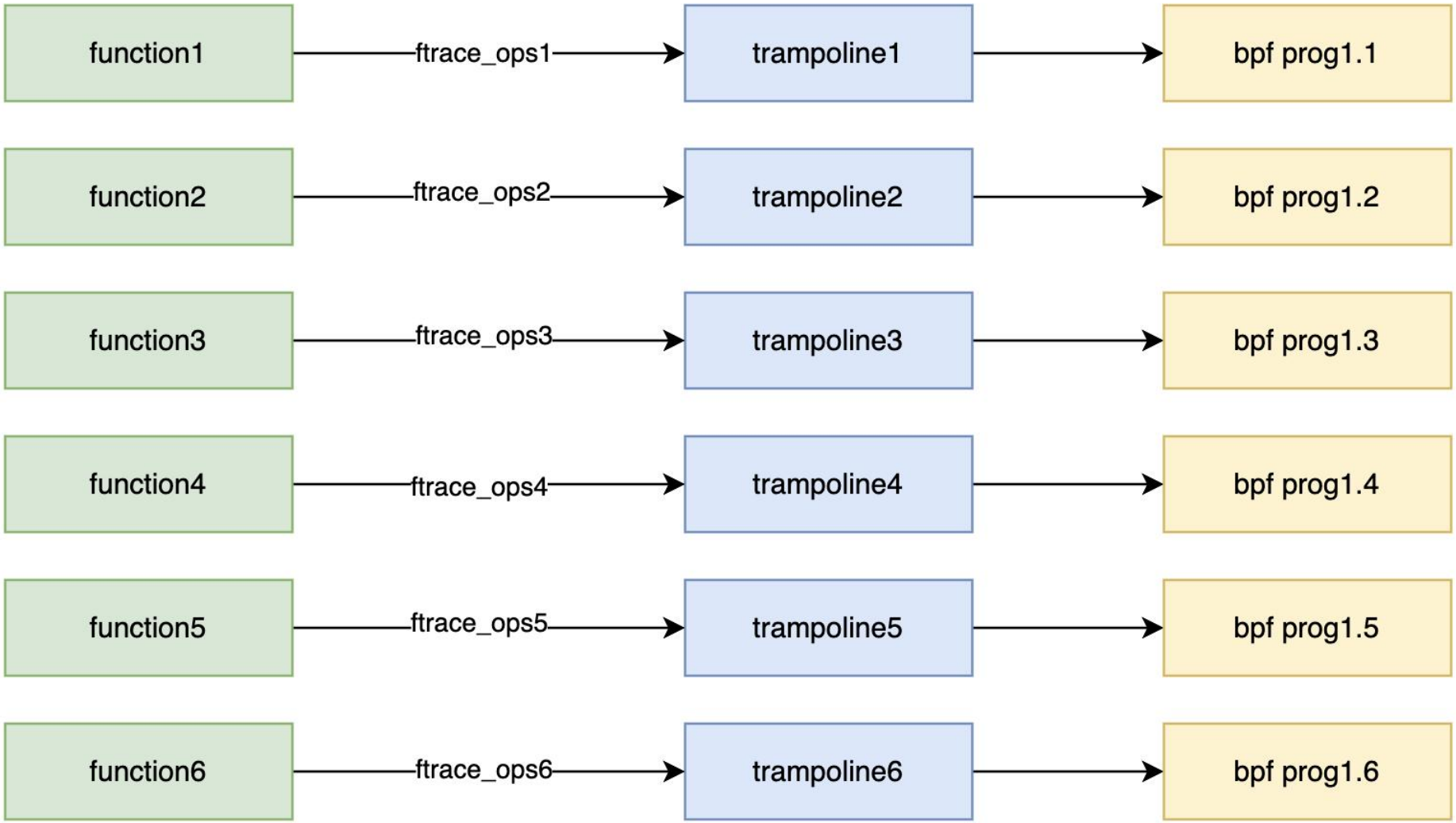
what’s bpf trampoline?

