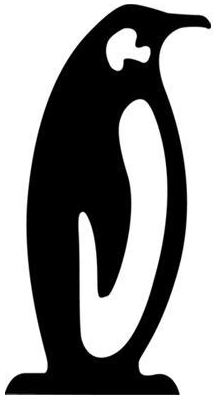


東京 2025 LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025

BPF Verifier Visualizer (bpfvv)

Ihor Solodrai
Jordan Rome

東京 2025 LINUX PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

Meta

Has this happened to you?

```
; if ((tuple->tcp_flag & TCP_FLAG_SYN) && !(tuple->tcp_flag & TCP_FLAG_ACK)) { @ Log.h:62
860: (61) r2 = *(u32 *)(r3 +40)       ; frame1: R2_w=scalar(smin=0,smax=umax=0xffffffff,var_off=(0x0; 0xffffffff)) R3=fp[0]-56 refs=53
861: (54) w2 &= 4608                  ; frame1: R2_w=scalar(smin=smin32=0,smax=umax=smax32=umax32=4608,var_off=(0x0; 0x1200)) refs=539
862: (56) if w2 != 0x200 goto pc+177          ; frame1: R2_w=512 refs=539
863: (bf) r7 = r3                     ; frame1: R3=fp[0]-56 R7_w=fp[0]-56 refs=539
864: (b4) w2 = 0                      ; frame1: R2_w=0 refs=539
; attach_type == BPF_CGROUP_INET_EGRESS @ Log.h:65
865: (16) if w1 == 0x1 goto pc+1 867: frame1: R0=0xffffffff R1=1 R2=0 R3=fp[0]-56 R4=6 R5=0 R7=fp[0]-56 R8=ctx() R10=fp0 refs=539
; attach_type == BPF_CGROUP_INET_EGRESS @ Log.h:65
867: (b7) r1 = 0                      ; frame1: R1_w=0 refs=539
; __builtin_memset(&log_event, 0, sizeof(log_event)); @ Log.h:19
868: (7b) *(u64 *)(r10 -88) = r1      ; frame1: R1_w=0 R10=fp0 fp-88_w=0 refs=539
869: (7b) *(u64 *)(r10 -24) = r1      ; frame1: R1_w=0 R10=fp0 fp-24_w=0 refs=539
870: (7b) *(u64 *)(r10 -32) = r1      ; frame1: R1_w=0 R10=fp0 fp-32_w=0 refs=539
871: (7b) *(u64 *)(r10 -40) = r1      ; frame1: R1_w=0 R10=fp0 fp-40_w=0 refs=539
872: (7b) *(u64 *)(r10 -48) = r1      ; frame1: R1_w=0 R10=fp0 fp-48_w=0 refs=539
873: (7b) *(u64 *)(r10 -56) = r1      ; frame1: R1_w=0 R10=fp0 fp-56_w=0 refs=539
874: (7b) *(u64 *)(r10 -64) = r1      ; frame1: R1_w=0 R10=fp0 fp-64_w=0 refs=539
875: (7b) *(u64 *)(r10 -72) = r1      ; frame1: R1_w=0 R10=fp0 fp-72_w=0 refs=539
876: (7b) *(u64 *)(r10 -80) = r1      ; frame1: R1_w=0 R10=fp0 fp-80_w=0 refs=539
; log_event.type = type; @ Log.h:20
877: (63) *(u32 *)(r10 -88) = r2      ; frame1: R2=0 R10=fp0 fp-88_w=mmmm0 refs=539
; log_event.timestamp = bpf_ktime_get_ns(); @ Log.h:21
878: (85) call bpf_ktime_get_ns#5     ; frame1: R0_w=scalar() refs=539
879: (bf) r6 = r0                     ; frame1: R0_w=scalar(id=25764) R6_w=scalar(id=25764) refs=539
880: (7b) *(u64 *)(r10 -80) = r6      ; frame1: R6_w=scalar(id=25764) R10=fp0 fp-80_w=scalar(id=25764) refs=539
881: (bf) r2 = r7                     ; frame1: R2_w=fp[0]-56 R7=fp[0]-56 refs=539
; log_event.protocol = tuple->protocol; @ Log.h:22
882: (71) r1 = *(u8 *)(r2 +38)
BPF program is too large. Processed 1000001 insn
processed 1000001 insns (limit 1000000) max_states_per_insn 7 total_states 77450 peak_states 679 mark_read 30
```

