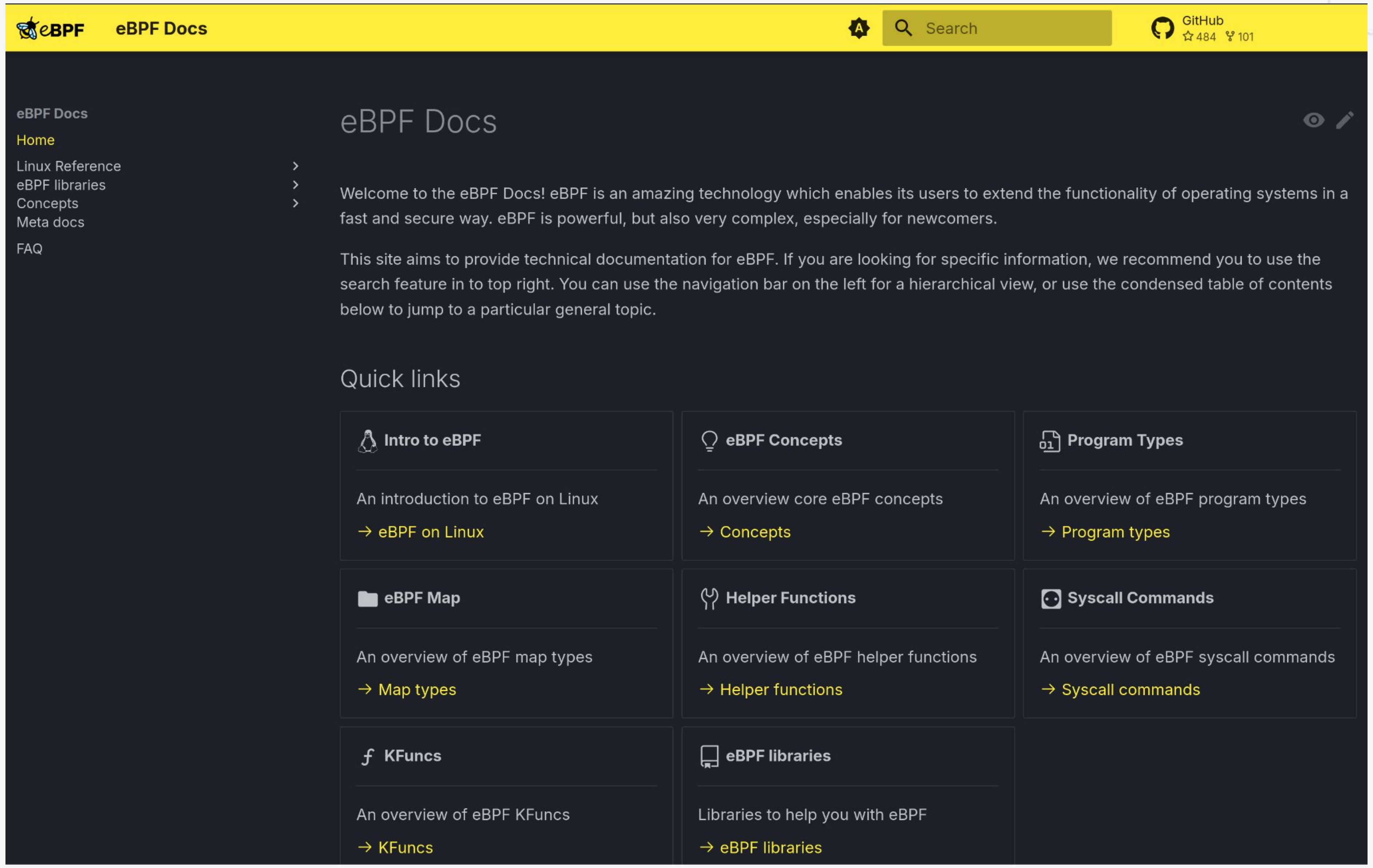
ISOVALENT
now part of CISCO

eBPF Reachability Analysis

Speaker: **Dylan Reimerink**

@me

- Dylan Reimerink
- Software Engineer - Isovalent @ Cisco
- Cilium committer
- Maintainer of ebpf-go (cilium/ebpf)
- Maintainer of docs.ebpf.io