- a bridge between BPF progs and target function
- generated from arch_prepare_bpf_trampoline() according to the target and the BPF progs
- per-function
- call the BPF progs directly
- some metadata is stored in the trampoline instruction, such as function arguments count, function address, etc.

what’s bpf trampoline?



issues of tracing

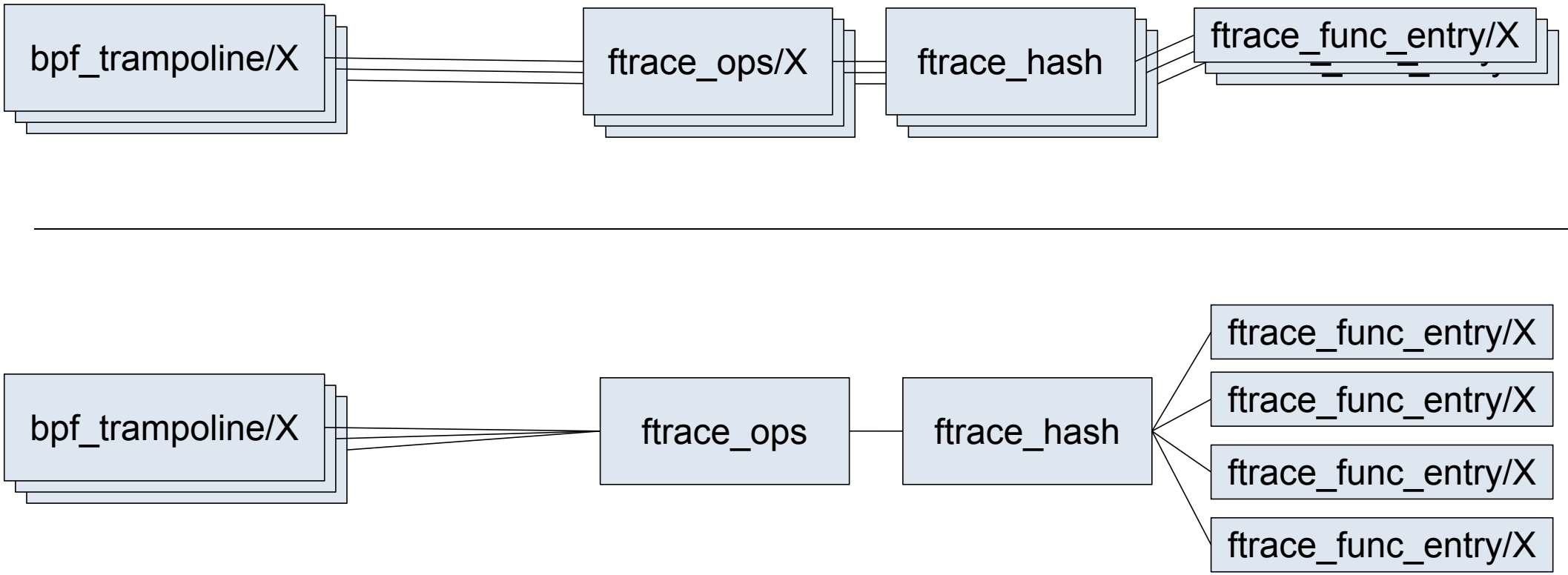


issue of tracing

- to trace 20000 kernel functions, we need: 20000 bpf progs -> 20000 trampoline -> 20000 ftrace_ops
- memory consume: redundant bpf prog instances, trampolines and ftrace_ops
- not convenient
- attaching slow：take about 60s for 1000 functions

1. bpf trampoline
2. **single direct ftrace_ops**
3. tracing multi-link
4. verifier & attchment
5. performance

