Real World Debugging with eBPF

Zhichuan Liang

SRECON ASIA PACIFIC

SINGAPORE
14–16 June, 2023

Self Introduction

- 2014 ~ 2015: Civil engineer@Rwanda
- 2016 ~ 2022: Python@Beijing, Golang@Singapore
- 2023 ~ 20\d\d: eBPF@Isovalent

ISOVALENT

Agenda

Start from 2 issues from open source community

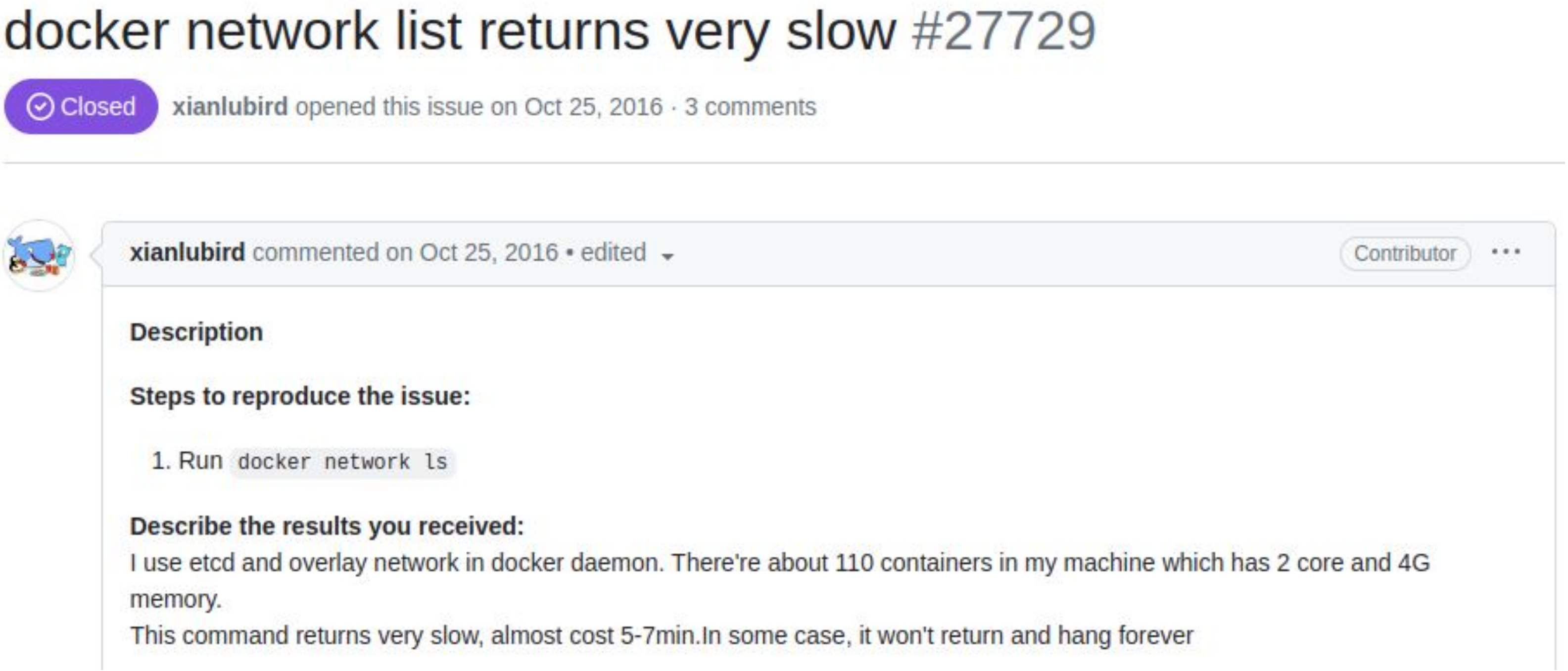
- Docker issue #27729
- Cilium issue #14222

We'll discuss

- eBPF vs traditional debugging approaches
- bpftrace for Golang tracing (amd64)

docker

https://github.com/moby/moby/issues/27729



docker network list returns very slow #27729

xianlubird opened this issue on Oct 25, 2016 · 3 comments

xianlubird commented on Oct 25, 2016 • edited

Description

Steps to reproduce the issue:

1. Run `docker network ls`

Describe the results you received:
I use etcd and overlay network in docker daemon. There're about 110 containers in my machine which has 2 core and 4G memory.
This command returns very slow, almost cost 5-7min.In some case, it won't return and hang forever

Reproduce & Narrow down

Reproducing: same build, similar env, time(1) command

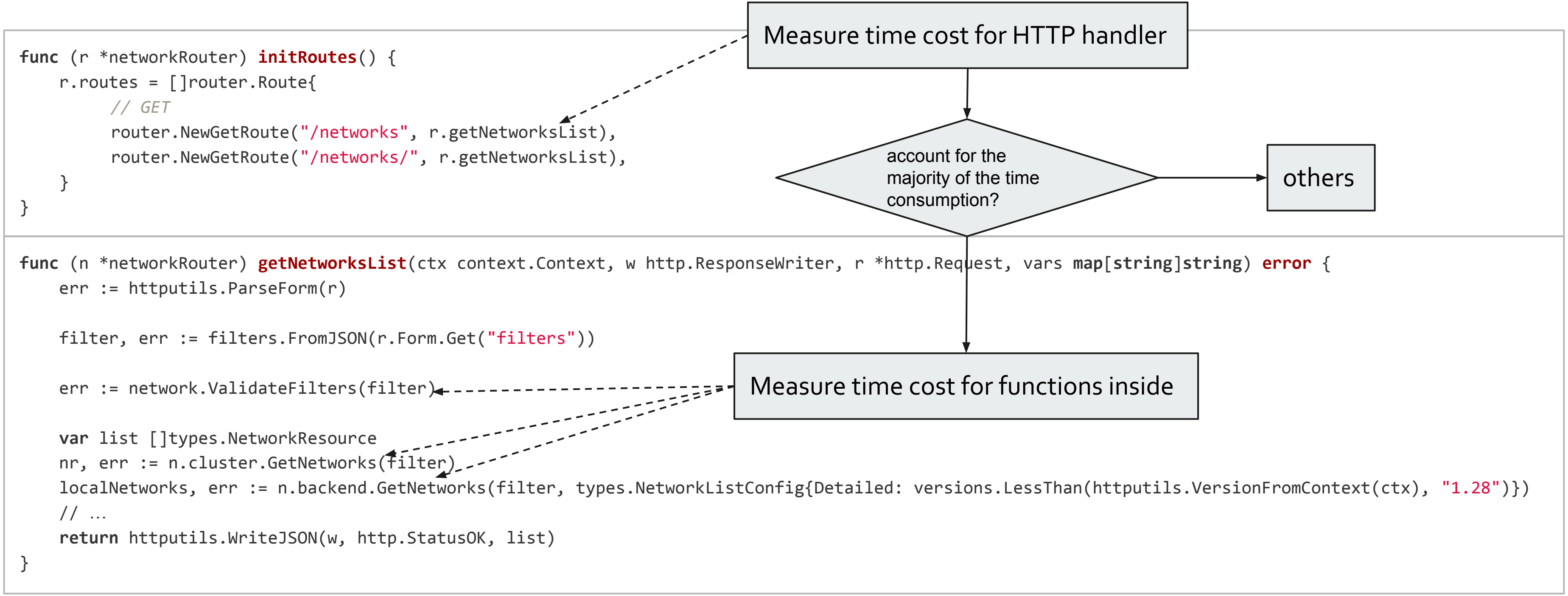
```
$ time ./docker network ls &>/dev/null

real    0m3.707s
user    0m0.005s
sys     0m0.025s
```

Narrowing down: client side or server side

```
$ time curl localhost:2376/v1.23/networks &>/dev/null

real    0m3.661s
user    0m0.007s
sys     0m0.007s
```

Debugging Ideas

Goal: to identify the functions costing most time



```go
func (r *networkRouter) initRoutes() {
    r.routes = []router.Route{
        // GET
        router.NewGetRoute("/networks", r.getNetworksList),
        router.NewGetRoute("/networks/", r.getNetworksList),
    }
}
```

```go
func (n *networkRouter) getNetworksList(ctx context.Context, w http.ResponseWriter, r *http.Request, vars map[string]string) error {
    err := httputils.ParseForm(r)

    filter, err := filters.FromJSON(r.Form.Get("filters"))

    err := network.ValidateFilters(filter)

    var list []types.NetworkResource
    nr, err := n.cluster.GetNetworks(filter)
    localNetworks, err := n.backend.GetNetworks(filter, types.NetworkListConfig{Detailed: versions.LessThan(httputils.VersionFromContext(ctx), "1.28")})
    // …
    return httputils.WriteJSON(w, http.StatusOK, list)
}
```

Measure time cost for HTTP handler
account for the majority of the time consumption?
others
Measure time cost for functions inside

Silver bullet: printf

Idea: record start time at entry, calculate elapsed time at exit

```
func (n *networkRouter) getNetworksList(ctx context.Context, w http.ResponseWriter, r *http.Request, vars map[string]string) error {
+   startAt := time.Now()
+   defer func() {
+       fmt.Println(time.Since(startAt))
+   }()

    err := httputils.ParseForm(r)

    filter, err := filters.FromJSON(r.Form.Get("filters"))

    err := network.ValidateFilters(filter)

    var list []types.NetworkResource
    nr, err := n.cluster.GetNetworks(filter)
    localNetworks, err := n.backend.GetNetworks(filter, types.NetworkListConfig{Detailed: versions.LessThan(httputils.VersionFromContext(ctx), "1.28")})
    // …
    return httputils.WriteJSON(w, http.StatusOK, list)
}
```

Problems:

- Repeatedly restart the running process: unsuitable for production troubleshooting

Traditional debugger: GDB

Idea: add breakpoints at entry and exit

Step 1: find the entry breakpoint: symbol name

```
(gdb) file /usr/bin/dockerd
Reading symbols from /usr/bin/dockerd...

(gdb) info functions getNetworksList
File /go/src/github.com/docker/docker/api/server/router/network/network_routes.go:
    void github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList;
```