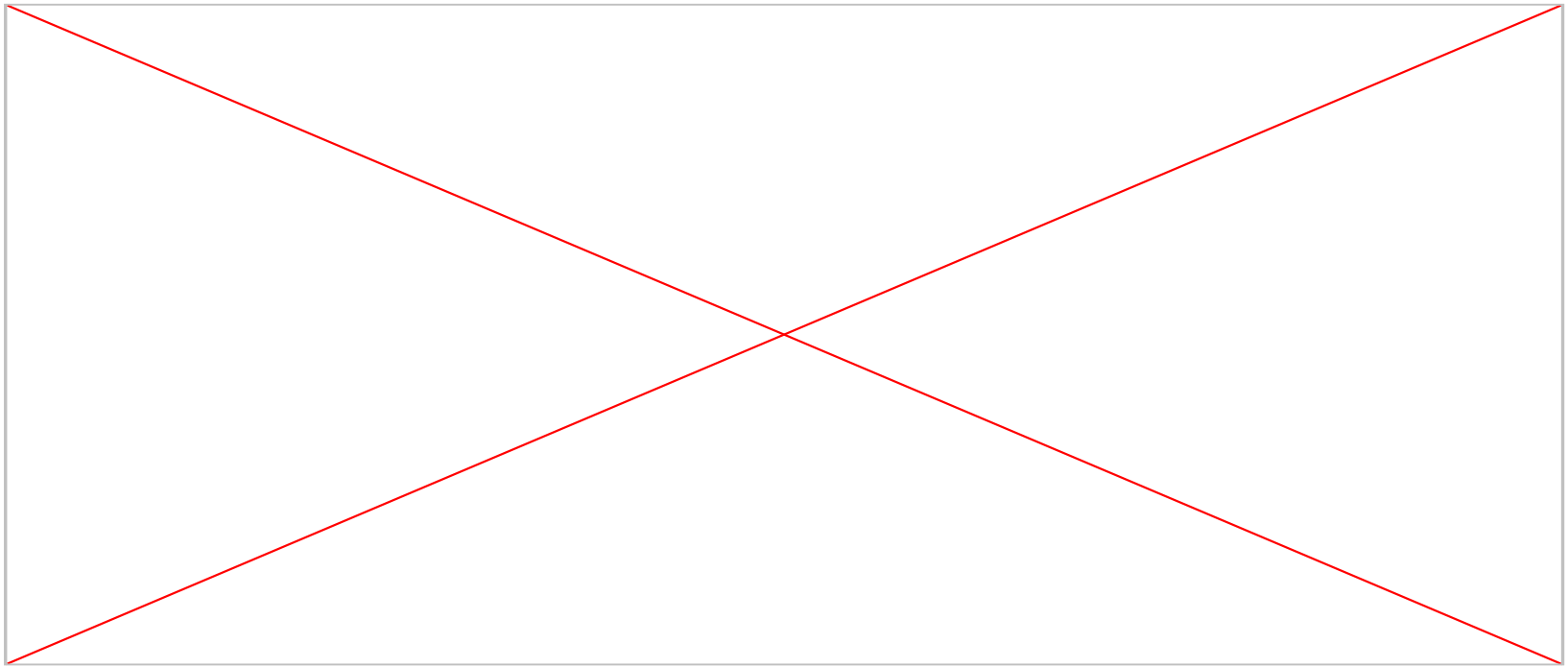
The Problem

The log is hard to understand

- BPF verifier is attempting to explain why your program is invalid
  - you might believe it is valid, but you can't argue with the verifier
- The explanation in the verifier log consists of:
  - a complete trace of the program evaluation leading to the invalid state
  - a short error message and stats
- To understand the trace, you must learn how to read it
  - it's a BPF instruction stream mixed with program state information, all in plain text
  - even after you learned how to read it, it's hard to keep the program state in your head (some logs are millions of lines long)

The Solution?

Make the verifier log easier to understand with a better UI (Demo)



How does it work?