single direct ftrace_ops



single direct ftrace_ops

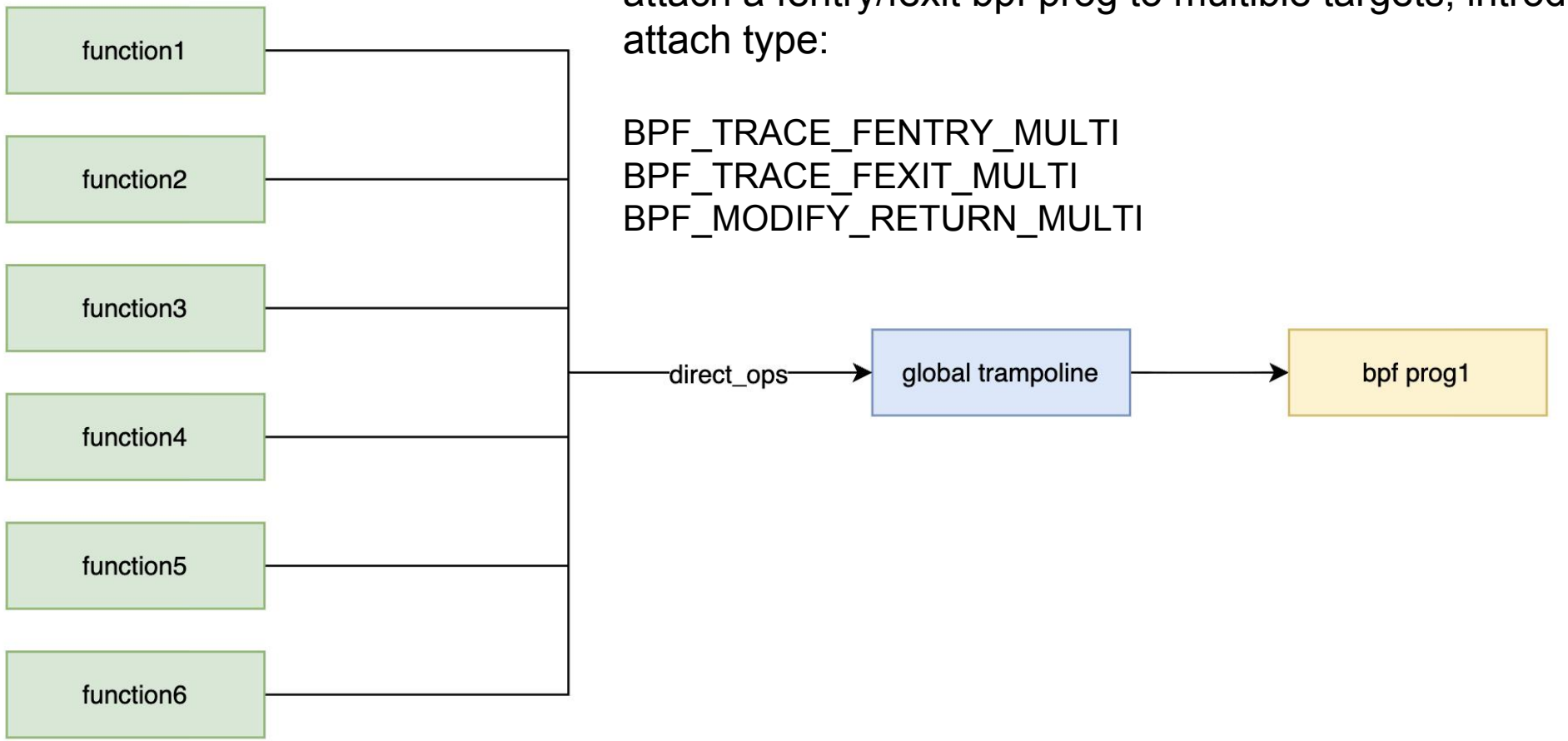
use a single ftrace_ops for all the bpf trampolines

update_ftrace_direct_add(ops, hash)
update_ftrace_direct_del(ops, hash)
update_ftrace_direct_mod(ops, hash)

Time for the command “bpftrace -e ‘fentry:vmlinux:ksys_* {}’ -c true” decrease from 4.416s to 1.666s

1. bpf trampoline
2. single direct ftrace_ops
3. **tracing multi-link**
4. verifier & attchment
5. performance

tracing multi-link



attach a fentry/fexit bpf prog to multible targets, introduce new attach type:

BPF_TRACE_FENTRY_MULTI
BPF_TRACE_FEXIT_MULTI
BPF_MODIFY_RETURN_MULTI

what’s global trampoline?

A function written mostly in C and partly in assembly(arch-specifical, x86 only for now):

save function arguments
md = kfunc_md_get(ip)
prepare the stack(save function argument count, etc)

call all the bpf progs in md->bpf_progs[FENTRY]
return if no fexit

percpu_ref_get(&md->pcref)
call origin
save return value

call all the bpf progs in md->bpf_progs[FEXIT]
percpu_ref_put(&md->pcref)
skip the rip and return to the caller

entry of global trampoline

```
__naked void bpf_global_caller(void)
{
        asm volatile(
                "subq $" stack_size ", %rsp\n"
                SAVE_ARGS
        );
        asm volatile(
                "leaq " __stringify(FUNC_ARGS_1) "(%rsp), %rdi\n"
                "movq " stack_size "(%rsp), %rsi\n"
                "call bpf_global_caller_run\n"
                "test %rax, %rax\n"
                "jne 1f\n"
        );
        asm volatile(
                RESTORE_ARGS
                "addq $" stack_size ", %rsp\n"
                ASM_RET
        );
        asm volatile(
                "1: movq (%rax), %rax\n"
                "addq $(" stack_size " + 8), %rsp\n"
                ASM_RET);
}
```