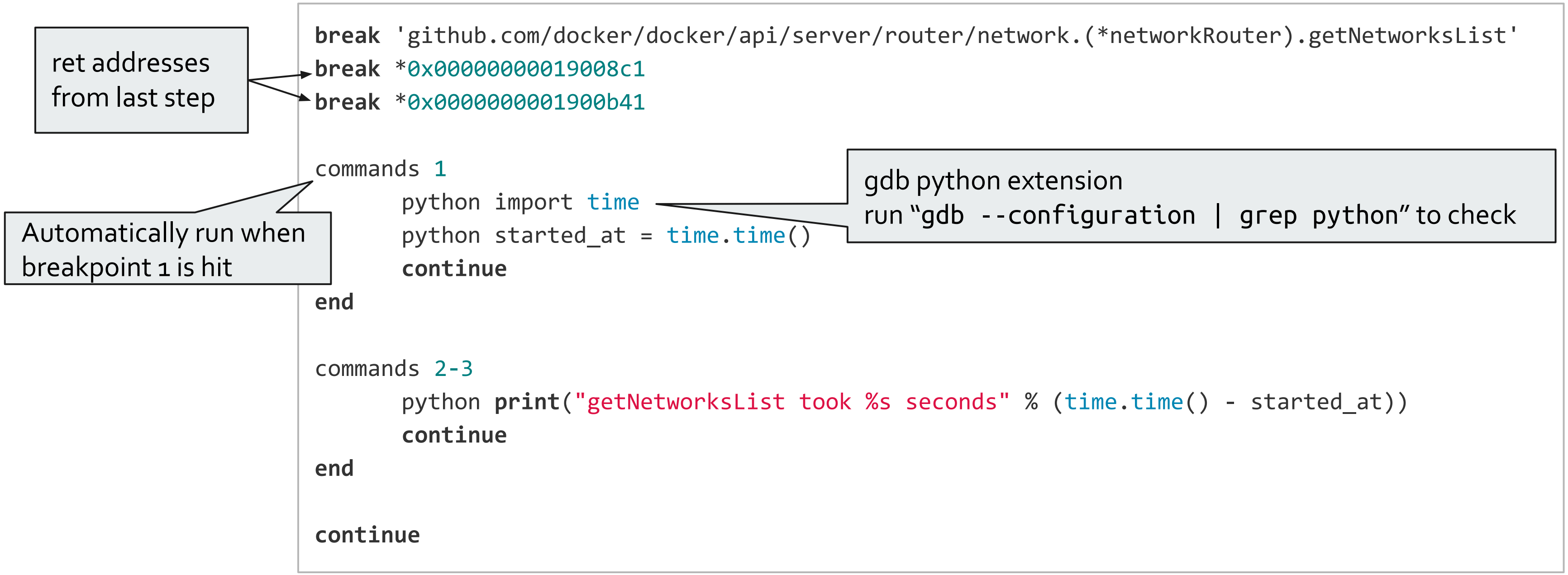
Step 2: find the exit breakpoint: ret address

```
(gdb) disassemble 'github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList'
Dump of assembler code for function
github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList:
   0x0000000001900780 <+0>:    lea     -0x340(%rsp),%r12
   0x0000000001900788 <+8>:    cmp     0x10(%r14),%r12
...
   0x00000000019008c1 <+321>:   ret
...
   0x0000000001900b41 <+961>:   ret
```

ret addresses, could be more than one

Traditional debugger: GDB

Step 3: write a gdb script



ret addresses from last step

Automatically run when breakpoint 1 is hit

gdb python extension
run “gdb --configuration | grep python” to check

```
break 'github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList'
break *0x00000000019008c1
break *0x0000000001900b41

commands 1
        python import time
        python started_at = time.time()
        continue
end

commands 2-3
        python print("getNetworksList took %s seconds" % (time.time() - started_at))
        continue
end

continue
```

Problems: performance overhead

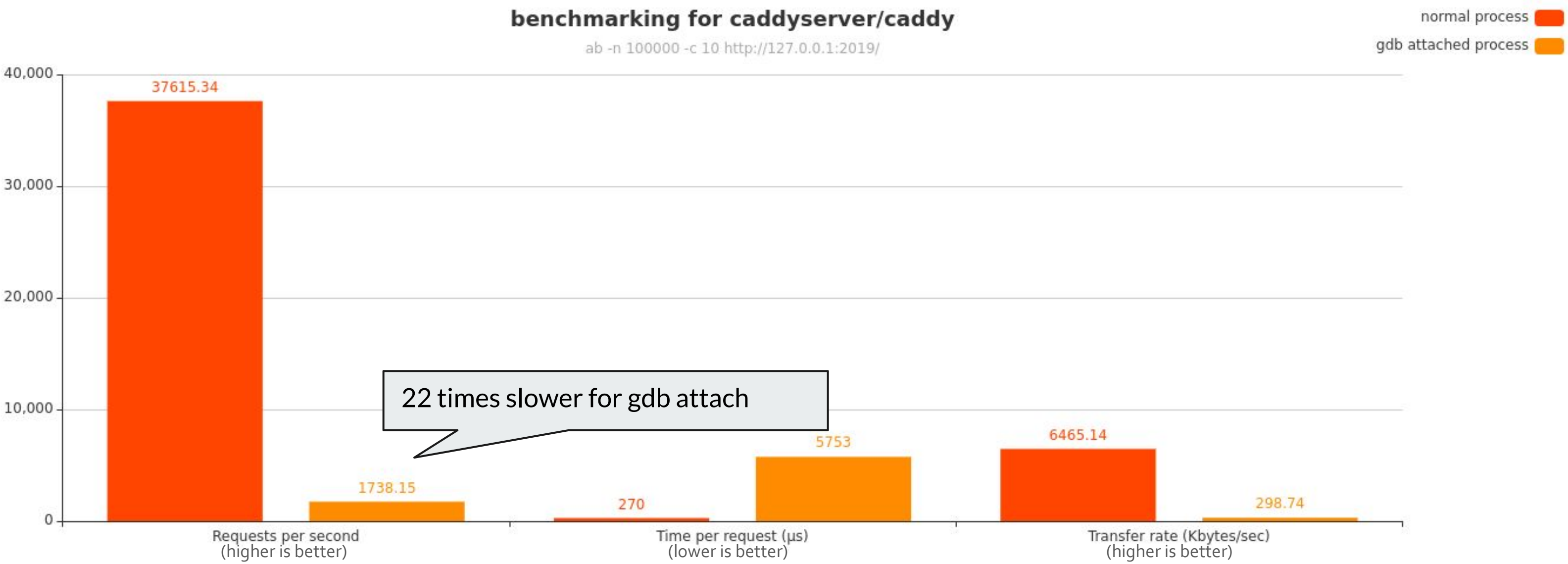
Benchmark for GDB

```
break 'github.com/caddyserver/caddy/v2.(*adminHandler).ServeHTTP'
commands
    continue
end
continue
```

Do nothing, avoid slow python dragging down

```
gdb -x bench.gdb -p $(pidof caddy) &> /dev/null
```

Redirect, avoid slow stdio dragging down



22 times slower for gdb attach

Internal tool: pprof

pprof/profile

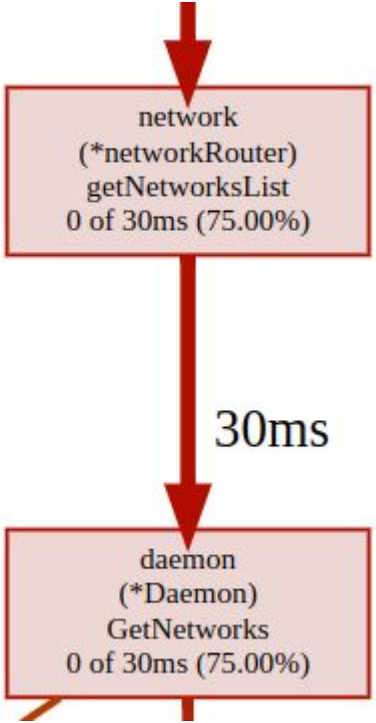
- `go tool pprof localhost:2376/debug/pprof/profile?seconds=30`
- pprof/profile measures on-CPU time and doesn't account for off-CPU time

pprof/block:

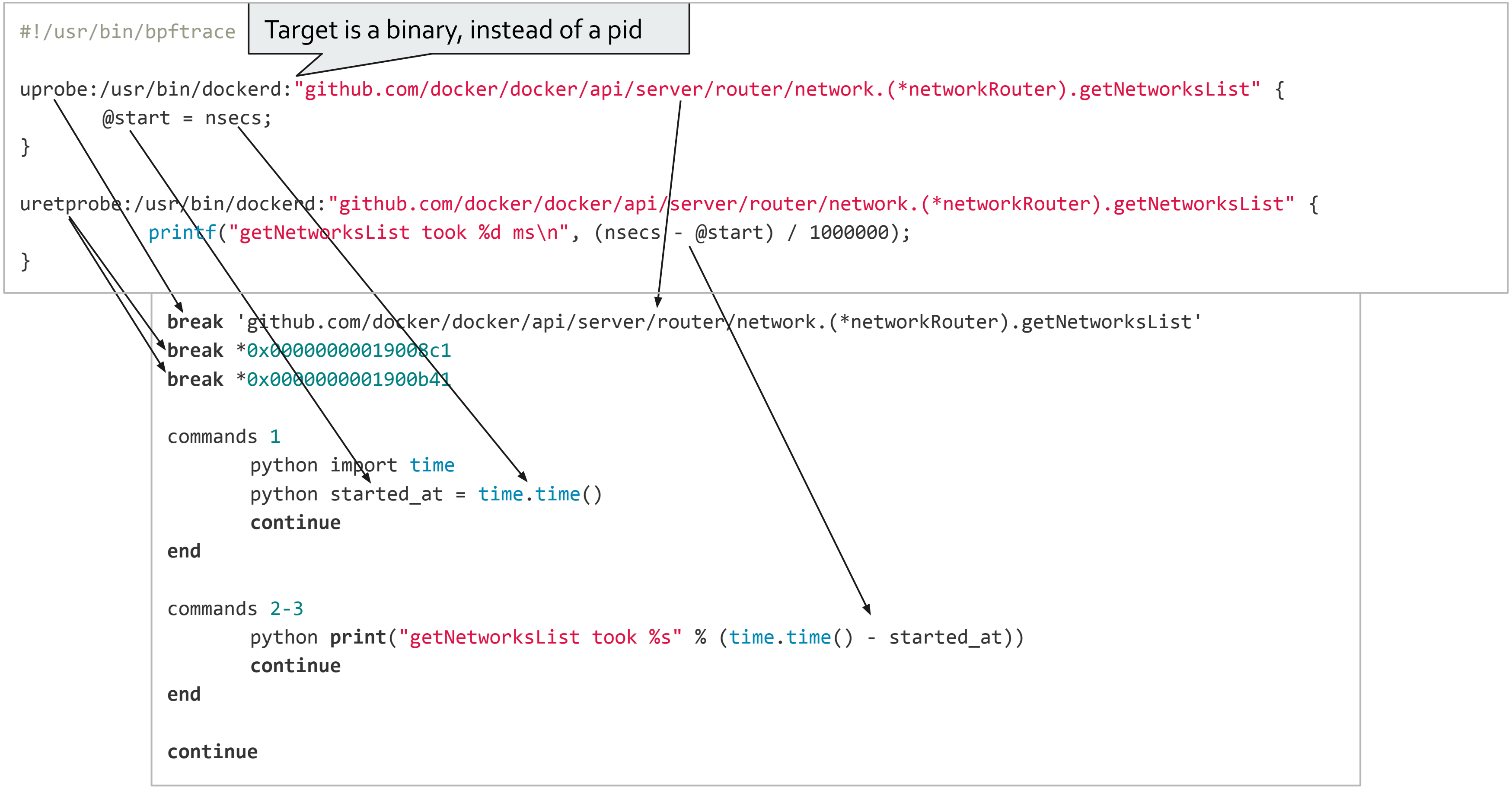
- `go tool pprof localhost:2376/debug/pprof/block?seconds=10`
- Result includes time cost for
  - select
  - chan send / receive
  - Mutex.Lock, WaitGroup.Wait
  - Cond.Wait
- Result doesn't include time cost for
  - syscalls (network IO / file IO)
  - Blocking in cgo calls
  - time.Sleep

