User Space TCP based on LKL

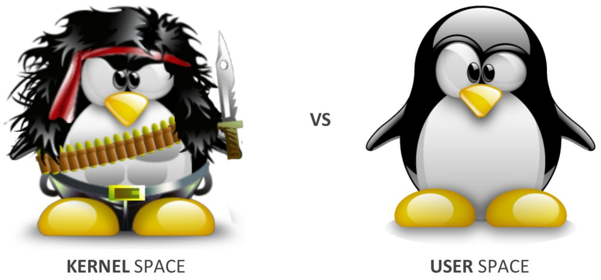
H.K. Jerry Chu, Yuan Liu, Andreas Abel

Google Inc.

Netdev 1.2 Conference Oct 5-7, 2016; Tokyo, Japan

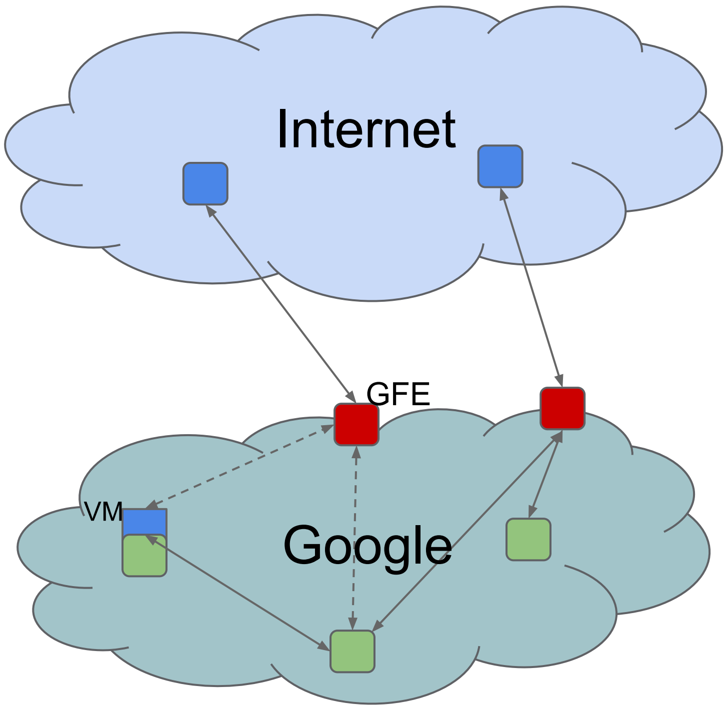
User-space TCP

- Traditionally, TCP stack in kernel space
- A TCP stack in user space can have advantages w.r.t.
    - µsec level latency performance (demanded by HPC, Wall Street,...)
    - Avoid kernel overhead - but kernel bypass often requires hardware assist



Cloud use case - terminate guest TCP conns to Google

- Tighter security
- Better isolation
  - Failure containment - single user process vs the whole kernel
- Release velocity
  - vulnerability can be patched quickly
- Accurate accounting
- Not for high performance (yet)



Existing user-space TCP stacks

- Many home grown user space TCP stacks inside Google
  - Most for specific use cases; fall apart when go beyond limited use
- Need a mature, high quality production-ready TCP stack
  - Interoperability, compatibility, maintainability,..., etc
- Commercial/open-source user-space TCP stacks often for high performance : OpenOnload libuinet lwIP Seastar ...