function metadata

- a rhashtable, the key is function address and the value is struct kfunc_md

```
struct kfunc_md {
        struct rhash_head hlist;
        struct rcu_head rcu;
        unsigned long func;
        struct hlist_head __rcu
                bpf_progs[BPF_TRAMP_MAX];
        struct percpu_ref pcref;
        u16 users;
        bool bpf_origin_call;
        u8 bpf_prog_cnt;
        u8 nr_args;
};
```

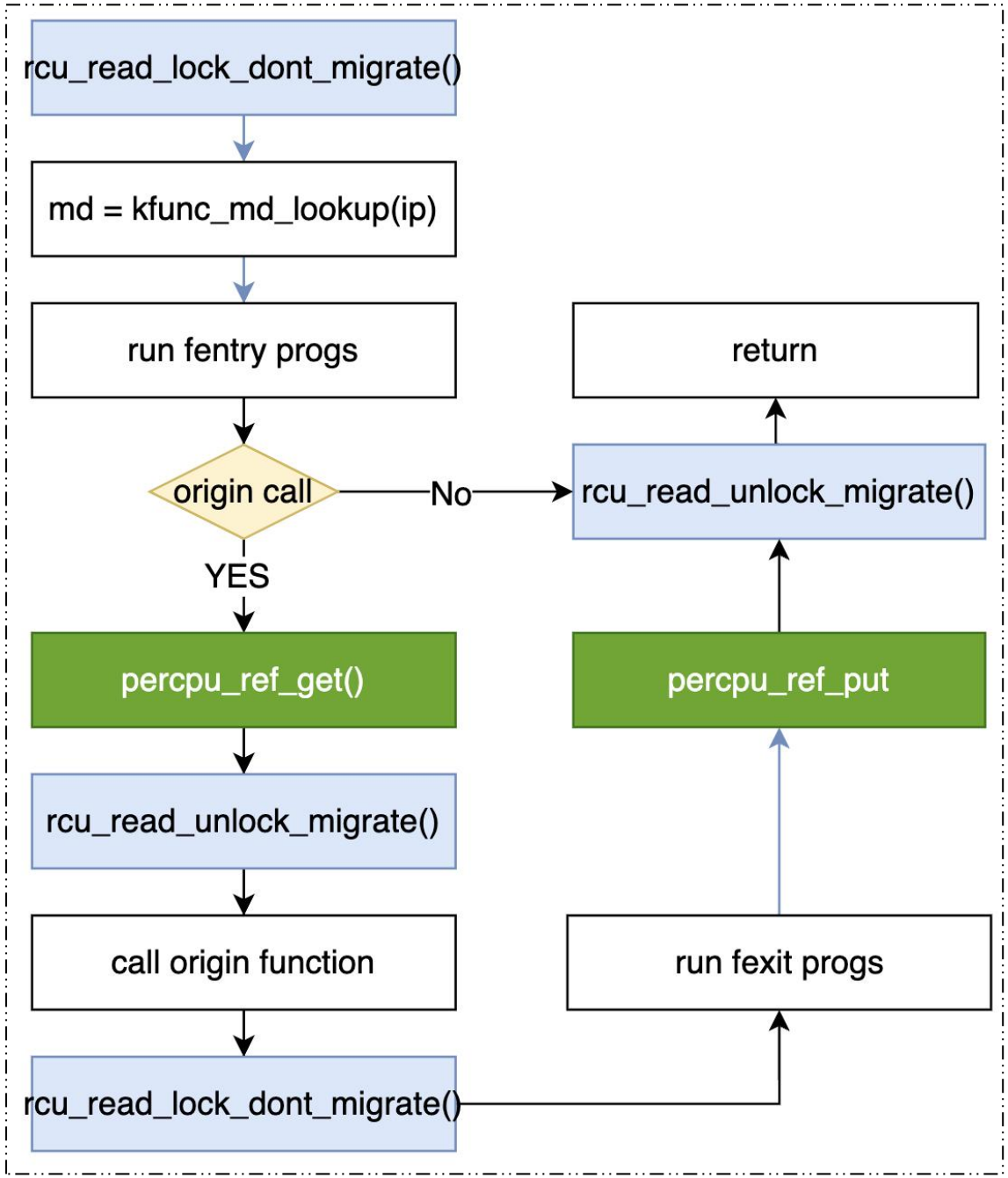
```
struct kfunc_md_tramp_prog {
        struct hlist_node list;
        struct bpf_prog *prog;
        u64 cookie;
        struct rcu_head rcu;
};
```

