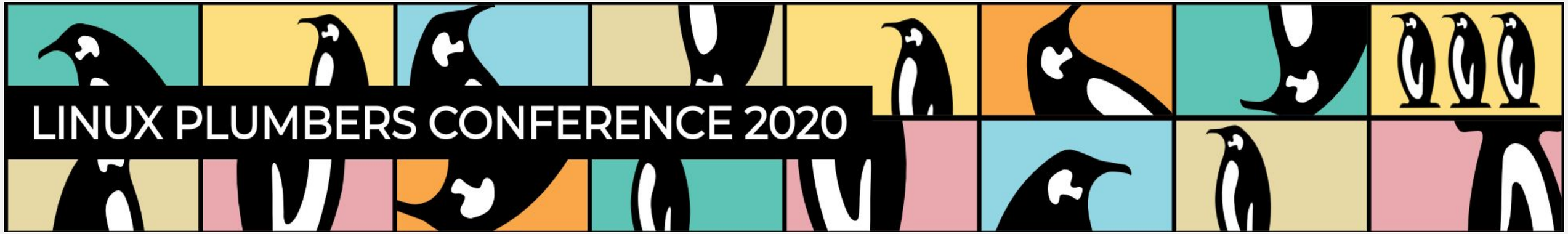


Android Networking 2020

(incl. eBPF use & Linux Kernel in general)

August 27, 2020

Previous slidedeck

There was a talk at the previous LPC 2019 in Lisbon at the Android micro-conference.

This will try to both refresh and extend upon that talk.

https://linuxplumbersconf.org/event/4/contributions/411/attachments/354/585/2019_LPC_Lisbon__eBPF_use_in_Android_Networking.pdf

Hopefully most of you haven't seen it (since it wasn't on the networking/bpf track)…

Even if you have, there's a fair bit of new stuff...

Who am I?

Maciej Żenczykowski (maze@google.com)

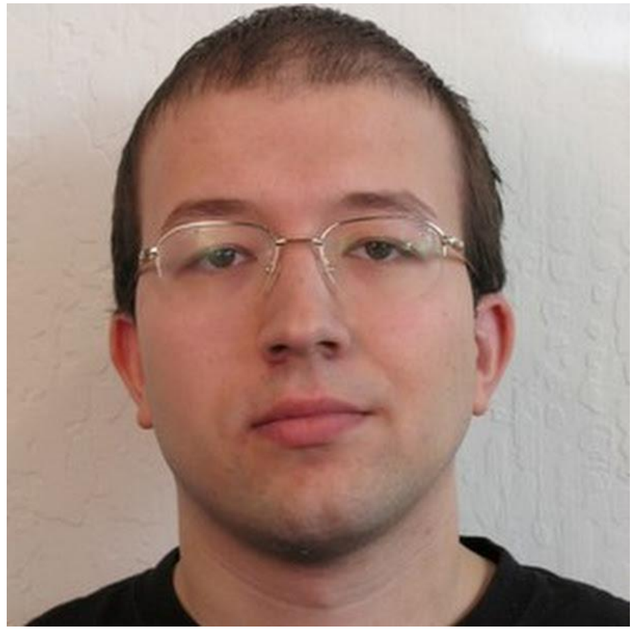
At Google since mid 2006, initially SRE.

2009-2018 Linux Kernel Networking on servers

(performance, tuning, configuration, frontend serving and load balancing, AnyIp, IPv6, 'glue')

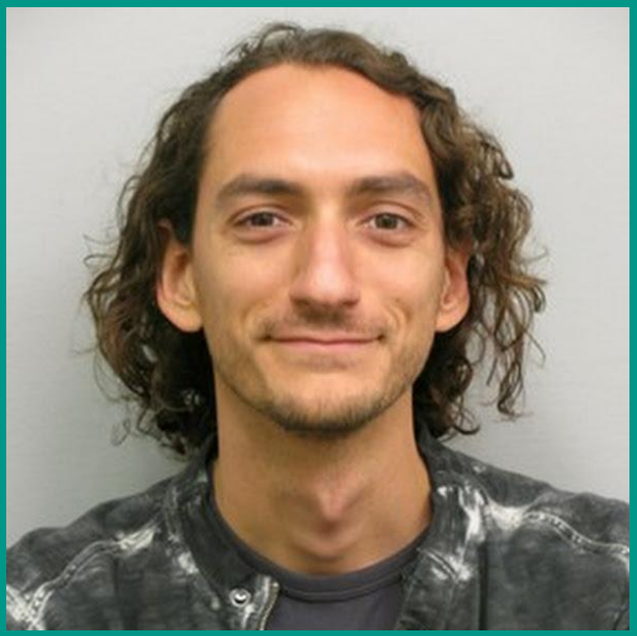
Since mid-2018 (**Q+**): **Android Core Networking**

With a focus on:
- *Kernel (upstreaming)*
- *eBPF & performance*
- *low level C stuff:*
  libs, core utils:
  iproute2, iptables,
  ethtool



android

My Team



Lorenzo Colitti (lorenzo@)

Manager of Android Core Networking (6.5 years)

Root cause of this work, code/design reviews, etc.

