Motivating example 1. Consider the following program:

L1: bpf\_mov32 r2 0xffffffff  /\* r2 = (u32)0xffffffff \*/

L2: bpf\_jne r2 0xffffffff 2   /\* if (r2 == -1) \*/

L3: bpf\_mov64 r0 0.  /\* exit(0); \*/

L4: bpf\_exit

L5: ..    /\* else {... malicious code ...} \*/

Line L1 moves an unsigned 32-bit value 0xffffffff to the register r2, hence the static analysis must track the 64-bit register r2 as containing unsigned int 0x00000000ffffffff. However, a bug in a prior version of the Linux analyzer (CVE-2017-16995) sign-extended the 32-bit value, resulting in tracking r2 as containing signed int 0xffffffffffffffff [108, 7].

According to this buggy analyzer, the branch condition is always true and the code terminates with the if branch and the else branch is never analyzed. However, an actual execution would run the else branch. Hence, a malicious user could insert BPF instructions that read and write arbitrary kernel addresses in the else branch and yet successfully load the program.

Motivating example 2. Consider the program  
/\* assume r3 initialized with some unknown value, e.g., from a map \*/

L1: bpf\_mov32 r1 0xfffffff8 // r1 = (u32)0xfffffff8

L2: bpf\_jgt r3 0xf 3   // if (r3 <= 15) {

L3: bpf\_add r1 r3    // r1 = (u64)(r1 + r3);   
L4: bpf\_mov32 r2 0   // r2=0;

L5: bpf\_add32 r1 r2  // r1 = (u32)(r1 + r2);

After instruction L1, r1 is tracked to contain unsigned int 0x00000000fffffff8.

With the value of r3 bounded in the range [0, 0xf] after L2, the symbolic range of r1 is tracked to [0xffffffff8 ... 0x100000007] after instruction L3.

Instruction L5 converts r1 to a 32-bit value, which means that the correct value of r1 is either between [0x0, 0x7] or [0xfffffff8, 0xffffffff].

However, a mistaken value coercion in a prior version of the analyzer (CVE-2017-16996 [79]) simply intersected the existing bounds with the 32-bit range [0x0, 0xffffffff], resulting in the incorrect estimated bound for r1 [0xfffffff8, 0xffffffff].

A malicious user can now slip in arbitrary BPF code through the analyzer in the same way as in the first example.

A mailing list comment by kernel developer Jann Horn summarizes the problems well [77]: “the [BPF] range tracking is security-relevant for the verification of [BPF] code provided by unprivileged users. There- fore, any tiny slip-up in the arithmetic range tracking now turns into an arbitrary read+write in the full kernel address space, which is easily exploitable.” Avoiding such vulnerabilities requires the analyzer to capture the precise execution semantics of each instruction.