- kfunc_md_create, kfunc_md_get, kfunc_md_put
- kfunc_md_bpf_link, kfunc_md_bpf_unlink

release of function metadata

- for no origin call case, the use of kfunc_md in global trampoline is protected with rcu_read_lock(), free it directly with **kfree_rcu(&md->rcu)**
- for origin call case, the kfunc_md is protected by both rcu_read_lock and percpu_ref, the step to free is:

  call_rcu(&md->rcu, callback) -> percpu_ref_kill(&md->pcref) -> kfree(md)



1. bpf trampoline
2. single direct ftrace_ops
3. tracing multi-link
4. **verifier & attchment**
5. performance

verifier

- During loading time, no BTF type id is specified. Therefore, the tracing-multi program is not BTF type awared for the function arguments. All the argument we get SCALAR_VALUE, and we need do the cast with bpf_core_cast() manually.
- Access to ctx is not allow, and we need to get the argument with *bpf_get_func_arg().*
- Allow access to ctx? Record the maximum accessed argument, and check all the targets during attaching:
  skb = bpf_core_cast(ctx[0])

```
SEC("fentry.multi/ip_rcv")
int BPF_PROG(ip_rcv)
{
        struct sk_buff *skb = NULL;

        bpf_get_func_arg(ctx, 0, &skb);
        skb = bpf_core_cast(skb);
        return 0;
}
```

attachment

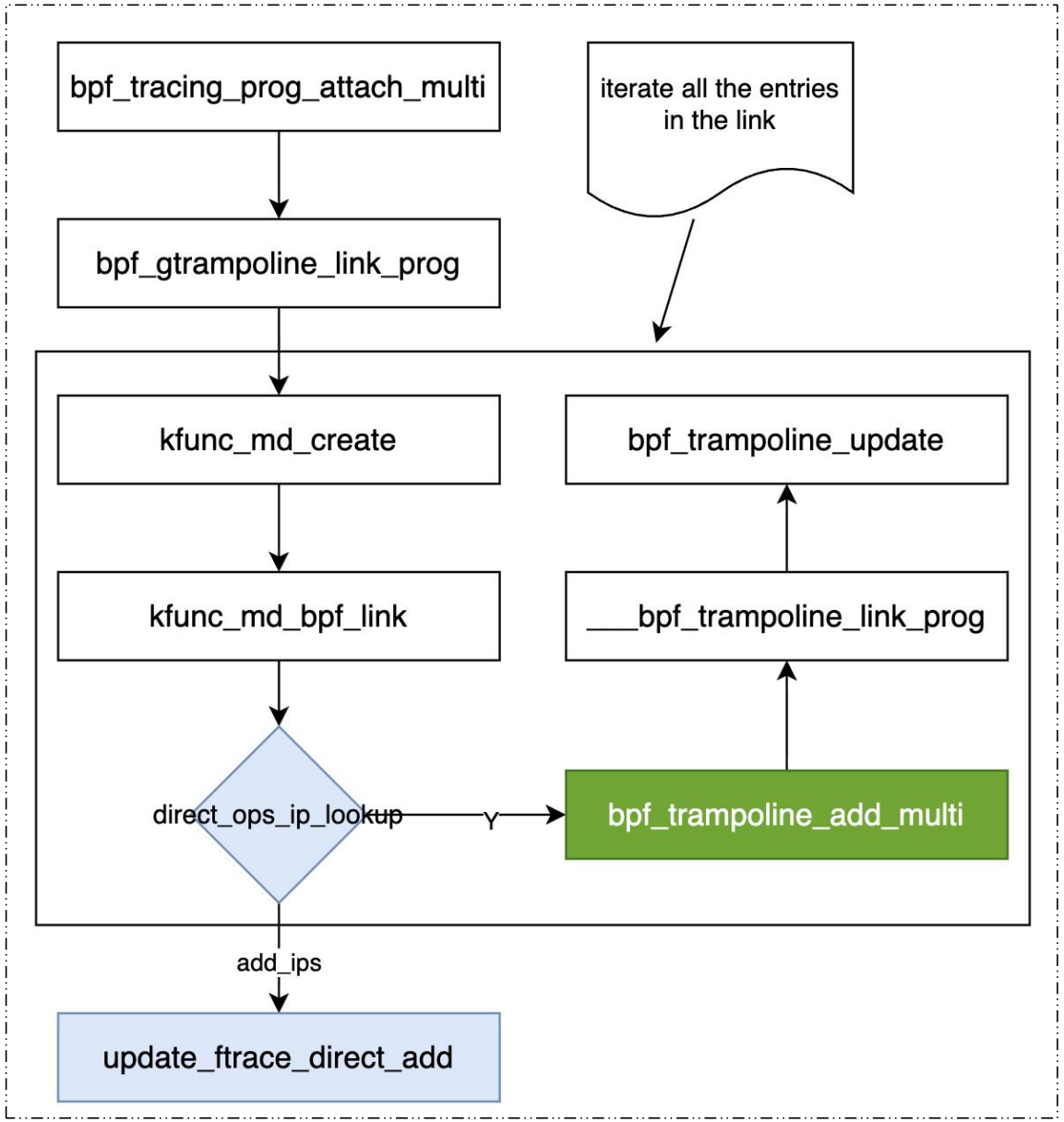
- bpf_tracing_prog_attach_multi:
  prepare the struct bpf_gtramp_link
  according to user arguments, and call
  bpf_gtrampoline_link_prog
- bpf_gtrampoline_link_prog
  kfunc_md_create
  kfunc_md_bpf_link
  update_ftrace_direct_add
- bpf_gtrampoline_unlink_prog
  kfunc_md_put
  kfunc_md_bpf_unlink
  update_ftrace_direct_del

```
struct bpf_gtramp_link_entry {
        struct module *attach_mod;
        struct btf *attach_btf;
        unsigned long ip;
        u64 cookie;
        u32 btf_id;
        u32 nr_args;
};

struct bpf_gtramp_link {
        struct bpf_link link;
        u32 entry_cnt;
        struct bpf_gtramp_link_entry entries[]
__counted_by(entry_cnt);
};
```