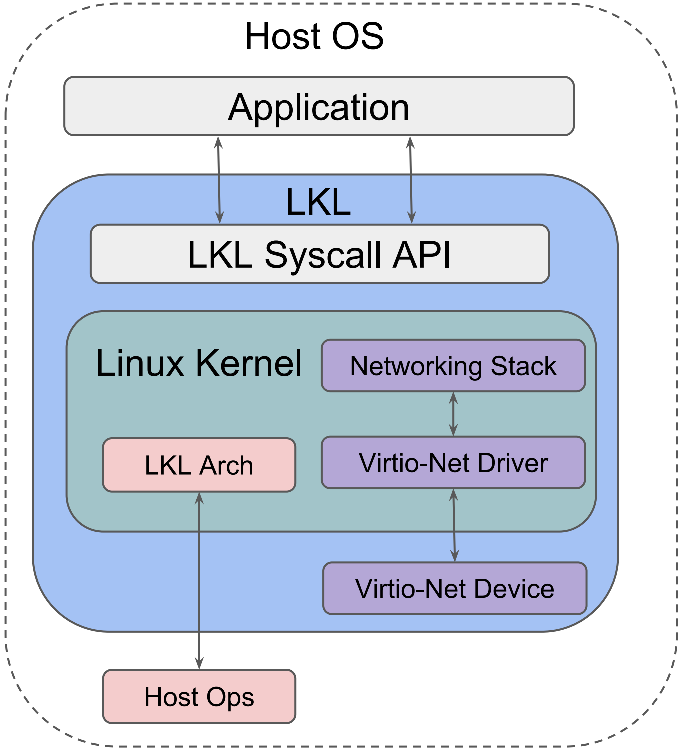
- Mature TCP stacks all kernel-based (Linux, BSD, Solaris,...)

How to run kernel code in user space?

- VM/hypervisor
- User Mode Linux (UML)
- Rump kernel (BSD)
- Extract only TCP code out of the kernel and stub around it
    - Need to separate code that intertwines with the rest of the kernel
    - Where to draw the boundary? (socket, IP, netdev,...)
    - Replacing interfaces to the rest of the kernel can get hairy (MM, synchronization, scheduler, IRQs,...)
    - LibOS?

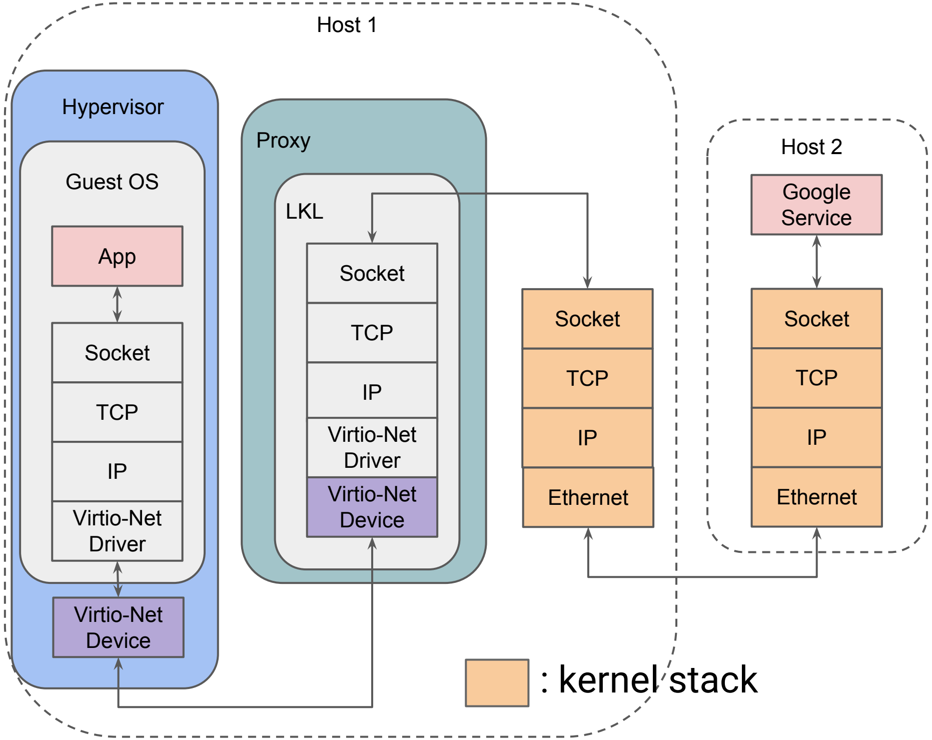
Linux Kernel Library

- Started by Octavian Purdila
- Designed as a port of Linux kernel
  - arch/lkl (~3500 lines of code)
  - LKL linked with apps to run in user space
- Relies on a set of host-ops provided by the host OS to function
  - semaphore, pthread, malloc, timer,...
- Well defined external interfaces
  - syscalls, virtio-net



