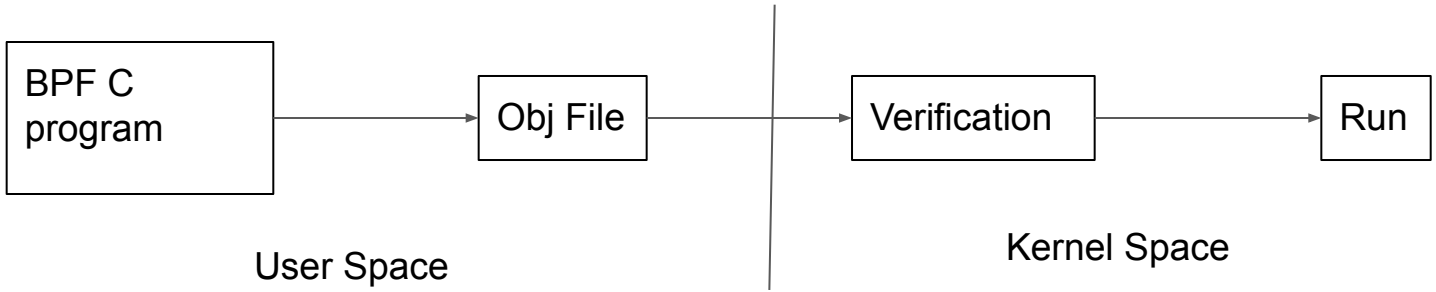
# Generating BPF Verifier Friendly Code with Clang

# BPF Prog Compilation and Verification

- Typical BPF prog is written in C
- Compile with clang/gcc and generate an object file
- The prog is loading into the kernel and get verified inside the kernel before executing the prog.
- The main goal of verification is to ensure that
  - The prog eventually will end, and
  - The prog won't cause kernel crash (e.g., illegal memory access caused memory crash, etc.)



# Linux Kernel Verifier - 1

- Flow/path sensitive: Every possible path is explored to ensure prog is sound
- A few techniques in verifier:
  - Maintain reg/stack states during path exploration.
  - Pruning:  if a path has been explored and covered by a previous exploration, skip it.
  - Simple liveness analysis to single out store without further load.
  - Some reg/stack states need precise range (e.g., for comparison, or some helpers), but some states does not care about the value (e.g., storing a value into the map).
- The reg/stack states are implemented with TNUM.

# Linux Kernel Verifier - 2

- Complete formal verification is NP-complete, also we want to
  - Fast verification time as sometime this is critical, e.g., replace a new prog with an old prog.
  - Control verifier complexity.
- **So verifier won't be able to maintain too much information (e.g., not straightforward relationships among registers/stack_slots) at any given moment, this may cause verification failure.**

# How to Prevent Verification Failure

- **Let clang/gcc generate verifier friendly code**
- In llvm BPF backend, some special passes are implemented which modify IR to prevent certain optimization which may cause verification failure
- **User modifies the source code**
- Source codes can be modified by changing code logic, adding asm code for force certain ordering, or using inline asm.
- Manual source code change can be minimized if LLVM can generate more verifier friendly code.

# An Exercise with No LLVM BPF Special passes

- https://reviews.llvm.org/D147968

```
/* Source */
...
id = ctx->protocol;
if (id < 4 || id > 12)
  return 0;
*(u64 *)((void *)v + id) = 0;
...
```

```
/* pseudo IR */
...
id = ctx->protocol;
tmp = id;
tmp += -13;
if (tmp < 0xfffffff7) goto next;
v += id;
*v = 0;
next:
```

Verification failure due to the ‘id’ range in ‘v += id’ is not known by verifier.

# Patching LLVM for Verifier Limitations In Old Kernels

- Sometimes, even if a verifier limitation is fixed in the latest kernel, it may be hard to backport. In such cases, it is a good idea to enhance LLVM to prevent certain code patterns as old kernel can use newer LLVM to compile bpf prog.
- https://reviews.llvm.org/D147968

```
for (i = 0; (i < VIRTIO_MAX_SGS) && (i < out_sgs); i++) {
        for (n = 0, sgp = get_sgp(sgs, i);
             sgp && (n < SG_MAX);
            sgp = __sg_next(sgp)) {
              bpf_probe_read_kernel(&len, sizeof(len),
                  &sgp->length);
              length1 += len;
              n++;
        }
}
```

```
upper = MIN(VIRTIO_MAX_SGS, out_sgs);
for (i = 0; i < upper; i++) {
        for (n = 0, sgp = get_sgp(sgs, i);
             sgp && (n < SG_MAX);
             sgp = __sg_next(sgp)) {
               bpf_probe_read_kernel(&len,
                  sizeof(len), &sgp->length);
               length1 += len;
               n++;
        }
}
```

# Source Code Change to Workaround Verification Failure

```
/* kernel BPF selftest: exhandler_kern.c */
 work = task->task_works;
 func = work->func;
 /* Currently verifier will fail for `btf_ptr |= btf_ptr` * instruction.
  * To workaround the issue, use barrier_var() and rewrite as below to
  * prevent compiler from generating verifier-unfriendly code.
  */
 barrier_var(work);
 if (work)
        return 0;
 barrier_var(func);
 if (func)
        return 0;
 exception_triggered++;
```

More examples can be found in kernel bpf selftests.

# How to Generate Verifier Friendly Code?

- Currently approach: LLVM BPF backend passes
  - We might be able to do more for some bpf selftest source hacks
- Alternative try in bpf upstream:
  - https://reviews.llvm.org/D147968
  - Try to provide TTI (TargetTransformInfo) hooks in the middle-end optimization so BPF backend can have a say whether a particular transformation should be done or not.
  - Rejected by upstream as IR legalization with BPF verifier requirement is preferred.
- Another approach:
  - An explicit option is provided with a list of transformations so those transformation can be disabled during optimization. Details to be decided.