Problems:

- pprof may not be turned on
- pprof doesn't accurately reflect the consumption of real time



eBPF: bpftrace script

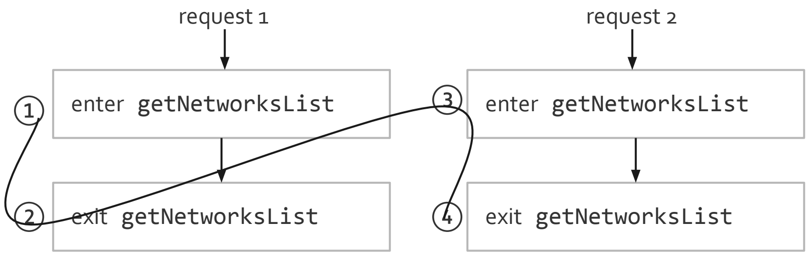


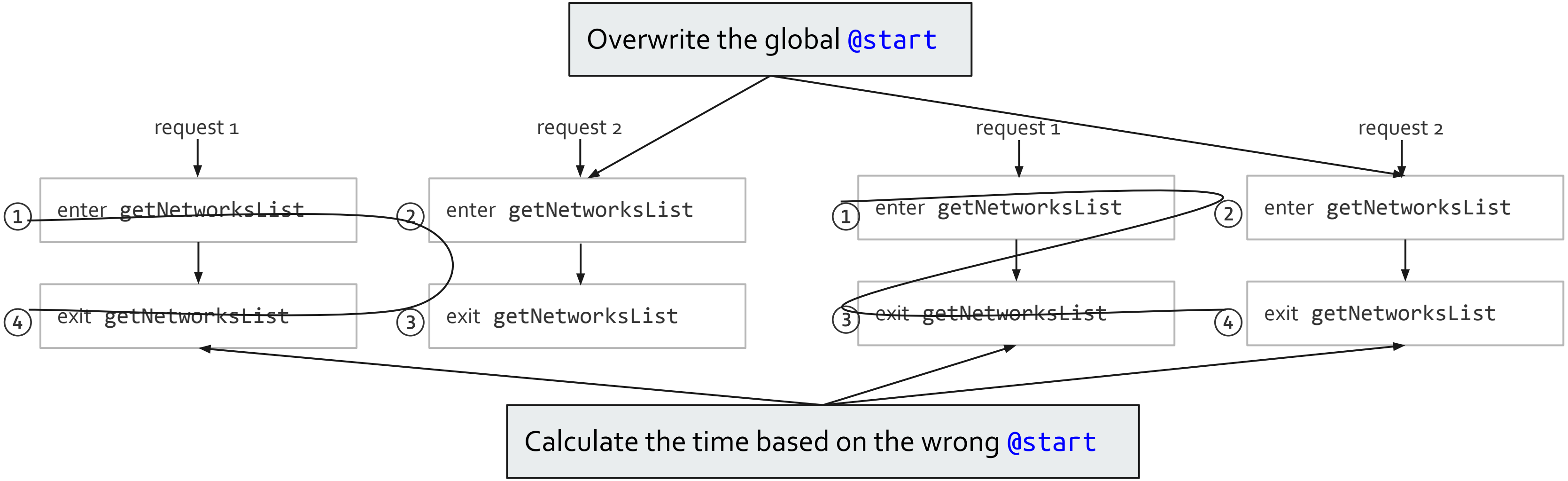
Target is a binary, instead of a pid

```
#!/usr/bin/bpftrace

uprobe:/usr/bin/dockerd:"github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList" {
        @start = nsecs;
}

uretprobe:/usr/bin/dockerd:"github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList" {
            printf("getNetworksList took %d ms\n", (nsecs - @start) / 1000000);
}
```

```
break 'github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList'
break *0x00000000019008c1
break *0x0000000001900b41

commands 1
        python import time
        python started_at = time.time()
        continue
end

commands 2-3
        python print("getNetworksList took %s" % (time.time() - started_at))
        continue
end

continue
```

Concurrency problem





Solution to concurrency problem

Idea: don’t use global variable, store the start time per thread

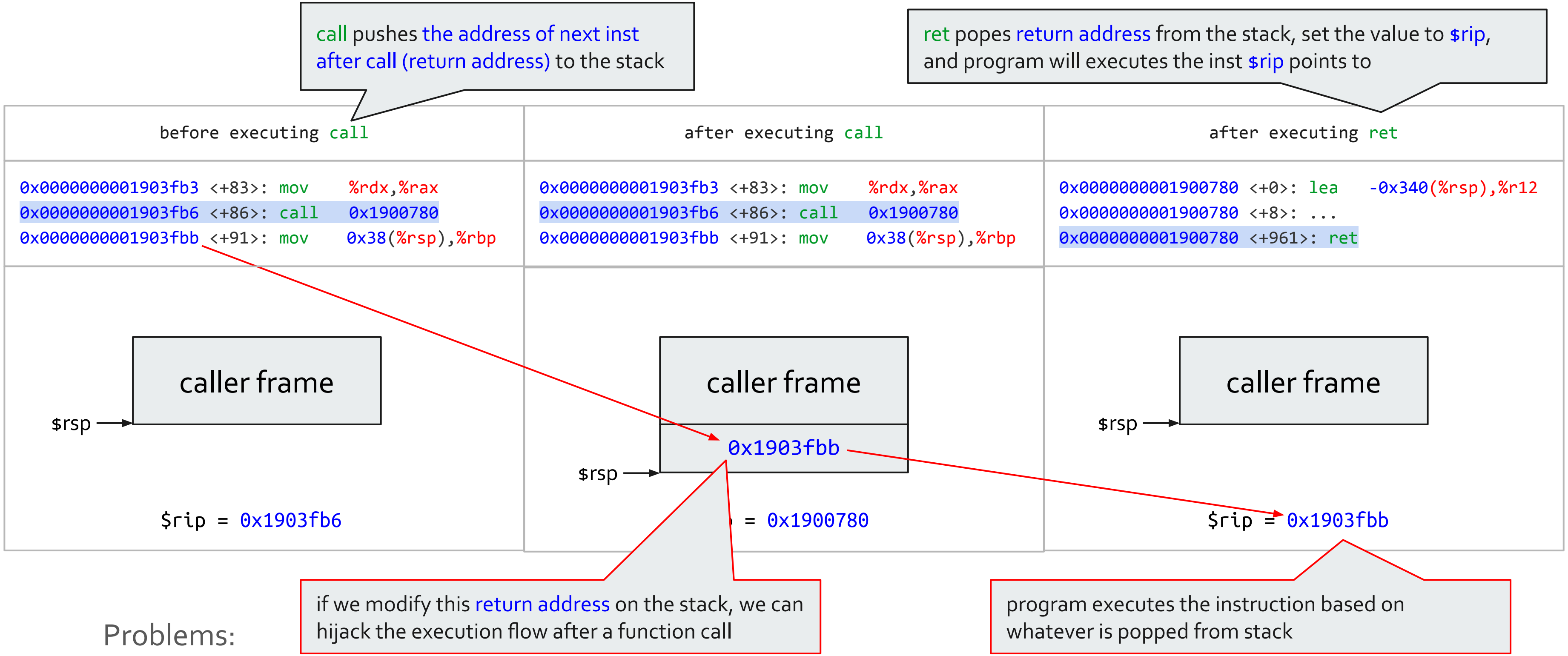
```
#!/usr/bin/bpftrace

uprobe:/usr/bin/dockerd:"github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList" {
        @start[tid] = nsecs;
}

uretprobe:/usr/bin/dockerd:"github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList" {
        if (@start[tid] != 0) {
            printf("getNetworksList took %d ms\n", (nsecs - @start[tid]) / 1000000);
            delete(@start[tid]);
        }
}
```

`tid` is a built-in variable holding thread id
`@start` now is a global map

Problems:

- uretprobe
- TID

uretprobe: implementation and problems



call pushes the address of next inst after call (return address) to the stack

ret popes return address from the stack, set the value to $rip, and program will executes the inst $rip points to

before executing call

```
0x0000000001903fb3 <+83>: mov    %rdx,%rax
0x0000000001903fb6 <+86>: call   0x1900780
0x0000000001903fbb <+91>: mov    0x38(%rsp),%rbp
```

after executing call

```
0x0000000001903fb3 <+83>: mov    %rdx,%rax
0x0000000001903fb6 <+86>: call   0x1900780
0x0000000001903fbb <+91>: mov    0x38(%rsp),%rbp
```

after executing ret

```
0x0000000001900780 <+0>: lea   -0x340(%rsp),%r12
0x0000000001900780 <+8>: ...
0x0000000001900780 <+961>: ret
```

if we modify this return address on the stack, we can hijack the execution flow after a function call

program executes the instruction based on whatever is popped from stack

Problems:

- uretprobe is implemented by modifying return address on the stack to hijack execution flow
- Golang dynamic stack management: check return addresses on stack extension or shrinkage, and crash if unrecognised return address found

Solution to uretprobe problem

Idea: don’t use uretprobe, use uprobe for each exit points

```
#!/usr/bin/bpftrace
uprobe:/usr/bin/dockerd:0x0000000001900780 {
        @start[tid] = nsecs;
}
uprobe:/usr/bin/dockerd:0x00000000019008c1 {
        if (@start[tid] != 0) {
            printf("getNetworksList took %d ms\n", (nsecs - @start[tid]) / 1000000);
            delete(@start[tid]);
        }
}
uprobe:/usr/bin/dockerd:0x0000000001900b41 {
        if (@start[tid] != 0) {
            printf("getNetworksList took %d ms\n", (nsecs - @start[tid]) / 1000000);
            delete(@start[tid]);
        }
}
```

use uprobe instead of uretprobe

the same `ret` addresses in our gdb script

Advantages from not using uretprobe:

- uretprobe has limitation of maximum 64 events on the same address
- ret addresses give better information of where a function returns