Main use case - TCP proxy

- Terminates guest packets
- Proxies to a remote service
  - Can run any protocol the host supports
- May run the proxy remotely
  - Guest packets will be tunnelled through

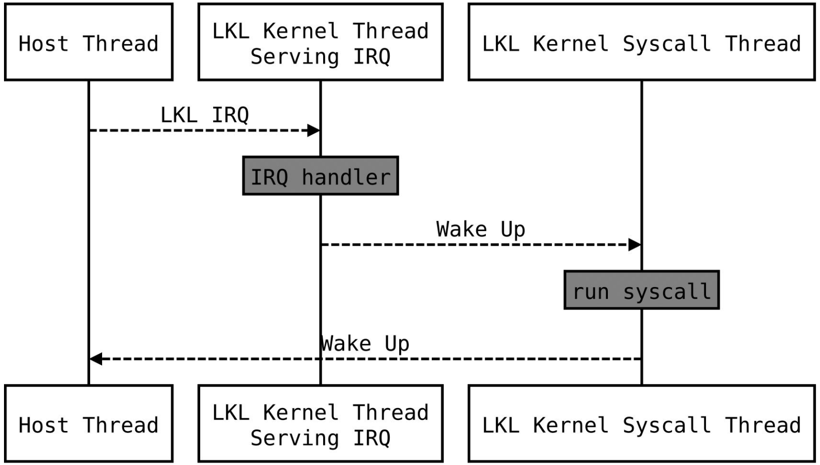


Architectural constraints

- App/host thread not recognized by LKL kernel scheduler
  - Can’t enter LKL to execute code directly - must wake up a LKL kernel thread to perform syscall on its behalf.
- User address allocated by host OS not recognized by LKL
  - syscalls into LKL kernel will fail when invoking address space operation
- no-MMU/FLATMEM architecture (va == pa)
  - No memory protection between app and LKL - both in the same space
- No SMP support
  - Entries into the LKL kernel (syscalls, irqs) must be serialized

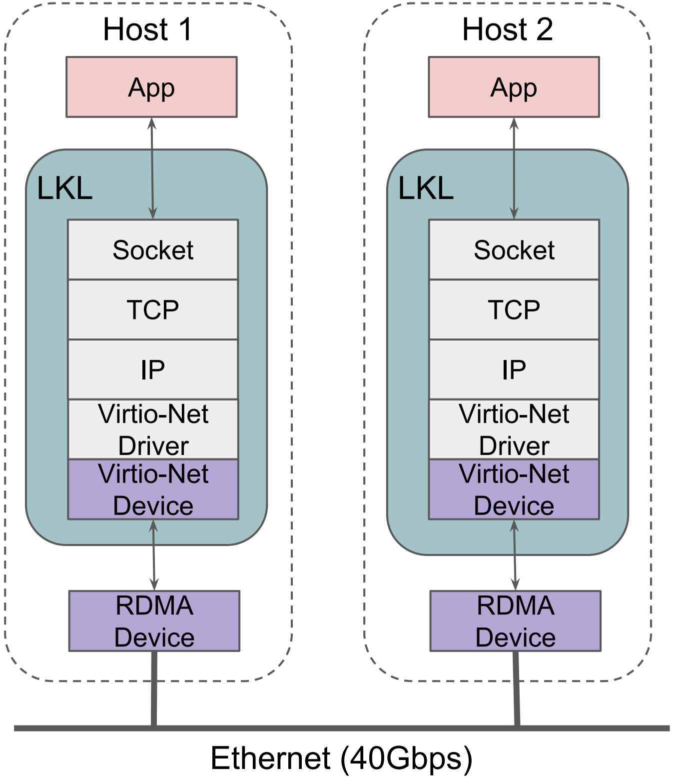
Getting latency down

- Significant latency overhead - three context switches to run one LKL syscall
- LKL *getppid(2)* takes 10 µs vs host 0.4 µs
- Solution: create a shadow LKL kernel thread and let host thread borrow shadow's *task_struct* to execute LKL syscall directly
- Blocking syscall: hack *__schedule()* to block the thread on a host semaphore
- getppid(2) down to 0.2 µs