- Parse the log into instruction stream and verifier reported values
- Re-construct a sequence of program states based on instruction semantics and verifier messages
- Compute a use-def analysis for registers and stack slots
- Expose this data to the UI components

UI Feature Spotlight

- Register and stack navigation in main log view
- Handle nested stack frames
- Local Storage of previous logs
- C source code view (pasteable)
- Fast scrolling no matter the size of the log
- Keyboard navigation
- Cross view highlighting and scroll-to
- Links to BPF helpers
- Collapsible views

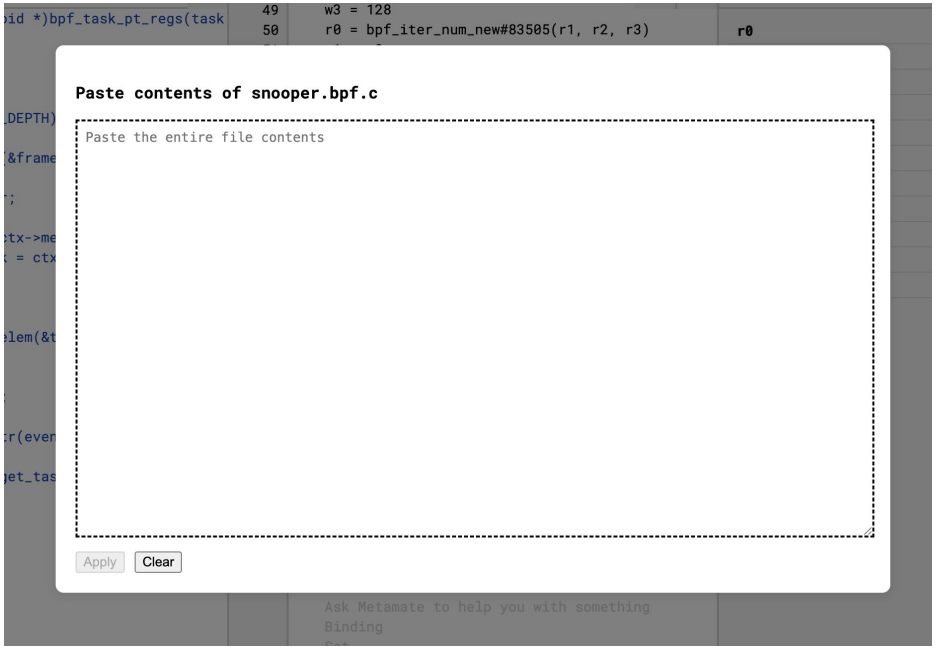
Register and Stack Navigation

```
25  if (r0 == 0x0) goto pc+79
26  r7 = 2
27  *(u64 *)(r0 +8) = r7
28  r1 = 1
29  *(u64 *)(r0 +0) = r1
30  r1 = 0xff434b28008e3de8
32  r9 = r0
33  r0 = bpf_spin_lock(lock: r1)
34  r2 = r6
35  r2 += 16
36  r1 = 0xff434b28008e3dd8
38  r3 = 0x53
40  r4 = 0
41  r5 = 0
42  r0 = bpf_rbtree_add_impl#54894(r1, r2, r3, r4,
43  r8 += 16
```

Nested Subprograms

```
3250        r1 = r0
3251        r3 = r8
3252        r2 = addr_space_cast(r1, 0, 1)
3253        w0 = 0
3254        if (r1 == 0x0) goto pc+163
3418        *(u64 *)(r2 +48) = r3
3419        *(u64 *)(r2 +40) = r7
3420        r3 = *(u64 *)(r10 -16)
3421        *(u64 *)(r2 +0) = r3
3422        *(u64 *)(r6 +0) = r1
3423        goto pc-42
3382      } exit ; return to stack frame 0
          returning from callee:
          frame1: R0=0 R1=0 R2=arena R3=scalar(id=5817) R6=arena R7=scalar(umin=1) R
          to caller at 2879:
          R0=0 R6=scalar(id=5800,umin=1) R7=0xfffffffea R10=fp0 fp-8=0xbadcafe
2879      w7 = w0
2880      if (w7 != 0x0) goto pc+8
2881      r1 = r6
2882      r2 = 0xbabababa
2884      r3 = 0
2885      w4 = 0
2886      pc+162() { ; enter new stack frame 1
            caller:
             R6=scalar(id=5800,umin=1) R7_w=0 R10=fp0 fp-8=0xbadcafe
            callee:
             frame1: R1_w=scalar(id=5800,umin=1) R2_w=0xbabababa R3_w=0 R4_w=0 R10=
            3049: frame1: R1=scalar(id=5800,umin=1) R2=0xbabababa R3=0 R4=0 R10=fp0
3049        r6 = addr_space_cast(r1, 0, 1)
3050        r7 = *(u64 *)(r6 +0)
3051        r1 = addr_space_cast(r7, 0, 1)
```