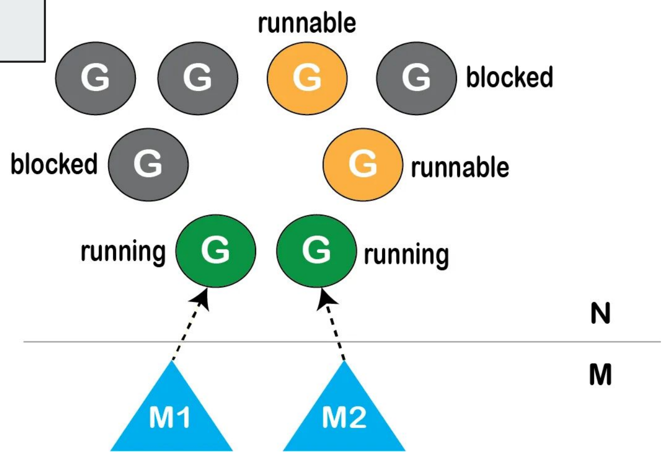
TID (Thread ID) problems

```
#!/usr/bin/bpftrace
uprobe:/usr/bin/dockerd:0x0000000001900780 {
        @start[tid] = nsecs;
}
uprobe:/usr/bin/dockerd:0x00000000019008c1 {
        if (@start[tid] != 0) {
            printf("getNetworksList took %d ms\n", (nsecs - @start[tid]) / 1000000);
            delete(@start[tid]);
        }
}
```

Entry address

Exit address

These 2 thread IDs are likely to be different



Problems:

- tid doesn't stay unchanged during a goroutine's lifetime

Solution to TID problem

```
// src/runtime/runtime2.go
type g struct {
...
    goid            uint64
...
}
```

Ideas:

- goid seems to be a perfect alternative for tid
- goid = g->goid
- How to get *runtime.g?

```
// src/runtime/asm_amd64.s
// func setg(gg *g)
// set g. for use by needm.
TEXT runtime·setg(SB), NOSPLIT, $0-8
    MOVQ    gg+0(FP), BX
    get_tls(CX)
    MOVQ    BX, g(CX)
    RET

#ifdef GOARCH_amd64
#define    get_tls(r)    MOVQ TLS, r
#define    g(r)    0(r)(TLS*1)
#endif
```

Solution to TID problem

Step 1: get `*runtime.g`

```
(gdb) disas/m 'runtime.setg'
Dump of assembler code for function runtime.setg:
1044            MOVQ    gg+0(FP), BX
   0x000000000046dec0 <+0>:     mov     0x8(%rsp),%rbx
1045            get_tls(CX)
   0x000000000046dec5 <+5>:     mov     $0xfffffffffffffffb0,%rcx
1046            MOVQ    BX, g(CX)
   0x000000000046decc <+12>:     mov    %rbx,%fs:(%rcx)
1047            RET
   0x000000000046ded0 <+16>:     ret
```

`first_arg+0(FP)` is the first argument to the function

`%rbx` holds *runtime.g

`%rcx` = -8
`$0xfffffffffffffffb0` is -8 in 2's complement

`%fs:(%rcx) == %fs:(-8) == -8(%fs) == *(%fs-8)`
`*(%fs-8)` holds *runtime.g instance

FS register is used to store thread-local information, we can access it via `struct pthread`

```
#!/usr/bin/bpftrace
#include <linux/sched.h>

$task = (struct task_struct*)curtask;
$fs = (uint64)$task->thread.fsbase;    # %fs
$gaddr = *(uint64*)uptr($fs-8);        # %fs:0xfffffffffffffffb0
```

`curtask` is a built-in variable holding `struct task_struct*`

`uptr` is an annotation to let bpftrace know this pointer belongs to userspace address space

Solution to TID problem

Step 2: get `goid` from `*runtime.g`

Idea: `g->goid => *(g + offsetof(goid))`

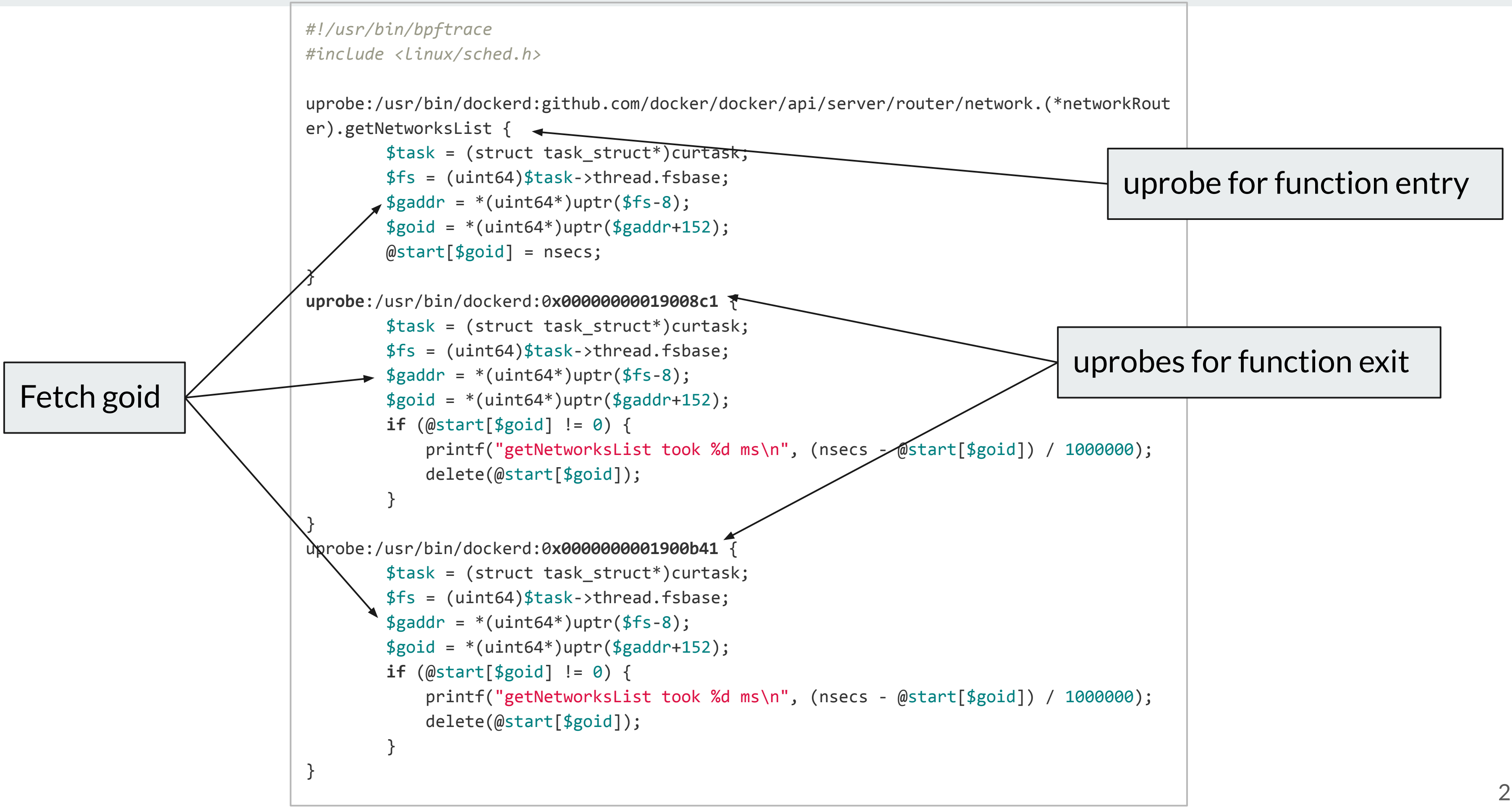
`pahole(1)` parses debug info (DWARF) from a binary and outputs data structure layout

```
$ pahole -C runtime.g /usr/bin/dockerd 2>/dev/null
struct runtime.g {
...
    int64                        goid;                    /*   152     8 */
}
```

`offsetof(runtime.g, goid) = 152`

```
#!/usr/bin/bpftrace
#include <linux/sched.h>

$task = (struct task_struct*)curtask;
$fs = (uint64)$task->thread.fsbase;      # %fs
$gaddr = *(uint64*)uptr($fs-8);          # %fs:0xfffffffffffffb0

$goid = *(uint64*)uptr($gaddr+152);      # g->goid
```

bpftrace script



```
#!/usr/bin/bpftrace
#include <linux/sched.h>

uprobe:/usr/bin/dockerd:github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList {
        $task = (struct task_struct*)curtask;
        $fs = (uint64)$task->thread.fsbase;
        $gaddr = *(uint64*)uptr($fs-8);
        $goid = *(uint64*)uptr($gaddr+152);
        @start[$goid] = nsecs;
}
uprobe:/usr/bin/dockerd:0x00000000019008c1 {
        $task = (struct task_struct*)curtask;
        $fs = (uint64)$task->thread.fsbase;
        $gaddr = *(uint64*)uptr($fs-8);
        $goid = *(uint64*)uptr($gaddr+152);
        if (@start[$goid] != 0) {
            printf("getNetworksList took %d ms\n", (nsecs - @start[$goid]) / 1000000);
            delete(@start[$goid]);
        }
}
uprobe:/usr/bin/dockerd:0x0000000001900b41 {
        $task = (struct task_struct*)curtask;
        $fs = (uint64)$task->thread.fsbase;
        $gaddr = *(uint64*)uptr($fs-8);
        $goid = *(uint64*)uptr($gaddr+152);
        if (@start[$goid] != 0) {
            printf("getNetworksList took %d ms\n", (nsecs - @start[$goid]) / 1000000);
            delete(@start[$goid]);
        }
}
```

uprobe for function entry

uprobes for function exit

Fetch goid

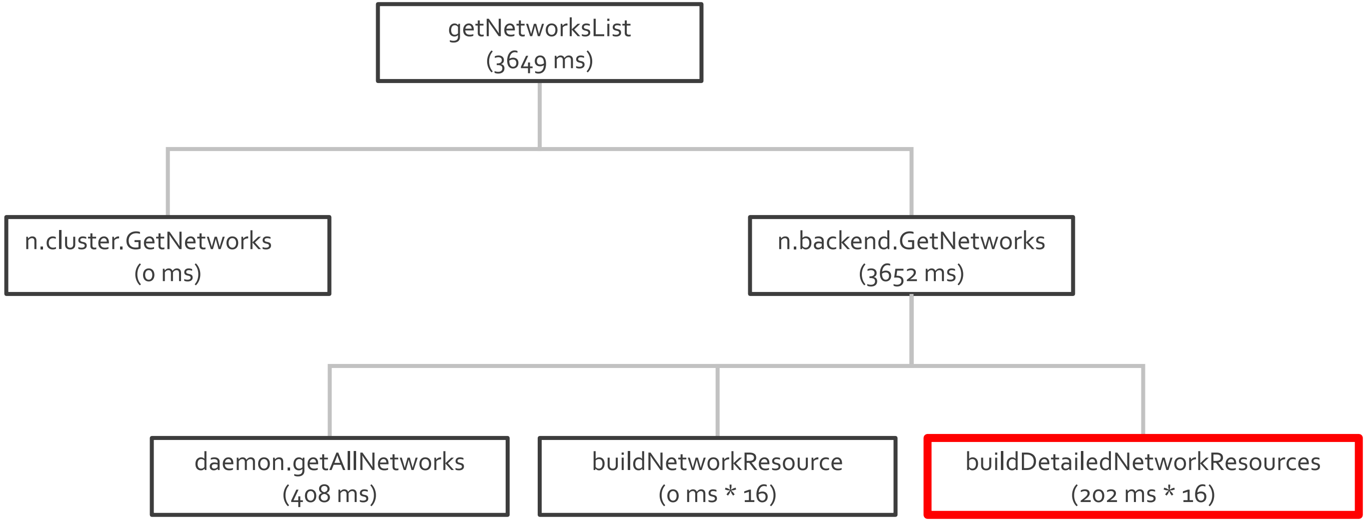
Run bpftrace script

```
$ sudo bpftrace tracing-docker.bt

Attaching 3 probes...
getNetworksList took 3707 ms
^C
```