ISOVALENT
now part of cisco

Acknowledgement

- Timo Beckers
  - Cilium committer
  - ebpf-go maintainer
  - co-author of reachability analysis
- Robin Gögge
  - Cilium reviewer + org member
  - ebpf-go reviewer
  - Primary reachability analysis reviewer

Modernizing Cilium

- Cilium is over 10 years old, one of the first eBPF users
- Cilium compiles its eBPF programs at runtime (for now)
- Our goal is to pre-compile all eBPF programs
  - Faster program loading
  - Smaller docker images (not shipping clang toolchain and dependencies)
  - Less attack surface
  - Less complexity, so hopefully less bugs
- Working towards this since 2023

Load time configuration

- Cilium has a significant amount of optional features and settings
  - Cilium has 94 datapath settings (at the moment)
- Disabled features were conditionally compiled with pre-processor macros
- Compile time config needs to become load time config
  - v5.1: bpf: dead code elimination
  - v5.5: Track read-only map contents as known scalars in BPF verifiers

```c
volatile const bool enable_feature_a;

int SEC("tc") entrypoint(struct __sk_buff *ctx) {
    if (enable_feature_a) {
        // ...
    }
    return TC_ACT_OK;
}
```

The “problem”

- Cilium core value: Only pay for what you use
- So how do I prevent paying for a map I don’t use?

```
#ifdef ENABLE_FEATURE_A
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, struct some_key);
    __type(value, struct some_value);
    __uint(max_entries, 1000000);
} feature_a_map SEC(".maps");
#endif

int SEC("tc") entrypoint(struct __sk_buff *ctx) {
#ifdef ENABLE_FEATURE_A
   struct some_key key = {/*...*/};
   struct some_value *val = bpf_map_lookup_elem(&feature_a_map, &key);
   // ...
#endif
   return TC_ACT_OK;
}
```

The naive fix?

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, struct some_key);
    __type(value, struct some_value);
    __uint(max_entries, 1000000);
} feature_a_map SEC(".maps");

volatile const bool enable_feature_a;

int SEC("tc") entrypoint(struct __sk_buff *ctx) {
    if (enable_feature_a) {
        struct some_key key = {/*...*/};
        struct some_value *val = bpf_map_lookup_elem(&feature_a_map, &key);
        // ...
    }
    return TC_ACT_OK;
}
```

The naive fix?

```
$ sudo bpftool prog dump xlated id 2390
int entrypoint(struct __sk_buff * ctx):
; int SEC("tc") entrypoint(struct __sk_buff *ctx)
   0: (b7) r0 = 0
; if (enable_feature_a)
   1: (18) r1 = map[id:281][0]+0
   3: (71) r1 = *(u8 *)(r1 +0)
; if (enable_feature_a)
   4: (15) if r1 == 0x0 goto pc+14
   5: (b7) r1 = 0
; struct some_key key = {/*...*/};
   6: (63) *(u32 *)(r10 -8) = r1
   7: (bf) r2 = r10
   8: (07) r2 += -8
   9: (18) r1 = map[id:279]
  11: (85) call __htab_map_lookup_elem#294448
  12: (15) if r0 == 0x0 goto pc+1
  13: (07) r0 += 56
  14: (bf) r1 = r0
  15: (b7) r0 = 1
  16: (55) if r1 != 0x0 goto pc+1
  17: (b7) r0 = 0
  18: (67) r0 <<= 1
; }
  19: (95) exit
```

```
$ sudo bpftool prog
2390: sched_cls  name entrypoint  tag 61f771b5f10a00fc
    loaded_at 2026-01-28T15:32:14+0100  uid 0
    xlated 160B  jited 97B  memlock 4096B  map_ids 281,279
    btf_id 310
```

The naive fix? - Dead code maintains refcount

```
$ sudo bpftool prog dump xlated id 2392
int entrypoint(struct __sk_buff * ctx):
; int SEC("tc") entrypoint(struct __sk_buff *ctx)
   0: (b7) r0 = 0
; if (enable_feature_a)
   1: (18) r1 = map[id:286][0]+0
   3: (71) r1 = *(u8 *)(r1 +0)
; }
   4: (95) exit
```