Pasting Actual Source Code

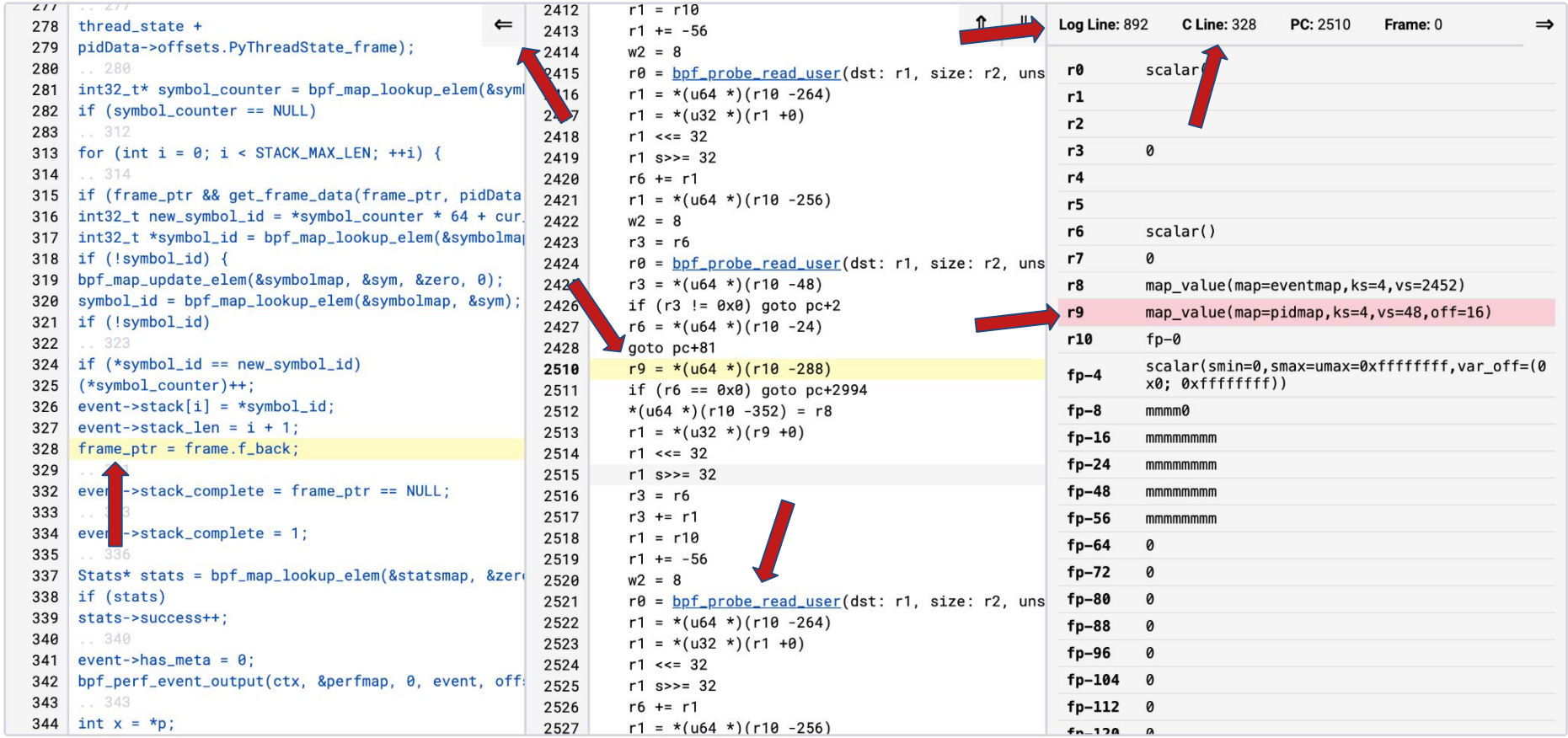


snooper.bpf.c

Click to replace lines with actual source code

```
42  .. 43
44  stack[0] = regs->ip;
45  fp = regs->bp;
46  bpf_for(i, 1, MAX_STACK_DEPTH) {
47  .. 47
48  if (bpf_probe_read_user(&frame, sizeof(frame), (vo
49  .. 50
51  stack[i]= frame.ret_addr;
52  .. 61
62  struct seq_file *seq = ctx->meta->seq;
63  struct task_struct *task = ctx->task;
64  .. 65
66  if (!task)
67  .. 68
69  event = bpf_map_lookup_elem(&task_event_buf, &zero
70  if (!event)
71  .. 72
73  event->pid = task->tgid;
74  event->tid = task->pid;
75  bpf_probe_read_kernel_str(event->comm, TASK_COMM_LI
76  .. 76
77  event->kstack_sz = bpf_get_task_stack(task, event-
```