Tracing other functions:

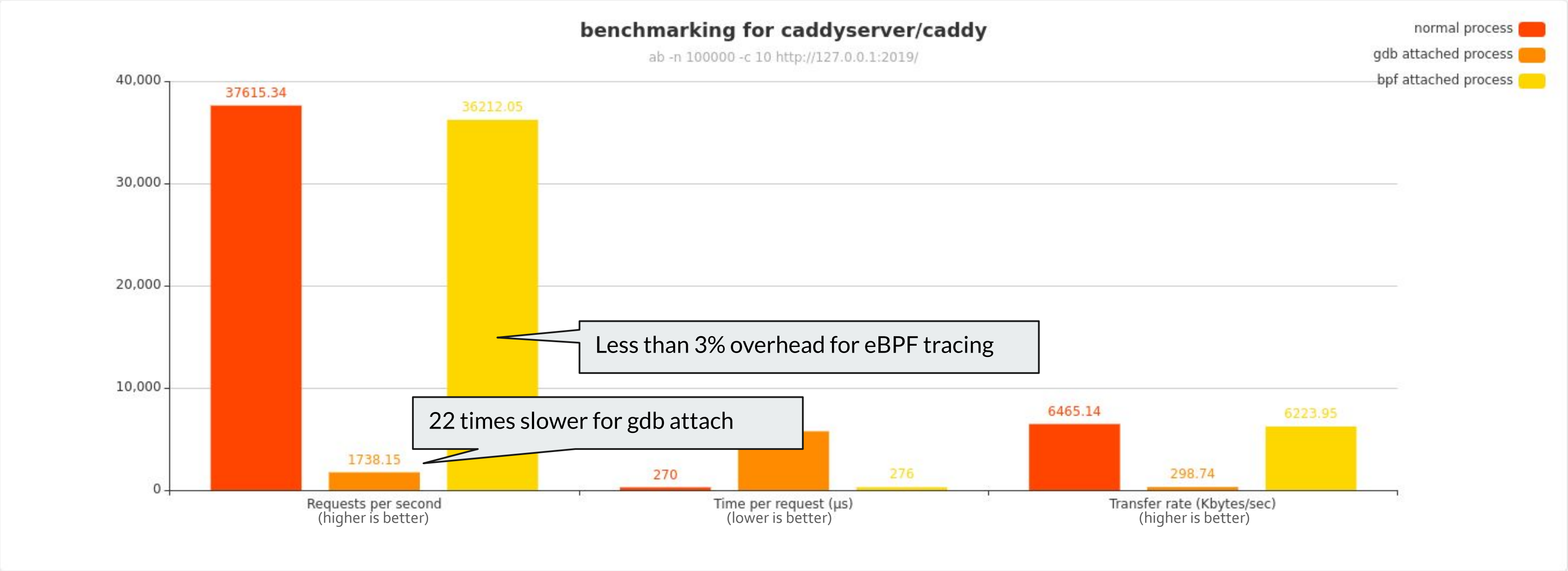
1. Change symbol name (entrypoint)
2. Change ret addresses (exitpoints)
3. Copy & paste

Back to the Docker issue #27729



Root cause: dockerd fetches unnecessary network details from etcd for network list request

Benchmark for eBPF



Less than 3% overhead for eBPF tracing

22 times slower for gdb attach

Recap for Docker issue

We discussed:

- Traditional debugging approaches vs eBPF
- uprobe, uretprobe
- goid

Scenarios: real time profiling

- CPU time profiling (perf) + off-CPU time profiling

Inspirations:

- uprobes can be attached on if condition / for loop => execution details
- Trace all the functions at once
    - Summarize latencies as a histogram => (bcc) funclatency for go
    - Group events by goid=> (ftrace) funcgraph for go

funcgraph

```
perf ftrace --graph-funcs do_sys_open
```

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |

 0)               |  sys_open() {
 0)               |        do_sys_open() {
 0)               |            getname() {
 0)               |                kmem_cache_alloc() {
 0)   1.382 us    |                  __might_sleep();
 0)   2.478 us    |                }
 0)               |                strncpy_from_user() {
 0)               |                  might_fault() {
 0)   1.389 us    |            __might_sleep();
 0)   2.553 us    |                }
 0)   3.807 us    |            }
 0)   7.876 us    |        }
 0)               |        alloc_fd() {
 0)   0.668 us    |        _spin_lock();
 0)   0.570 us    |        expand_files();
 0)   0.586 us    |        _spin_unlock();
```

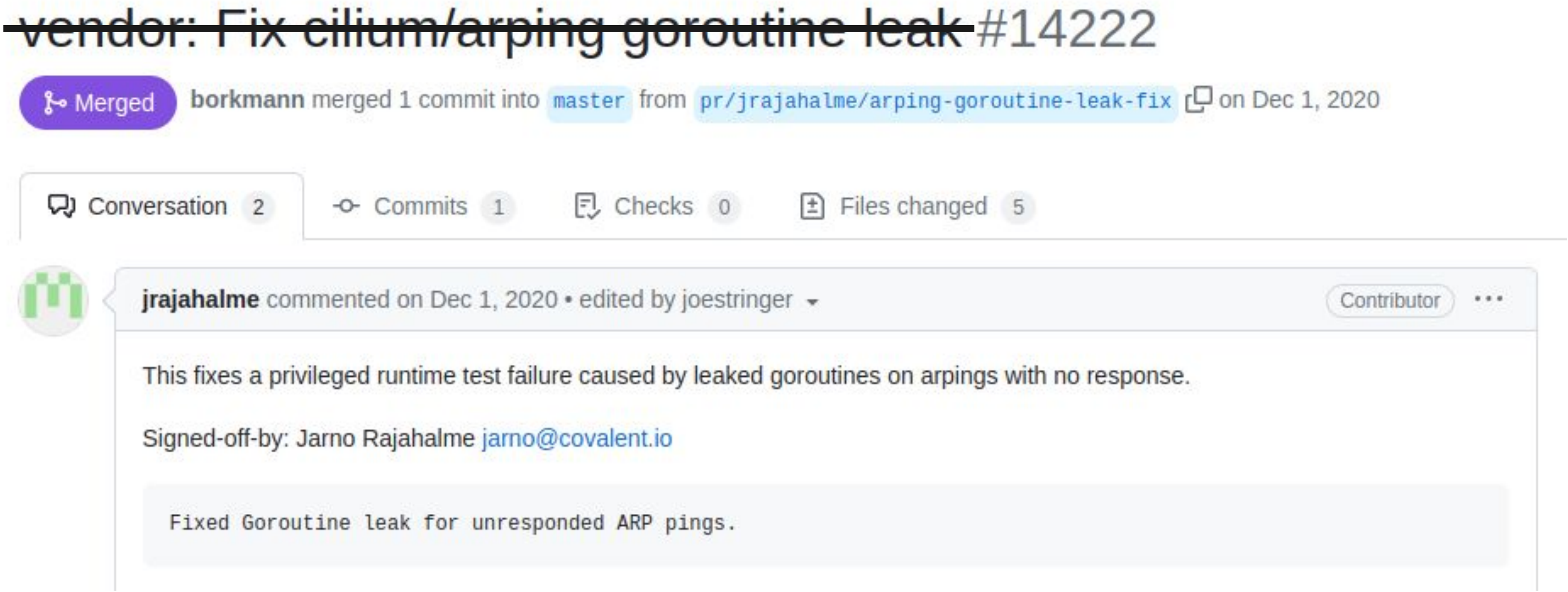
```
gofuncgraph /usr/bin/dockerd '*getNetworksList'
```