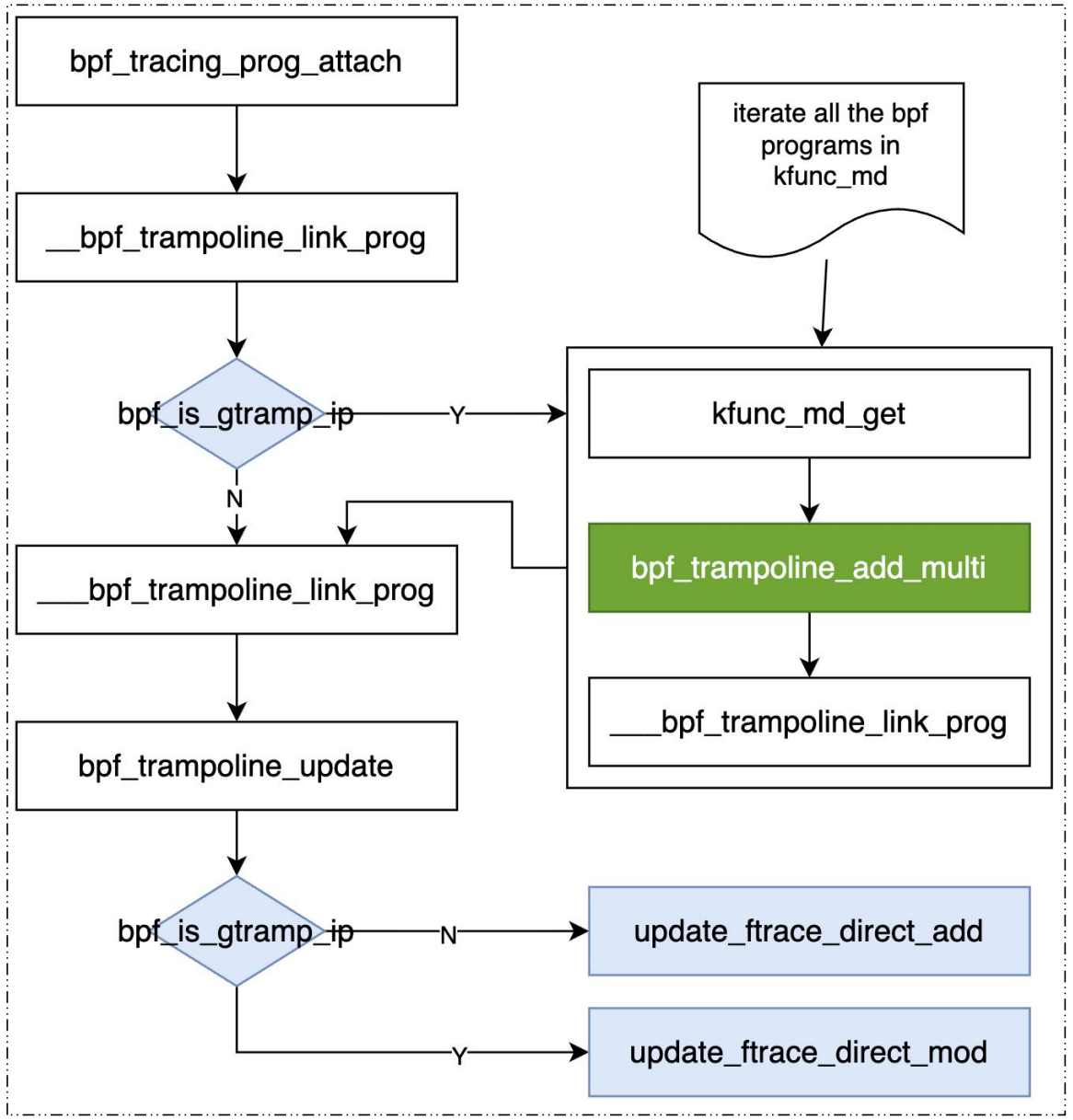
attachment of tracing-multi

- mixture: check if bpf trampoline exist. Allocate a bpf_shim_tramp_link and link it to the trampoline directly.



attachment of tracing

- mixture: for all the bpf programs in kfunc_md, allocate bpf_shim_tramp_link for every one, and add it to the trampoline (without update it).
- bpf_is_gtramp_ip: ftrace_find_rec_direct(ip) == bpf_global_caller



usage example

The usage is similar to kprobe-multi and wildcards is supported:

```
SEC("fentry.multi/bpf_fentry_test*")
int BPF_PROG(fentry_cookie_test)
{
        return 0;
}
```

1. bpf trampoline
2. single direct ftrace_ops
3. tracing multi-link
4. verifier & attchment
5. **performance**

performance of attach

0.328s for 50000 kernel functions

issues

- the maintain of global trampoline, which is partly architecture specifical
- stack arguments is not supported, can’t attached to functions whose arguments count more than 6. Fallback to bpf trampoline(TODO)?
- ipmodify: kfunc_md can’t be found for the ipmodify(livepatch) cast. Fallback to bpf trampoline(TODO)?
- ‘jmp’ mode for fexit is not supported
- performance decrease