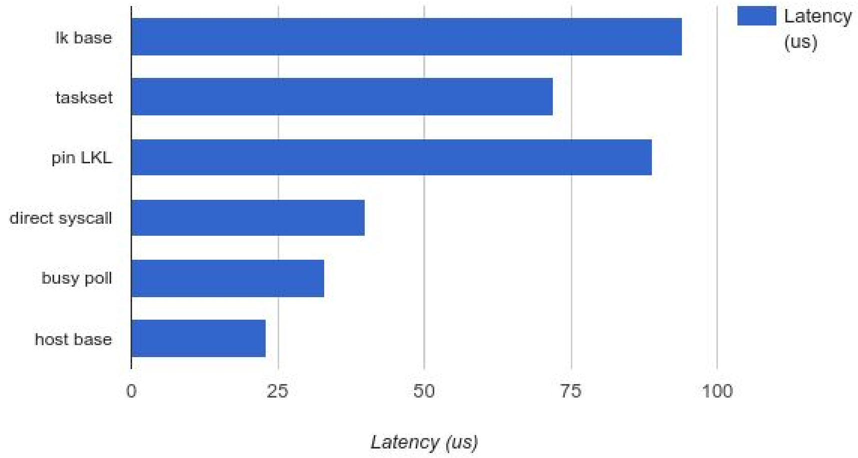
Networking performance - LKL vs host

- Runs LKL directly on top of NICs to bypass host kernel altogether
- LKL started at 5-10x slower than the host stack



Latency comparison against kernel stack

- 1-byte TCP_RR
- host stack baseline - 23 µs
- LKL busy poll - 33 µs (1.4X)
- w/o busy poll - 40 µs (1.8X)
- Gap to host: no hardware IRQ

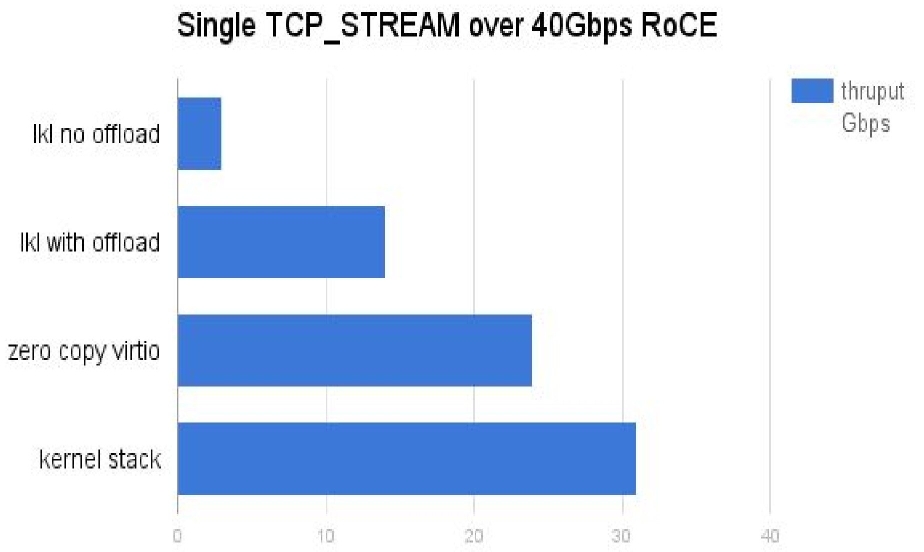


Boosting bulk data throughput

- Simple formula -> Large segments + csum offload
- GSO & GRO support already part of the kernel
  - LKL GSO alone doubles the thruput (one line change in virtio-net device code)
- GUEST/HOST_TSO requires virtio-net device support
- All flavors of offloads were added to LKL (incl. both "large-packet" and "mergeable-RX-buffer" modes)