```
$ sudo bpftool prog
2392: sched_cls  name entrypoint  tag 61f771b5f10a00fc
    loaded_at 2026-01-28T15:37:22+0100  uid 0
    xlated 40B  jited 31B  memlock 4096B  map_ids 286,284
    btf_id 319
```

The naive fix? - Dead code maintains refcount

- resolve_pseudo_ldimm64 runs early
    - Does a flat scan, converting FDs to pointers
    - Adds map to `env→used_maps`
    - Increments map refcount `bpf_map_inc(map);`
- Majority of verification happens...
- Dead code elimination happens
    - opt_hard_wire_dead_code_branches
    - opt_remove_dead_code
    - opt_remove_nops
- But map refcounts are never released after pointers are eliminated 😔

First workaround - The double load

- Load our program once, maps set to max_entries 1
- Read back the jitted code, see which maps remain
- Change LDIMM64 of unused maps FDs to load imm 0xDEADC0DE
- Load with modified instructions and only used maps with full max_entries
- Works (sort of), but...
  - Loading takes long, a lot of tailcall programs, and large programs
  - We still have to create maps, not free, even small ones
  - We risk hitting the 64 map limit on the first run
  - We cannot gate unsupported map types behind features (arena maps)
  - Reading back instructions impossible when `kernel.kptr_restrict` + `net.core.bpf_jit_harden` are set.

Second workaround - matching userspace logic

- Write userspace logic that matches the logic in eBPF to disable maps
- We did not go this route
  - If the compiler re-orders code, the compiled program may not match userspace logic, even though sources do
  - Chances for human error are significant
  - Burdon on maintainers undesirable

Final workaround - reachability analysis

- What if we knew before loading which instructions in our program are reachable under the given load time configuration?
- A reachability analysis if you will
- Applicable to both maps, tailcalls to hardcoded slots and bpf-to-bpf functions
- Addresses most concerns, but did cost some engineering effort

Reachability analysis - basic block

```
 0: r0 = 0
 1: r1 = map[id:281][0]+0
 3: r1 = *(u8 *)(r1 +0)
 4: if r1 == 0x0 goto pc+14
 5: r1 = 0
 6: *(u32 *)(r10 -8) = r1
 7: r2 = r10
 8: r2 += -8
 9: r1 = map[id:279]
11: call __htab_map_lookup_elem#294448
12: if r0 == 0x0 goto pc+1
13: r0 += 56
14: r1 = r0
15: r0 = 1
16: if r1 != 0x0 goto pc+1
17: r0 = 0
18: r0 <<= 1
19: exit
```

Reachability analysis - basic block