& others from:
Android Core Networking,
Pixel Kernel Team,
AOSP Kernel Team
with members in Mountain View, Taipei and Tokyo

In particular: ***Alistair Delva*** *(cuttlefish Android VM...) and* ***Greg Kroah-Hartman*** *(stable trees, etc.)*

Past work:
Chenbo Feng *(started eBPF, android P/Q dev work)*

Disclaimer: There's a lot of (kernel) networking...

And not a lot of people… (and most of them are working on higher levels of the stack)

As such out of necessity, most of my time is consumed by:

- *Exploration -- even 2 years later I still feel like I don't know the code base*
- *Maintenance work (ie. upgrading to newer versions of upstream kernel and/or cli's)*
- *Code cleanup / simplification / refactoring*
- *Keeping our tests passing*
- *Debugging various low level problems*
- *Helping OEMs debug breakages when they try to upgrade to newer Android releases*

There's not a lot of time left for future facing feature work.

Additionally device kernel upgrades take forever, so there's little immediate benefit.

We're pretty much forced into a 'consumer of upstream code' role (especially for kernel).

This is unfortunate, but it is what it is… eBPF is to a certain extent our saviour.

Why eBPF?

Because:

- **Get rid of code divergence vs upstream Linux**
  - moving kernel hacks into eBPF thus eliminating them (xt_qtaguid, Paranoid Android)
- **Performance improvements *(and simplicity)***
  - Clatd xlat464 user space daemon -> eBPF offload *(& it's actually easier!!!)*
  - Due to multi-network architecture Android has a really complex firewall + routing setup - this (and/or rndis performance) requires tethering to use HW offload, we hope to offload this to eBPF SW bypass and simplify things while improving performance.
- **More nuanced & more dynamic** (than capabilities like CAP_NET_ADMIN)
- Real hard to upgrade kernel on shipped devices, even new devices run old kernels...
- **It's cool** *(personal bias? I like low level stuff...)*

To quote another talk, cause: '***eBPF is eating the world***'

However...

Previous slide was written for last year's LPC conference...
In hindsight, with an extra year of experience, it's not all roses.
eBPF is still an overall win - but only because we can't actually realistically ship kernel upgrades.

As such, some things can only be done via eBPF...

But: eBPF is very hard to use:

- Hard to write (easier to write kernel code, often feels like writing ebpf requires more in depth knowledge of both the kernel and the bpf helpers, and the bpf verifier, and workarounds…)
- Very hard to debug - crazy verifier errors, tcpdump packet swallowing…
  - additionally in Android we currently suffer from a lack of tooling...
- eBPF isn't actually 'ready' - partially because forced into using old kernels (4.9)
  - in general feels like what you need is always available in the next (unreleased) version...
- Still have limitations due to old(er) kernels - for example R runs on eBPF-less 4.4, and eBPF capable (but to different levels) 4.9/4.14/4.19/5.4. Need to support multiple cases...

What we’ve done...

What we did in Android Pie (P)

First use of eBPF in Android - eBPF compilation (tool chain) support, bpfloader, etc.

Requires:

- Linux 4.9+ (for Kernel eBPF support) *←eBPF will finally be required by Android S*
- Devices launching with Android P
  - Treble means we can't require kernel changes on device upgrade,
    even if it's just enabling kernel config options or backporting fixes (policy may change).
  - Trouble figuring out if a 4.9+ device which launched with Android N/O actually supports eBPF or not - have to assume it does not.
- *for Google devices this means: Pixel 3+*

Replaced use of ancient 'xt_qtaguid' custom Android Linux kernel netfilter patch with eBPF bandwidth statistics collection. *(but was only actually reverted in Q kernels)*

Mostly done by fengc@ and predates me.

What we did in Android Pie (continued)

See 2017 LPC talks on this topic.