CLICK ALL THE THINGS
imgflip.com


thread_state +
pidData->offsets.PyThreadState_frame);
int32_t* symbol_counter = bpf_map_lookup_elem(&syml
if (symbol_counter == NULL)
for (int i = 0; i < STACK_MAX_LEN; ++i) {
if (frame_ptr && get_frame_data(frame_ptr, pidData
int32_t new_symbol_id = *symbol_counter * 64 + cur
int32_t *symbol_id = bpf_map_lookup_elem(&symbolma
if (!symbol_id) {
bpf_map_update_elem(&symbolmap, &sym, &zero, 0);
symbol_id = bpf_map_lookup_elem(&symbolmap, &sym);
if (!symbol_id)
if (*symbol_id == new_symbol_id)
(*symbol_counter)++;
event->stack[i] = *symbol_id;
event->stack_len = i + 1;
frame_ptr = frame.f_back;
event->stack_complete = frame_ptr == NULL;
event->stack_complete = 1;
Stats* stats = bpf_map_lookup_elem(&statsmap, &zer
if (stats)
stats->success++;
event->has_meta = 0;
bpf_perf_event_output(ctx, &perfmap, 0, event, off
int x = *p;
r1 = r10
r1 += -56
w2 = 8
r0 = bpf_probe_read_user(dst: r1, size: r2, uns
r1 = *(u64 *)(r10 -264)
r1 = *(u32 *)(r1 +0)
r1 <<= 32
r1 s>>= 32
r6 += r1
r1 = *(u64 *)(r10 -256)
w2 = 8
r3 = r6
r0 = bpf_probe_read_user(dst: r1, size: r2, uns
r3 = *(u64 *)(r10 -48)
if (r3 != 0x0) goto pc+2
r6 = *(u64 *)(r10 -24)
goto pc+81
r9 = *(u64 *)(r10 -288)
if (r6 == 0x0) goto pc+2994
*(u64 *)(r10 -352) = r8
r1 = *(u32 *)(r9 +0)
r1 <<= 32
r1 s>>= 32
r3 = r6
r3 += r1
r1 = r10
r1 += -56
w2 = 8
r0 = bpf_probe_read_user(dst: r1, size: r2, uns
r1 = *(u64 *)(r10 -264)
r1 = *(u32 *)(r1 +0)
r1 <<= 32
r1 s>>= 32
r6 += r1
r1 = *(u64 *)(r10 -256)
Log Line: 892
C Line: 328
PC: 2510
Frame: 0
r0 scalar
r1
r2
r3 0
r4
r5
r6 scalar()
r7 0
r8 map_value(map=eventmap,ks=4,vs=2452)
r9 map_value(map=pidmap,ks=4,vs=48,off=16)
r10 fp-0
fp-4 scalar(smin=0,smax=umax=0xffffffff,var_off=(0x0; 0xffffffff))
fp-8 mmmm0
fp-16 mmmmmmmm
fp-24 mmmmmmmm
fp-48 mmmmmmmm
fp-56 mmmmmmmm
fp-64 0
fp-72 0
fp-80 0
fp-88 0
fp-96 0
fp-104 0
fp-112 0