```
   0: r0 = 0
   1: r1 = map[id:281][0]+0
   3: r1 = *(u8 *)(r1 +0)
   4: if r1 == 0x0 goto pc+14
---
   5: r1 = 0
   6: *(u32 *)(r10 -8) = r1
   7: r2 = r10
   8: r2 += -8
   9: r1 = map[id:279]
  11: call   htab map lookup_elem#294448
  12: if r0 == 0x0 goto pc+1
---
  13: r0 += 56
  14: r1 = r0
  15: r0 = 1
  16: if r1 != 0x0 goto pc+1
---
  17: r0 = 0
  18: r0 <<= 1
  19: exit
```

Reachability analysis - basic block

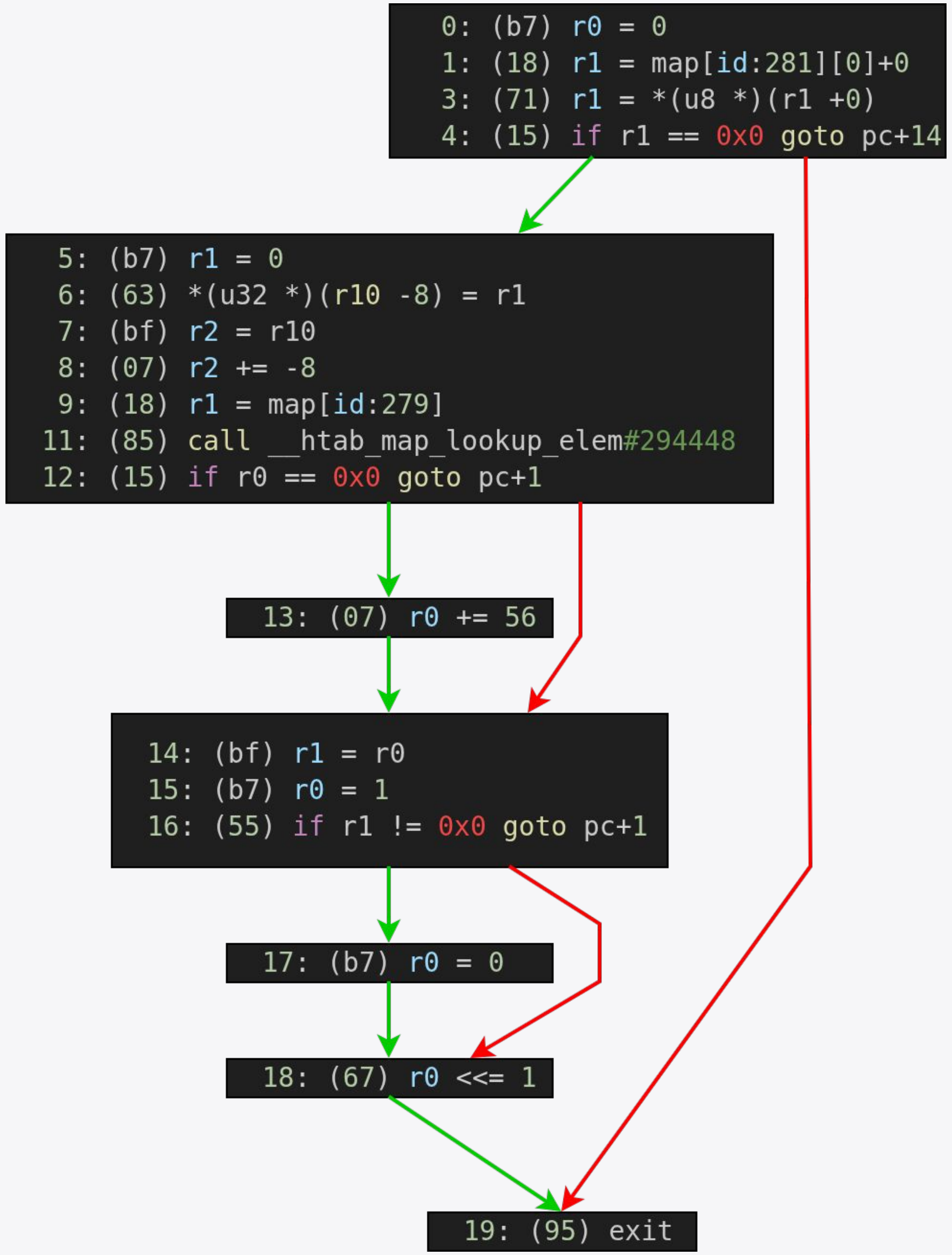
```
   0: r0 = 0
   1: r1 = map[id:281][0]+0
   3: r1 = *(u8 *)(r1 +0)
   4: if r1 == 0x0 goto pc+14
---
   5: r1 = 0
   6: *(u32 *)(r10 -8) = r1
   7: r2 = r10
   8: r2 += -8
   9: r1 = map[id:279]
  11: call   htab map lookup_elem#294448
  12: if r0 == 0x0 goto pc+1
---
  13: r0 += 56
  14: r1 = r0
  15: r0 = 1
  16: if r1 != 0x0 goto pc+1
---
  17: r0 = 0
  18: r0 <<= 1
  19: exit
```

Reachability analysis - basic block

```
   0: r0 = 0
   1: r1 = map[id:281][0]+0
   3: r1 = *(u8 *)(r1 +0)
   4: if r1 == 0x0 goto pc+14
---
   5: r1 = 0
   6: *(u32 *)(r10 -8) = r1
   7: r2 = r10
   8: r2 += -8
   9: r1 = map[id:279]
  11: call   htab map lookup_elem#294448
  12: if r0 == 0x0 goto pc+1
---
  13: r0 += 56
---
  14: r1 = r0
---
  15: r0 = 1
  16: if r1 != 0x0 goto pc+1
---
  17: r0 = 0
---
  18: r0 <<= 1
---
  19: exit
```

Reachability analysis - basic block