Replacing xt_qtaguid with an upstream eBPF implementation

https://blog.linuxplumbersconf.org/2017/ocw/sessions/4786.html

https://blog.linuxplumbersconf.org/2017/ocw//system/presentations/4786/original/Replacing%20xt_qtaguid%20with%20an%20upstream%20eBPF%20implementation.pdf

eBPF cgroup filters for data usage accounting on Android

https://blog.linuxplumbersconf.org/2017/ocw/sessions/4791.html

http://www.linuxplumbersconf.net/2017/ocw//system/presentations/4791/original/eBPF%20cgroup%20filters%20for%20data%20usage%20accounting%20on%20Android.pdf

What we did in Android 10 (Q)

‘Paranoid Android’ kernel patch: unpriv apps cannot create AF_INET/AF_INET6 sockets

- this is the app ‘internet’ permission
- now implemented in eBPF (fengc@), requires 4.14+ *(and thus Pixel 4+)*
- kernel patch not ported to Android common 4.19-Q and reverted in 4.14-R ← zombie effects
- *patch also had a feature auto-granting CAP_NET_ADMIN / CAP_NET_RAW based on process supplementary group membership, which couldn’t be replicated...*

Android has long supported **IPv6-only** networks (many cell networks are such for simplicity).
The ‘**Clatd**’ userspace daemon doing **464XLAT packet translation** was a **performance problem**.

- added RX-only TCP & UDP non-fragmented packet eBPF offload (maze@)
- requires 4.9+ with LTS fixes and config changes
- all Android Q devices launched on 4.9+ (and some launched on P)
- *ie. Pixel 3+ (launched on 4.9-P, kernel upgraded, now on 4.9-Q)*
- **tcpdump visibility hack**: tc ingress + bpf_redirect(same_ifindex, BPF_F_INGRESS)

See also Android Bootcamp 2019 slides.

Done in Android 11 (R), and plans for 12/S+

More CLAT:

- TX: normal tcp/udp non-fragmented packets *(this is the big performance win)* ← *for cellular, not wifi*
- RX & TX: ip fragments [*] *(covers large udp packets)*
- RX & TX: icmp packets - *complex and ugly*
- Hopefully eliminate need for userspace clat daemon entirely

[*] this may be hard to accomplish on 4.9…
bpf_skb_change_proto() only supports IPv4 [20] <-> IPv6 [40], not IPv4 frag [20] <-> IPv6 frag [40+8]
IPv6 -> IPv4 can cheat by using 8 bytes of NOP ip options...

But:
How to make a tx packet visible in tcpdump both pre and post mutation (tc egress)?

May not actually be worth fully completing before Android S, since Android R still needs to support 4.4 kernel devices and thus userspace clatd daemon has to exist on at least some devices anyway...

Done in Android 11 (R), and plans for 12/S+ (cont.)

Offload tethering, incl. NAT… (R: unidirectional ipv6 tethering offload from cellular to wifi/usb)

Android multi-network support causes a pretty complex routing and netfilter setup: presumed to be at least a cause of poor tethering performance (no multi-gigabit 5G speeds on a phone CPU in 3W power budget).

We'd like to get rid of the current reliance on hardware offload solutions:
hard to debug, and every device is different

Move to some sort of eBPF packet mangling/forwarding offload (or perhaps better call it bypass?).

Try to use XDP… *(belief: problems not core stack, but the drivers: no offloads, skb alloc/free, memcpy ahoy…)*

But: *there's so many different drivers…*
*Cellular (many...), wireless (many… some don't even do checksum offload), usb dongles, usb **rndis** (or cdc?)*

Enable BPF JIT *(basically an oversight it wasn't enabled, but CLANG CFI is currently incompatible) ← 64-bit only*

...but been busy dealing with Android Q fallout...

Done in Android 11 (R), and plans for 12/S+ (cont.)

In R significant (unplanned) work went in to improving the bpfloader:

- Support for bpf map uid/gid/chmod permissions (listed in the bpf.o) ← selinux?
- Program files (.o's) can be marked as critical, program functions as optional
- bpf functions can specify a [min,max) kernel version range, and fallback to alt implementation
- Bpfloader is no longer allowed to fail loading critical .o's (system reboot on bpfloader failure)
- Thus the rest of the system is able to depend on bpf progs/maps simply always existing

We never got to the bottom of why our bpfloader would randomly fail during boot...

Obviously there's *tons* of stuff leftover from R. No guarantees about anything...

XDP is looking … mighty challenging...

We do want to clean up all the old pre-eBPF support code.