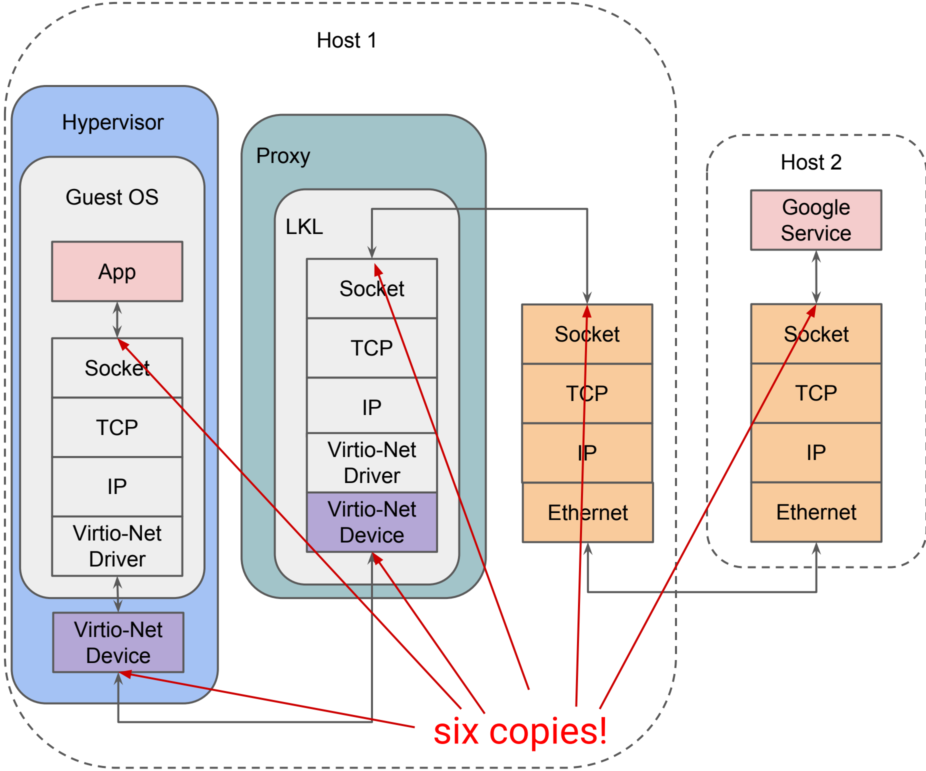
Thruput comparison against kernel stack

- LKL gets ~5x boost from the offload support
- Removing copy in virtio-net gets LKL within 75% of host
- LKL saturates ~1 CPU vs only 50% for the host
- LKL costs ~2.5x CPU cycles compared to host



Reducing copy overhead

- Copy is the simplest mechanism to move data
- But burns lots of CPU cycles (after offloads enabled)
    - ~30% CPU for TCP proxy
- Six copy operations for each byte transferred in TCP proxy



Zero-copy sockets - TX

- Same addr space & protection domain for user & LKL kernel
  - But kernel tracks physical pages (e.g., skb_frag_t) so not much easier (still needs to use API like vmsplice(2))
- Host allocated user address not recognized by LKL kernel
  - Syscalls involving addr space operation (e.g., vmsplice(2)) will fail
  - Solution - call LKL mmap(MAP_ANONYMOUS) to allocate buffer
- LKL needs to notify user when is safe to reuse a buffer
  - Has to ensure buffer not just ack'ed, but also freed to avoid security hole
  - Patches exist from willemb@google.com

Zero-copy socket - RX

- Returns skb from *sk_receive_queue* to the app directly
- App extracts data addresses from skb, e.g., use *page_address()* to convert *struct page* to pa (== va)
- App needs to deal with iovec of possibly odd size/unaligned buffers unfortunately (especially for “mergeable-RX-buffer”)
- Call back to LKL to free skb
- Changes to kernel code outside of arch/lkl
- Still WIP