```
--- #0 p:[], b:#6, f:#1
   0: r0 = 0
   1: r1 = map[id:281][0]+0
   3: r1 = *(u8 *)(r1 +0)
   4: if r1 == 0x0 goto pc+14
--- #1 p:[#0], b:#3, f:#2
   5: r1 = 0
   6: *(u32 *)(r10 -8) = r1
   7: r2 = r10
   8: r2 += -8
   9: r1 = map[id:279]
  11: call __htab_map_lookup_elem#294448
  12: if r0 == 0x0 goto pc+1
--- #2 p:[#1], f:#3
  13: r0 += 56
--- #3 p:[#1,#2], b:#5 f:#4
  14: r1 = r0
  15: r0 = 1
  16: if r1 != 0x0 goto pc+1
--- #4 p:[#3], f:#5
  17: r0 = 0
--- #5 p:[#3,#4], f:#6
  18: r0 <<= 1
--- #6 p:[#0,#5]
  19: exit
```



Reachability analysis - load time config

```
0: r0 = 0
; Get pointer to .rodata map
1: r1 = map[id:281][0]+0
; Deref variable at offset
3: r1 = *(u8 *)(r1 +0)
; Compare to some constant
4: if r1 == 0x0 goto pc+14
```

Reachability analysis - backtracking

```
--- #0 p:[], b:#55, f:#1
   0: r0 = 0
   1: r1 = map[id:281][0]+0
   3: r1 = *(u8 *)(r1 +0)
   4: r3 = 0
   5: r2 = *(u64 *)(r10 -16)
   6: if r2 == 0x123 goto pc+200
--- #1 p:[#0], b:#3, f:#2
   7: r3 = 1
   8: if r1 == 0x0 goto pc+14
```

Reachability analysis - sign extension

```
0: r0 = 0
; Get pointer to .rodata map
1: r1 = map[id:281][0]+0
; Deref variable at offset
3: r1 = *(u16 *)(r1 +0)
; Cast s16 to 64-bit
4: r1 <<= 48
5: r1 s>>= 48
; Compare to some constant
6: if r1 s> 0x0A goto pc+14
```

```
0: r0 = 0
; Get pointer to .rodata map
1: r1 = map[id:281][0]+0
; Deref variable at offset
3: r1 = *(u16 *)(r1 +0)
; Cast s16 to 32-bit
4: w1 <<= 24 ; new in ISAv3
5: w1 s>>= 24
; Compare to some constant
6: if w1 s> 0x0A goto pc+14
```

Reachability analysis - masks

```
0: r0 = 0
; Get pointer to .rodata map
1: r1 = map[id:281][0]+0
; Deref variable at offset
3: r1 = *(u16 *)(r1 +0)
4: r1 &= 0x01
; Compare to some constant
5: if r1 == 0x00 goto pc+14
```

```
0: r0 = 0
; Get pointer to .rodata map
1: r1 = map[id:281][0]+0
; Deref variable at offset
3: r1 = *(u16 *)(r1 +0)
; Mask and shift bitfield
4: r1 &= 0x04
5: r1 >>= 2
; Compare to some constant
6: if r1 == 0x00 goto pc+14
```

Reachability analysis - 64 bit constants

```
0: r0 = 0
; Get pointer to .rodata map
1: r1 = map[id:281][0]+0
; Deref variable at offset
3: r1 = *(u16 *)(r1 +0)
; LD64IMM, branching instructions have a 32-bit imm
4: r2 = 0xFFFFFFFFFFFFFFFF
; Compare register to register
6: if r1 == r2 goto pc+14
```

Reachability analysis - edge cases

```
  0: r0 = 0
  1: r1 = map[id:281][0]+0
  3: r1 = *(u16 *)(r1 +0)
  4: *(u16 *)(r10 -8) = r1
...
 80: r8 = *(u16 *)(r10 -8)
 81: if r8 == 0x00 goto pc+14
```

```
#define CONFIG(name)        \
(*({                        \
   void *out;               \
   asm volatile("%0 = " __stringify(name) " ll" \
           : "=r"(out));    \
   (typeof(name) *)out;     \
}))

if (CONFIG(enable_feature_a)) {
   //...
}
```

Conclusions / final notes

- Reachability analysis seems like a good tool for optimization
  - Reducing map creation and un-releasable maps
  - Reducing load time by pruning unused tail calls and global functions
- The verifier could be improved with regards to map refcounting
  - But even then this likely has a place
- This system has false negatives but no false positives
- Is this a Cilium specific use case? Or might this be useful elsewhere?
- Can we make signing work with this? (at some point in the future)

ISOVALENT
now part of CISCO

Questions?

ISOVALENT
now part of CISCO

Thank you!