  1. the overhead of rhashtable(can be optimized by “**function padding** based hash table” optionally), the main cause
  2. extra memory read(function metadata)
  3. more condition check code

bench trigger performance

usermode-count : 930.223 ± 0.913M/s
kernel-count : 427.750 ± 0.512M/s
syscall-count : 31.799 ± 0.051M/s
fentry : 162.859 ± 0.483M/s
fexit : 80.086 ± 0.066M/s
fmodret : 85.933 ± 0.066M/s
fentry-multi : 123.506 ± 0.637M/s (-25%)
fexit-multi : 74.695 ± 0.147M/s (-7%)
fmodret-multi : 79.104 ± 0.142M/s (-8%)
rawtp : 196.124 ± 0.306M/s
tp : 85.600 ± 0.063M/s
kprobe : 61.080 ± 0.070M/s
kprobe-multi : 63.278 ± 0.042M/s
kprobe-multi-all: 5.074 ± 0.005M/s
kretprobe : 23.936 ± 0.032M/s
kretprobe-multi: 28.505 ± 0.014M/s
kretprobe-multi-all: 4.117 ± 0.005M/s

tracing multi-link：

https://lore.kernel.org/netdev/20250303065345.229298-1-dongml2@chinatelecom.cn/

https://github.com/menglongdong/bpf/tree/tracing-multi-link-single-ops

single direct ftrace_ops:

https://lore.kernel.org/bpf/20251203082402.78816-1-jolsa@kernel.org/

function padding based metadata:

https://lore.kernel.org/bpf/20250226121537.752241-1-dongml2@chinatelecom.cn/