```
         github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList-fm() {
           github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList() {
             github.com/docker/docker/daemon/cluster.(*Cluster).GetNetworks() {
               github.com/docker/docker/daemon/cluster.(*Cluster).getNetworks() {
                 github.com/docker/docker/daemon/cluster.(*nodeRunner).State() {
000.0000         }
                 github.com/docker/docker/daemon/cluster.(*Cluster).errNoManager() {
000.0000         }
                 github.com/docker/docker/daemon/cluster.(*Cluster).getNetworks.func1() {
000.0000         }
000.0000       }
000.0000     }
             github.com/docker/docker/daemon.(*Daemon).GetNetworks() {
               github.com/docker/docker/daemon.buildNetworkResource() {
                 github.com/docker/docker/daemon.buildIpamResources() {
000.0000         }
000.0001       }
               github.com/docker/docker/daemon/network.FilterNetworks() {
000.0000       }
               github.com/docker/docker/daemon.buildDetailedNetworkResources() {
000.2032       }
               github.com/docker/docker/daemon.buildDetailedNetworkResources() {
                 github.com/docker/docker/daemon.buildEndpointResource() {
000.0000         }
000.2030       }
               github.com/docker/docker/daemon.buildDetailedNetworkResources() {
000.2029       }
003.6532     }
003.6537   }
003.6538 }
```

cilium

https://github.com/cilium/cilium/pull/14222

cilium-agent has memory leak when nodes aren’t L2 connected



~~vendor: Fix cilium/arping goroutine leak~~ #14222

borkmann merged 1 commit into `master` from `pr/jrajahalme/arping-goroutine-leak-fix` on Dec 1, 2020

jrajahalme commented on Dec 1, 2020 • edited by joestringer

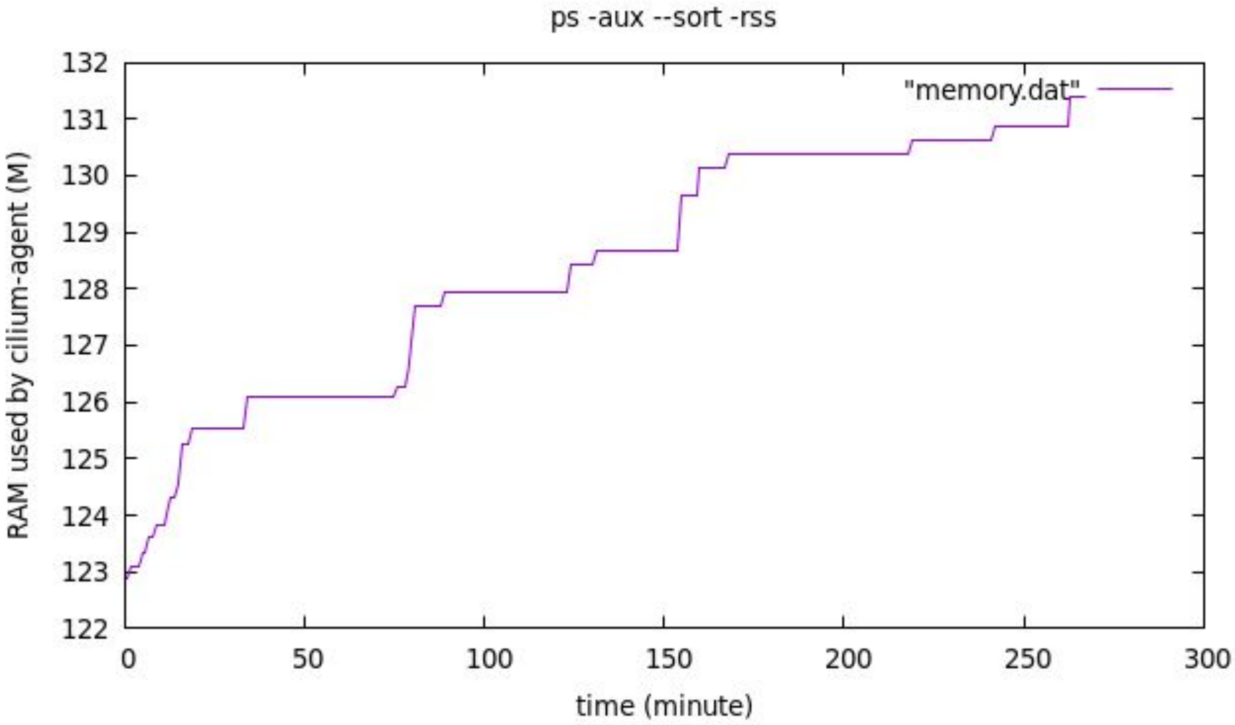
This fixes a privileged runtime test failure caused by leaked goroutines on arpings with no response.

Signed-off-by: Jarno Rajahalme jarno@covalent.io

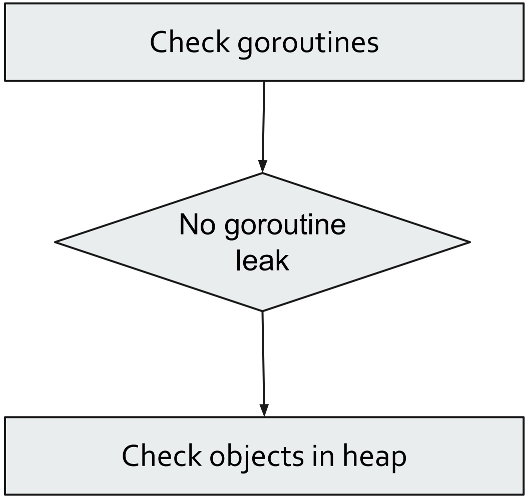
```
Fixed Goroutine leak for unresponded ARP pings.
```

Reproduce

Create a two-node cilium cluster with nothing deployed, then leave it alone and measure the memory usage once a minute



Debugging Ideas



Check goroutines

No goroutine leak

Check objects in heap

Traditional debugger: coredump

```
$ gdb -p $(pidof cilium-agent)

(gdb) gcore
warning: Memory read failed for corefile section, 4096 bytes at 0xffffffffff600000.
Saved corefile core.292332
```

```
$ dlv core ./cilium core.292332

(dlv) goroutines
...
  Goroutine 145552 - User: /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:143 github.com/cilium/arping.PingOverIface.func1
(0x257d6c5) [chan send 18131210974072]
  Goroutine 145617 - User: /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:143 github.com/cilium/arping.PingOverIface.func1
(0x257d6c5) [chan send 181544473419392]
[2337 goroutines]

(dlv) goroutine 145552
Switched from 0 to 145552 (thread 292332)

(dlv) bt
0  0x000000000043a6d6 in runtime.gopark
   at /usr/local/go/src/runtime/proc.go:382
1  0x000000000040730e in runtime.chansend
   at /usr/local/go/src/runtime/proc.go:259
2  0x0000000000406ebd in runtime.chansend1
   at /usr/local/go/src/runtime/proc.go:145
3  0x000000000067d945 in github.com/cilium/arping.PingOverIface.func1
   at /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:143
4  0x000000000046be81 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:1598
```

Delve is a debugger for Go

Problems:

- Long time to generate corefile
- No clue about a goroutine's creator

Internal tool: pprof

```
curl localhost:6060/debug/pprof/goroutine?debug=2
```

how long since created

```
goroutine 5132 [chan send, 3 minutes]:
github.com/cilium/arping.PingOverIface.func1()
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:143 +0x7a5
created by github.com/cilium/arping.PingOverIface
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:132 +0x3e5

goroutine 5573 [chan send]:
github.com/cilium/arping.PingOverIface.func1()
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:143 +0x7a5
created by github.com/cilium/arping.PingOverIface
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:132 +0x3e5

goroutine 5514 [chan send]:
github.com/cilium/arping.PingOverIface.func1()
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:143 +0x7a5
created by github.com/cilium/arping.PingOverIface
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:132 +0x3e5
```

creator

Problems:

- pprof may not be turned on
- Miss the details of creator: stack backtrack

eBPF: leaking goroutine tracing

Stage ONE: backtrace for leaking goroutines' creators

1. Trace goroutines' creation and destruction
2. Store creator's backtrace at creation, delete backtrace at destruction
3. Dump the stored backtrace

Stage 1: backtrace for leaking goroutines' creators

```
#!/usr/bin/bpftrace
#include <linux/sched.h>

uprobe:[bin]:[ scheduler:creaing_a_goroutine ] {
    // $task = (struct task_struct*)curtask;
    // $tls_base = (uint64)$task->thread.fsbase;
    // $gaddr = *(uint64*)uptr($tls_base-8);
    // $goid = *(uint64*)uptr($gaddr+152);
    $new_goid = [ get_new_goid ]
    @created_by[$new_goid] = ustack();
}

uprobe:[bin]:[ scheduler:ending_a_goroutine ] {
      $task = (struct task_struct*)curtask;
      $tls_base = (uint64)$task->thread.fsbase;
      $gaddr = *(uint64*)uptr($tls_base-8);
      $goid = *(uint64*)uptr($gaddr+152);
      delete(@create_by[$goid]);
}
```

We are in the creator's context, this goid is creator' goid, but we want new created goid

`ustack()` is a built-in function to fetch stack backtrace

Leaking goroutines have their goids stored in `@create_by` forever

Stage 1: backtrace for leaking goroutines’ creators

[scheduler:ending_a_goroutine]

```
// Finishes execution of the current goroutine.
func goexit1() {
    if raceenabled {
       racegoend()
    }
    if trace.enabled {
       traceGoEnd()
    }
    mcall(goexit0)
}
```

Stage 1: backtrace for leaking goroutines’ creators

[scheduler:creaing_a_goroutine]

```
// Create a new g in state _Grunnable, starting at fn. callerpc is the
// address of the go statement that created this. The caller is responsible
// for adding the new g to the scheduler.
func newproc1(fn *funcval, callergp *g, callerpc uintptr) *g {
...
        return newg
}
```

Stage 1: backtrace for leaking goroutines' creators

[get_new_goid]

Idea: get return value(newg), then newg->goid

```
(gdb) disas/m 'runtime.newproc1'

4347     return newg
   0x000000000044483b <+827>:    mov 0x20(%rsp),%rax
   0x000000000044486cq <+876>:    mov       0x40(%rsp),%rbp
   0x0000000000444871 <+881>:    add $0x48,%rsp
   0x0000000000444875 <+885>:    ret

(gdb) i scope 'runtime.newproc1'

Symbol newg is multi-location:
  Base address 0x444500  Range 0x44456
in $rax
  Range 0x444576-0x444587: a variable in $rax
  Range 0x444587-0x444596: a complex DWARF expression:
       0: DW_OP_fbreg -48
  Range 0x444596-0x4445d5: a variable in $rax
  Range 0x4445d5-0x444876: a complex DWARF expression:
       0: DW_OP_fbreg -48
```

①. info scope command gives information about how we can find a local variable through memory or register

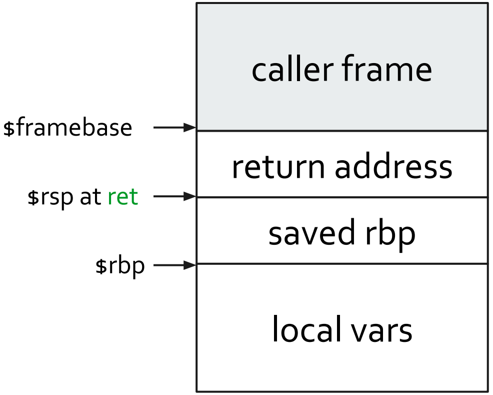
those information are parsed from binary's debug info (DWARF)

②. Find the block for the variable newg

③. ret address 0x0000000000444875 falls into range 0x4445d5-0x444876

④. DW_OP_fbreg-48 DWARF expression means this variable can be find on *($framebase-48), where $framebase is $rsp before executing call

at ret point, $framebase = $rsp+8



Stage 1: backtrace for leaking goroutines' creators

```
#!/usr/bin/bpftrace
#include <linux/sched.h>

u:./cilium-agent:0x0000000000444875 {
        $new_g = *(uint64*)uptr(reg("sp")+8-48);
        $new_goid = *(uint64*)uptr($new_g+152);
        @created_by[$new_goid] = ustack();
}

u:./cilium-agent:"runtime.goexit1" {
        $task = (struct task_struct*)curtask;
        $tls_base = (uint64)$task->thread.fsbase;
        $gaddr = *(uint64*)uptr($tls_base-8);
        $goid = *(uint64*)uptr($gaddr+152);
        delete(@created_by[$goid]);
}
```

Address of
`runtime.newproc1`'s `ret`