Case Study
The Andrii Snooper

東京 2025
LINUX
PLUMBERS CONFERENCE
TOKYO, JAPAN / DEC. 11-13, 2025

```
47    r1 = r9
48    w2 = 1
49    w3 = 128
50    r0 = bpf_iter_num_new#83505(r1, r2, r3)
51    r1 = r9
52    r0 = bpf_iter_num_next#83507(r1)
53    w9 = 0
54    if (r0 == 0x0) goto pc+21
55    r9 = *(u32 *)(r0 +0)
56    w1 = w9
57    w1 += -1
58    if (w1 > 0x7e) goto pc+17
59    r1 = r10
60    r1 += -16
61    w2 = 16
62    r3 = r8
63    r0 = bpf_probe_read_user(dst: r1, size: r2, unsafe_ptr: r3)
64    if (r0 != 0x0) goto pc+11
65    r1 = r9
66    r1 <<= 3
67    r2 = r7
68    r2 += r1
69    r1 = *(u64 *)(r10 -8)
70    *(u64 *)(r2 +0) = r1
      R2 unbounded memory access, make sure to bounds check any such access
```

```
 *
 * We walk the chain of frame pointers to collect return addresses.
 */
static int unwind_user_stack(struct task_struct *task, __u64 *stack, int max_
{
        struct pt_regs *regs;
        struct frame {
                __u64 next_fp;    /* saved frame pointer (rbp) */
                __u64 ret_addr;   /* return address */
        } frame;
        __u64 fp;
        int i = 0;

        regs = bpf_core_cast((void *)bpf_task_pt_regs(task), struct pt_regs);
        if (!(regs->cs & 3))
                return 0;/* not in user space mode */

        stack[0] = regs->ip;
        fp = regs->bp;
        bpf_for(i, 1, MAX_STACK_DEPTH) {
                /* Read the frame: [fp] = next_fp, [fp+8] = ret_addr */
                if (bpf_probe_read_user(&frame, sizeof(frame), (void *)fp))
                        break;

                stack[i]= frame.ret_addr;
                fp = frame.next_fp;
        }

        return i * sizeof(__u8);
}
```

```
int i = 0;

bpf_for(i, 1, MAX_STACK_DEPTH) {
    // R2 unbounded memory access
    stack[i] = frame.ret_addr;
}
```

```
r2      map_value(map=task_event_buf,ks=4,vs=2080,off
        =1056,smin=0,smax=umax=0x7ffffff8,smax32=0x7
        ffffff8,umax32=0xffffff8,var_off=(0x0; 0x7ff
        ffffff8))
```

```
int i = 0;

bpf_for(i, 1, MAX_STACK_DEPTH) {
    // R2 unbounded memory access
    stack[i] = frame.ret_addr;
}
```

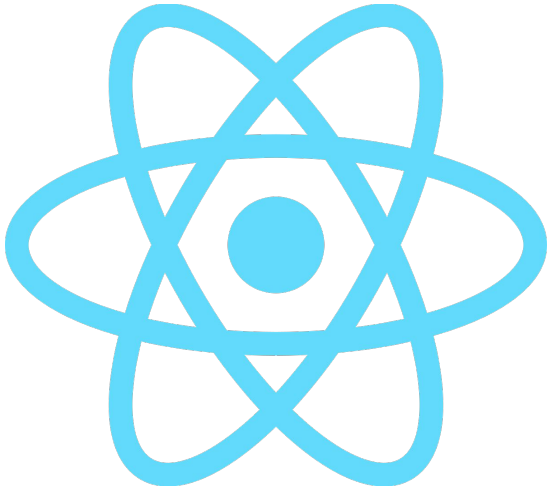
```
55    ┌─r9 = *(u32 *)(r0 +0)
56    ├─w1 = w9
57    ├─w1 += -1
58    └─if (w1 > 0x7e) goto pc+17
```

```
55    r9 = *(u32 *)(r0 +0)
56    w1 = w9
57    w1 += -1
58    if (w1 > 0x7e) goto pc+17
59    r1 = r10
60    r1 += -16
61    w2 = 16
62    r3 = r8
63    r0 = bpf_probe_read_user(dst: r1, size: r2, uns
64    if (r0 != 0x0) goto pc+11
65    r1 = r9
66    r1 <<= 3
67    r2 = r7
68    r2 += r1
69    r1 = *(u64 *)(r10 -8)
70    *(u64 *)(r2 +0) = r1
      R2 unbounded memory access, make sure to bounds
```

Internals

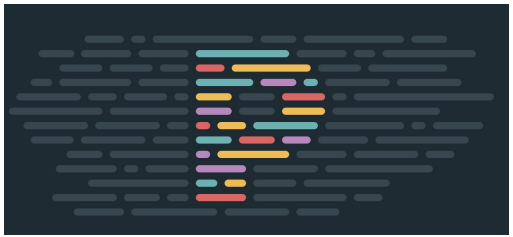
Raw to React

- Complex UI interactions
- Much better testing story (headless browser, snapshots, Jest)
- Large community of OS React components
- Rendering millions of log lines
- CSS is easier to manage
- Build systems



Shout Out to Dependencies

- Vite (for building and development)
- react-window (for rendering long lists of items)
- localdata
- Jest (for testing)
- Typescript
- Prettier (for formatting)



Meta CI Integration

- Internally hosted for Meta BPF programs
- Integration with Veristat
  - Every kernel commit we check against production BPF programs and create internal link to bpfvv if there is a failure

Resources

- The app: https://libbpf.github.io/bpfvv/
- Howto: https://github.com/libbpf/bpfvv/blob/master/HOWTO.md
- GitHub repo: https://github.com/libbpf/bpfvv