Configuration/diagnosis tools

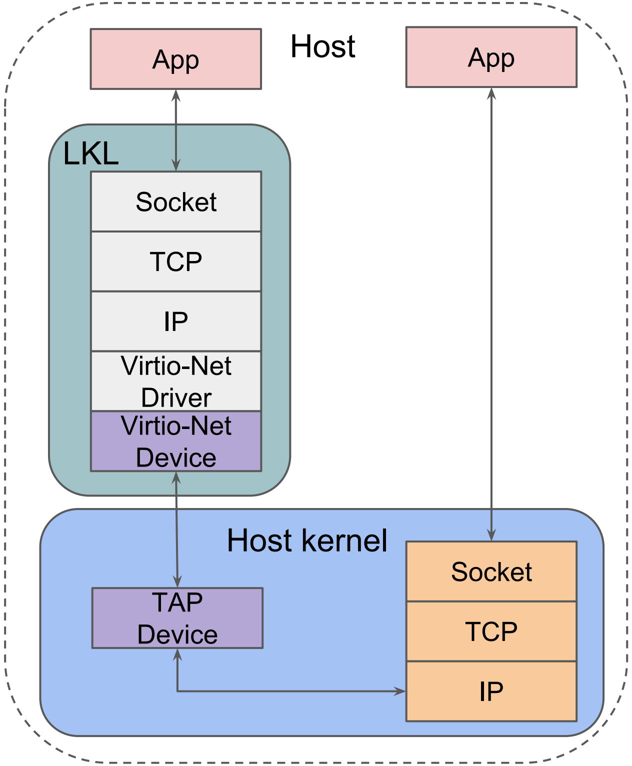
- Since LKL has all the kernel code, can we make various net-tools (ifconfig/ethtool/netstat/tcpdump/…) work?
- Constrained by a single process LKL is bounded
- A simple facility was added to spawn a thread providing a cmdline to mount procfs, sysfs, and retrieve counters, modify tunables,..., etc
- General solution - hijack syscalls from net-tools and execute in a remote LKL process, like *sysproxy* in *rump*

Questions?

Backup Slides

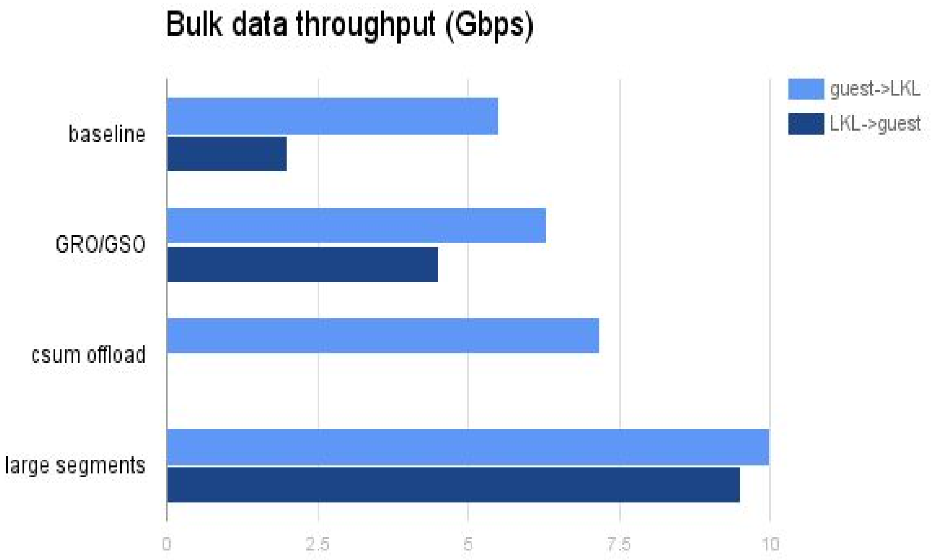
Testing configuration - tuntap to host kernel