```
DW_OP_fbreg-48
= *($framebase-48)
= *($rsp+8-48)
```

```
@created_by[16929]:
  runtime.newproc1+885
  runtime.systemstack.abi0+73
  github.com/cilium/arping.PingOverIface+997
  github.com/cilium/arping.Ping+254
  github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).insertNeighborCommon+443
  github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).insertNeighbor6+3570
  github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).insertNeighbor+885
  github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).nodeUpdate.func1+52
  runtime.goexit.abi0+1
```

eBPF: leaking goroutine tracing

Stage TWO: backtrace for leaking goroutines

1. We already stored leaking goroutines’ *g at `@created_by`, just need to backtrace for them when tracing completes
2. Complete tracing when 20 leaking goroutines detected

Stage 2: backtrace for leaking goroutines

```
uprobe:./cilium-agent:0x0000000000444875 {
        $new_g = *(uint64*)uptr(reg("sp")+8-48);
        $new_goid = *(uint64*)uptr($new_g+152);
        @created_by[$new_goid] = ustack;

        $i = 0;
        $found = 0;
        while ($i <= 20) {
            if (@gs[$i] == 0) {
                @gs[$i] = $new_g;
                @g_indices[$new_goid] = $i;
                $found = 1;
                break;
            }
            $i++;
        }

        if ($found == 0) {
            [ do stack backtrace for leaking goroutines ]
            exit();
        }
}
```

@gs stores 20 *runtime.g

Search for an empty slot in @gs to store *runtime.g

If can't find an empty slot, tracing completes, do stack backtrace for each *runtime.g in @gs

```
uprobe:./cilium-agent:"runtime.goexit1" {
        $task = (struct task_struct*)curtask;
        $tls_base = (uint64)$task->thread.fsbase;
        $gaddr = *(uint64*)uptr($tls_base-8);
        $goid = *(uint64*)uptr($gaddr+152);
        delete(@create_by[$goid]);

        @gs[@g_indices[$goid]] = 0;
        delete(@g_indices[$goid]);
```

Re-mark the slot in @gs empty for exited goroutines

Stage 2: backtrace for leaking goroutines

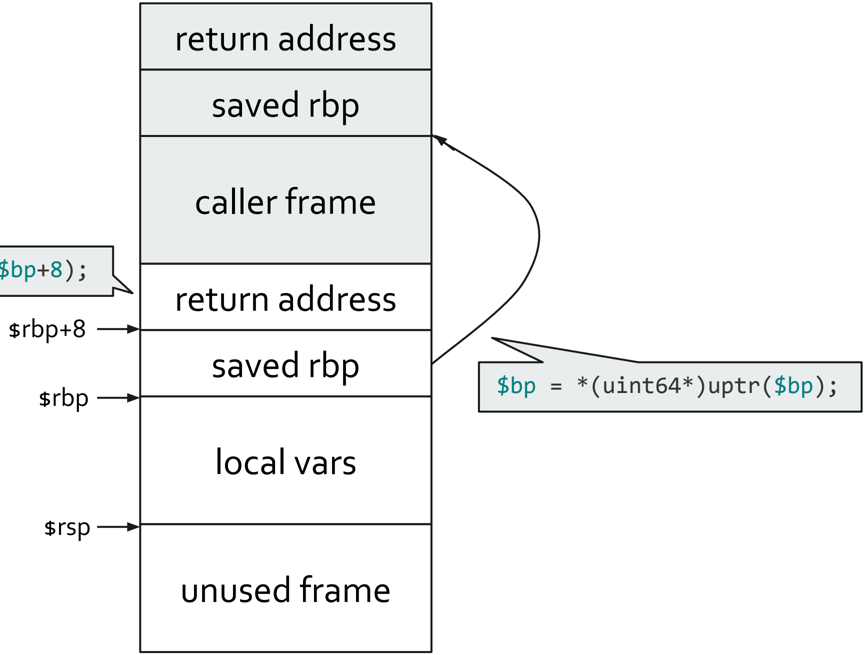
[do stack backtrace for leaking goroutines]

```
if ($found == 0) {
    $i = 0;
    while ($i <= 20) {
        $g = @gs[$i];
        $pc = [ retrieve $rip from *runtime.g ];
        $bp = [ retrieve $rbp from *runtime.g ];
        printf("%s\n", usym($pc))

        $j = 0;
        while ($j < 10) {
            $ra = *(uint64*)uptr($bp+8);
            if ($ra == 0) {
                break;
            }
            printf("%s\n", usym($ra));
            $bp = *(uint64*)uptr($bp);
            $j++;
        }

        $goid = *(uint64*)uptr($g+152);
        printf("created by %s\n", @created_by[$goid]);
        $i++;
    }
    exit();
}
```

Backtrace for 10 layers

usym() is a built-in function to resolve address to symbol

`$ra = *(uint64*)uptr($bp+8);`



`$bp = *(uint64*)uptr($bp);`

Stage 2: backtrace for leaking goroutines

[retrieve $rip from *runtime.g]

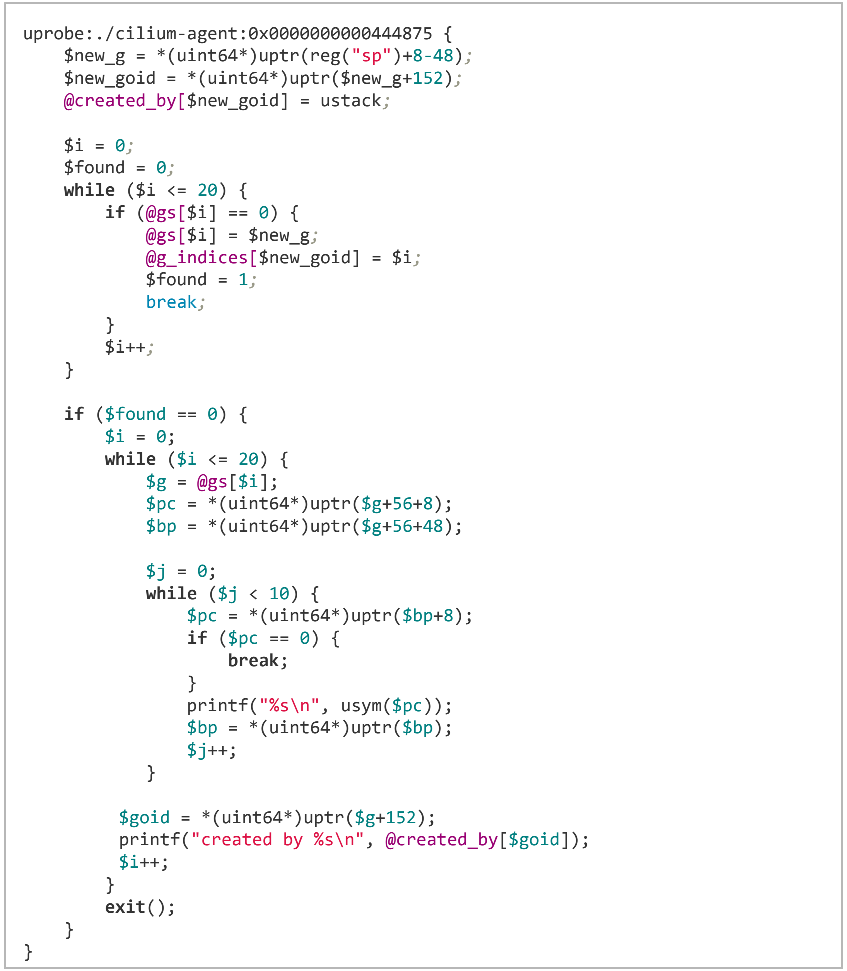
Idea: rip = g->sched.pc => *(g + offsetof(g, sched) + offsetof(gobuf, pc))

```
// src/runtime/runtime2.go
type g struct {
...
    sched           gobuf
...
}

type gobuf struct {
    sp   uintptr
    pc   uintptr
    g    guintptr
    ctxt unsafe.Pointer
    ret  uintptr
    lr   uintptr
    bp   uintptr // for
framepointer-enabled architectures
}
```

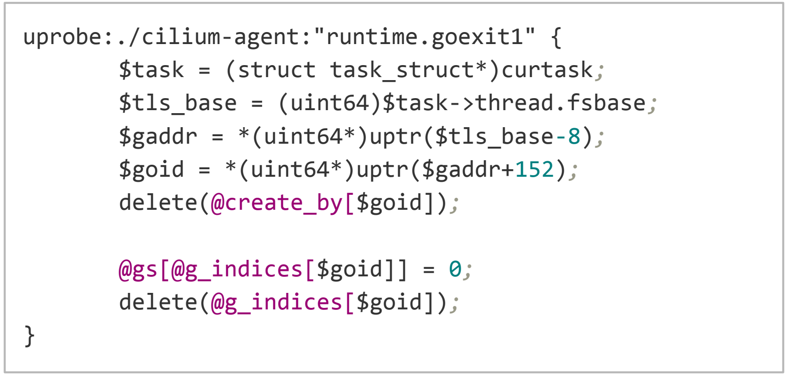
```
$ pahole -C runtime.g ./cilium-agent
struct runtime.g {
...
    runtime.gobuf            sched;                 /*    56      56 */
...
}

$ pahole -C runtime.gobuf ./cilium-agent
struct runtime.gobuf {
    uintptr                  sp;                    /*     0       8 */
    uintptr                  pc;                    /*     8       8 */
...
    uintptr                  bp;                    /*    48       8 */

    /* size: 56, cachelines: 1, members: 7 */
    /* last cacheline: 56 bytes */
};
```

```
$pc = *(uint64*)uptr($g+56+8);
$bp = *(uint64*)uptr($g+56+48);
```

bpftrace script



```
uprobe:./cilium-agent:0x0000000000444875 {
    $new_g = *(uint64*)uptr(reg("sp")+8-48);
    $new_goid = *(uint64*)uptr($new_g+152);
    @created_by[$new_goid] = ustack;

    $i = 0;
    $found = 0;
    while ($i <= 20) {
        if (@gs[$i] == 0) {
            @gs[$i] = $new_g;
            @g_indices[$new_goid] = $i;
            $found = 1;
            break;
        }
        $i++;
    }

    if ($found == 0) {
        $i = 0;
        while ($i <= 20) {
            $g = @gs[$i];
            $pc = *(uint64*)uptr($g+56+8);
            $bp = *(uint64*)uptr($g+56+48);

            $j = 0;
            while ($j < 10) {
                $pc = *(uint64*)uptr($bp+8);
                if ($pc == 0) {
                    break;
                }
                printf("%s\n", usym($pc));
                $bp = *(uint64*)uptr($bp);
                $j++;
            }

         $goid = *(uint64*)uptr($g+152);
         printf("created by %s\n", @created_by[$goid]);
         $i++;
        }
        exit();
    }
}
```



```
uprobe:./cilium-agent:"runtime.goexit1" {
        $task = (struct task_struct*)curtask;
        $tls_base = (uint64)$task->thread.fsbase;
        $gaddr = *(uint64*)uptr($tls_base-8);
        $goid = *(uint64*)uptr($gaddr+152);
        delete(@create_by[$goid]);

        @gs[@g_indices[$goid]] = 0;
        delete(@g_indices[$goid]);
}
```

Possible improvements:

We can record start time and calculate goroutines’ existing time as pprof/goroutine does

Back to the Cilium issue #14222

bpftrace output:

```
    runtime.chansend
    runtime.chansend1
    github.com/cilium/arping.PingOverIface.func1
    runtime.goexit.abi0
  created by
    runtime.newproc1+885
    runtime.systemstack.abi0+73
    github.com/cilium/arping.PingOverIface+997
    github.com/cilium/arping.Ping+254
    github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).insertNeighborCommon+443
    github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).insertNeighbor6+3570
    github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).insertNeighbor+885
    github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).refreshNeighbor+101
    github.com/cilium/cilium/pkg/datapath/linux.(*linuxNodeHandler).NodeNeighborRefresh.func1+54
    runtime.goexit.abi0+1
```

the newly created goroutine could be stuck at channel operation in PingOverIface.func1

PingOverIface created that goroutine

pprof output:

```
goroutine 5573 [chan send]:
github.com/cilium/arping.PingOverIface.func1()
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:143 +0x7a5
created by github.com/cilium/arping.PingOverIface
    /go/src/github.com/cilium/cilium/vendor/github.com/cilium/arping/arping.go:132 +0x3e5
```

Root cause: cilium/arping.PingOverIface gets stuck on channel IO when cilium's NodeHandler refreshes node neighbor

Recap Cilium issue

We discussed:

- Scheduling functions
- Get variables from stack
- Stack backtrace using *runtime.g

Scenarios: goroutine lifetime monitoring

- Who creates
- When starts
- How long lasts
- How many created
- Where blocks

Inspirations:

- tracing on memory allocation + memory free => memory leak tracing
- funcgraph + local variables => OpenTelemetry on the fly

eBPF

Pitfalls when tracing Golang

Entrypoint: duplicate events

Task: how many times `network.(*networkRouter).getNetworksList` has been called

```
uprobe:/usr/bin/dockerd:"github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList" {
        @called++;
}
```

ignore the concurrency issue here

```
(gdb) disas/m 'github.com/docker/docker/api/server/router/network.(*networkRouter).getNetworksList'

14      func getNetworksList() {
   0x0000000001900780 <+0>:     lea  -0x340(%rsp),%r12
   0x0000000001900788 <+8>:     cmp  0x10(%r14),%r12
   0x000000000190078c <+12>:     jbe 0x1900d8b <getNetworksList+1547>
...
   0x0000000001900d8b <+1547>:     call    0x46bf40 <runtime.morestack_noctxt>
   0x000000000067e83b <+1622>:     jmp     0x1900780 <getNetworksList+0>
```

check whether stack has enough space

if not, jump

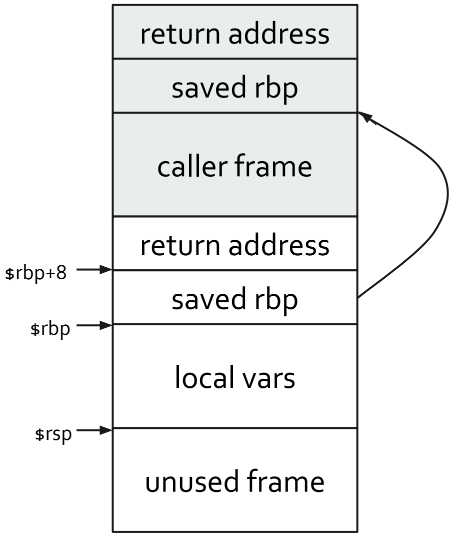
extend the stack

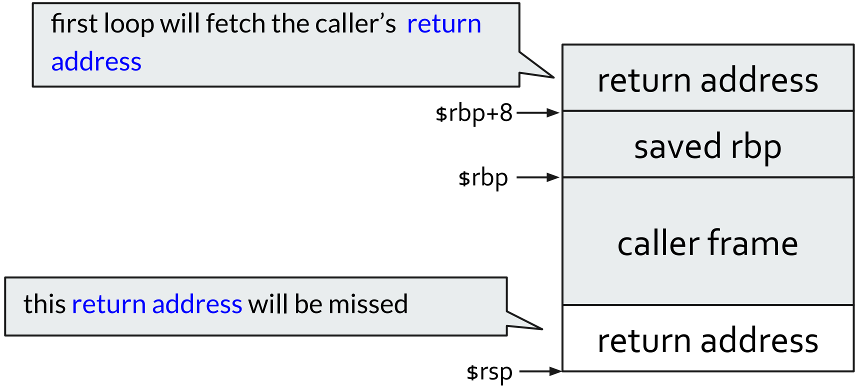
then jump back to entrypoint again

Problems:

- One function call might generate two uprobe events
- Some famous bpf-based tools, like funccount, has the same issue

backtrace: missing caller




first loop will fetch the caller's return address

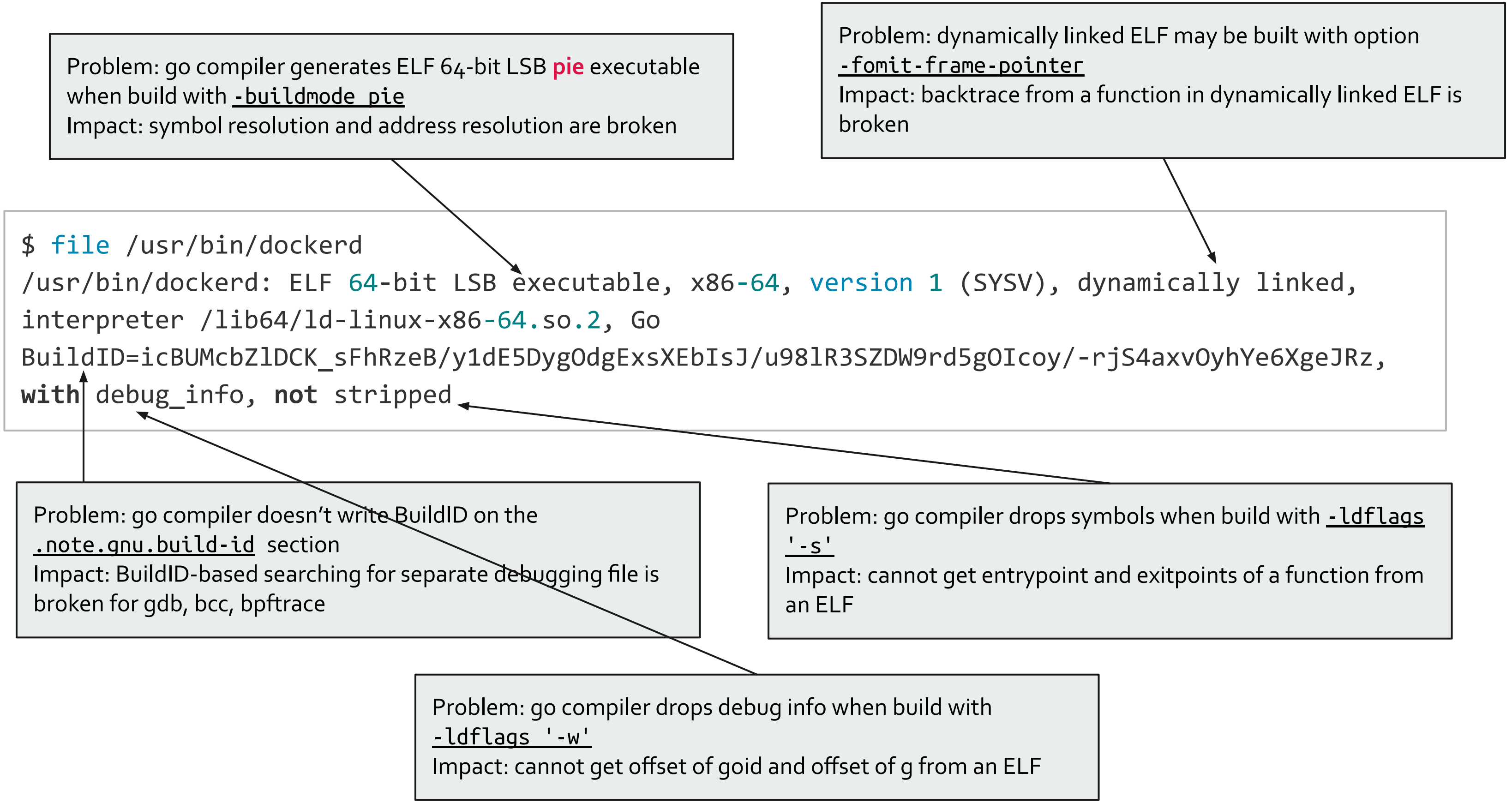
this return address will be missed

```
while ($j < 10) {
    $pc = *(uint64*)uptr($bp+8);
    if ($pc == 0) {
        break;
    }
    printf("%s\n", usym($pc));
    $bp = *(uint64*)uptr($bp);
    $j++;
}
```

Problems:

- A -> B -> C, but backtrace gives A -> C
- bpftrace built-in ustack() and kstack() have the same issue

ELF: what if build with options



Problem: go compiler generates ELF 64-bit LSB pie executable when build with `-buildmode pie`
Impact: symbol resolution and address resolution are broken

Problem: dynamically linked ELF may be built with option `-fomit-frame-pointer`
Impact: backtrace from a function in dynamically linked ELF is broken

```
$ file /usr/bin/dockerd
/usr/bin/dockerd: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, Go
BuildID=icBUMcbZlDCK_sFhRzeB/y1dE5DygOdgExsXEbIsJ/u98lR3SZDW9rd5gOIcoy/-rjS4axvOyhYe6XgeJRz,
with debug_info, not stripped
```

Problem: go compiler doesn’t write BuildID on the `.note.gnu.build-id` section
Impact: BuildID-based searching for separate debugging file is broken for gdb, bcc, bpftrace

Problem: go compiler drops symbols when build with `-ldflags '-s'`
Impact: cannot get entrypoint and exitpoints of a function from an ELF

Problem: go compiler drops debug info when build with `-ldflags '-w'`
Impact: cannot get offset of goid and offset of g from an ELF

Happy debugging

eBPF is the better choice for production troubleshooting

- Faster
- Flexible
- Frictionless
- Fine-grained
- Full-stack visibility
- Future-proof

eBPF

cilium

docker

QA