- Easy to setup
- Packet injection to/from the host kernel can be expensive hence not good for production use
- Best for debugging or regression test purpose



Thruput for a local TCP proxy

- All offloads enabled on the guest side
- LKL GSO alone doubles the thruput (one line change in virtio-net device code)
- Optimal performance - large segment end-to-end w/o any csum calculation

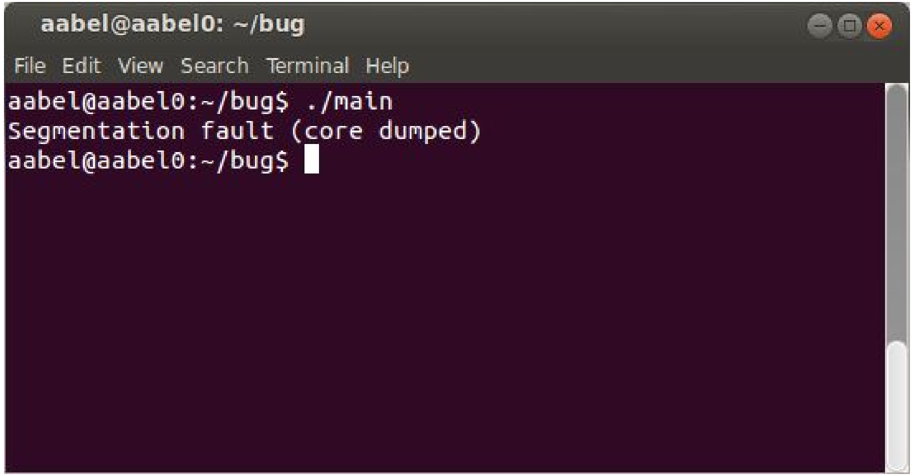


Dynamic Linker

- Loads shared libraries needed by an executable at run time
- Performs any necessary relocations
- Calls initialization functions provided by the dependencies
- Passes control to the application
- Kernel code compiled as shared library exposed to these bugs

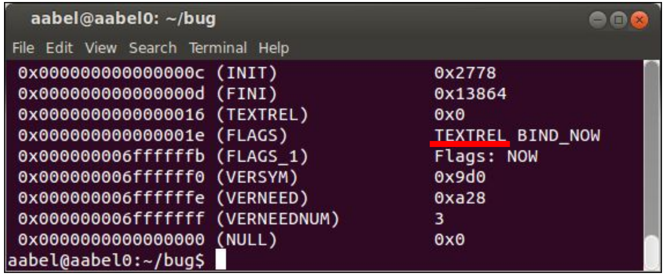
Linker/loader bugs

```c
#include <lkl.h>
#include <stdio.h>

int foo_v1 (void) { return 1; }

void * f_choice (void) {
  return foo_v1;
}

int foo (void) __attribute__ ((ifunc ("f_choice")));

int main() {
  printf("vfpv%d\n", foo());
  volatile int b = 0;
  if (b) {
    // never executed
    lkl_syscall(0, 0);
  }
  return 0;
}
```



TEXTREL (relocation in the text segment)

readelf -d:




```
 0x000000000000000c (INIT)               0x2778
 0x000000000000000d (FINI)               0x13864
 0x0000000000000016 (TEXTREL)            0x0
 0x000000000000001e (FLAGS)              TEXTREL BIND_NOW
 0x000000006ffffffb (FLAGS_1)            Flags: NOW
 0x000000006ffffff0 (VERSYM)             0x9d0
 0x000000006ffffffe (VERNEED)            0xa28
 0x000000006fffffff (VERNEEDNUM)         3
 0x0000000000000000 (NULL)               0x0
aabel@aabel0:~/bug$
```

- Shared library containing TEXTRELs can’t be shared anymore
- Text segment needs to be made writable - security issue (e.g., forbidden by SELinux)
- Android 6 does not support binaries with TEXTRELs.