Upstreaming progress

If you tease out the networking specific portions of ACK-mainline:

```
* 6577aef5022b BACKPORT: FROMLIST: net: introduce ip_local_unbindable_ports sysctl
* 1e50586e4b58 ANDROID: netfilter: xt_IDLETIMER: Add new netlink msg type [probably bad]
* 381a9e0fb6f0 ANDROID: netfilter: xt_quota2: adding the original quota2 from xtables-addons
* 0c2f8d019085 ANDROID: net: ipv6: autoconf routes into per-device tables
* 41d403d0bb8f ANDROID: net: bpf: permit redirect from ingress L3 to egress L2 devices at near max mtu
* 8bfabe54bfac ANDROID: xfrm: remove in_compat_syscall() checks [v2]
* 53bed55559c4 ANDROID: net: xfrm: make PF_KEY SHA256 use RFC-compliant truncation. [v2]
* ea5f6b1e1ec5 ANDROID: virt_wifi: fix export symbol types
* 8183677404c1 ANDROID: virt_wifi: Add data ops for scan data simulation
* 6eb2fd98c9a3 ANDROID: net: wireless: Add module_param(mac_prefix) to mac80211_hwsim
* bb3f3f760954 ANDROID: Unconditionally create bridge tracepoints
* 1987e390bef2 ANDROID: virtio-net: Skip set_features on non-cvq devices
* d012a7190fc1 (tag: remotes/linux/v5.9-rc2, tag: remotes/linux-stable/v5.9-rc2, linux-stable/master,
```

Android Common Kernel (Net) vs Upstream

Linus -> Stable/LTS -> Android Common -> Chipset Vendor (ie. Qualcomm) -> Device/OEM (ie. Google Pixel 4)

Android Common is many branches: {3.18, 4.4, 4.9, 4.14, 4.19, 5.4 LTS, mainline} x {O, O-MR1, P, Q, R, S, ...}
(though not every combination is valid, roughly 3 kernels per Android release)
Similarly **many** trees from chipset vendors, and *most* OEMs have one tree per device.
It's a true forest out there. *Though I understand the situation is slowly improving...*

Android Networking Tests (UML or QEMU based) pass on 5.9-rc2 + 5 + 3 + 3 = 11 networking changes

- 1 qemu virtio-net crash fix **[3 lines]**, 1 GKI change to unconditionally enable bridge tracepoints **[2 lines]**
- 3 patches for virtualized wifi **[around 85 lines]** ← *for cuttlefish Android VM*
- net: xfrm: make PF_KEY SHA256 use RFC-compliant truncation **[1 liner: 96 -> 128]**
- xfrm: remove in_compat_syscall() checks **[2x remove 'if (in_compat_syscall()) return -EOPNOTSUPP;']**
- net: bpf: permit redirect from ingress L3 to egress L2 devices at near max mtu **[3 lines, but need proper fix]**
- net: ipv6: autoconf routes into per-device tables **[4 files: 58 +, 39 -]** -> VRF? But also used by Chrome OS
- netfilter: xt_quota2: adding the original quota2 from xtables-addons **[4 files: ~450 +]** -> update xt_quota?
- netfilter: xt_IDLETIMER: Add new netlink msg type **[2 files: 245 +, 11 -]** -> simplify and upstream?

But: even further core networking stack divergence in chipset/device/OEM kernels…

Android Common Kernel (Net) vs Upstream

It is an explicit goal to upstream our stuff…

Trying our hardest to avoid adding more divergence:

in R we added (*just?*) 1 patch with 3 lines for ebpf (due to deadline time pressure).

We're basically down to 3 non trivial core changes:

- Netfilter - xt_IDLETIMER
  - I owe Pablo an upstreaming attempt… can't figure out what the requirements are...
- Netfilter - xt_quota2
  - Previously tried to update xt_quota to make it usable. Non trivial, need to try again.
- The ipv6 autoconf into multiple tables patch.

And a number of trivial few liners.

Challenges & Wishes...

Challenges

Security… *folks are paranoid… but for good reason, people really are out to get us...*

- loading only ebpf programs that are signed and/or from a dm-verity partition.
- no dynamically generated programs, but need to update maps (& constants?)...

Filtering and/or modifying netlink messages

- for example MAC addresses in Netlink Route messages *(‘ip link show’ output)*
- or kernel filtering of uevent to prevent spurious wakeups

Making it all work on old kernels

- or only on new ones and having extra support code for old ones

Huge variance from device to device (in chipset and vendor specific code)

- testing is a nightmare… *(and eBPF is hard to debug too…)*

Challenges

- Patches we revert can come back like zombies (via soc/vendor trees)…
  Need conformance tests for everything… much more work than the change itself.

- BPF verifier time complexity scaling… boot time regressions due to it...
  Maybe someway to reset it? To prevent 2**(number of branches) paths through the code?

- COVID has made access to new hardware running newer kernels even more painful than usual. Android R technically supports devices launching on 4.14/4.19/5.4 kernels, but all dev work was done on Pixel 4 (4.14) and cuttlefish VM (5.4).

- Poor eBPF handling of GSO frames. Bugs in proto4to6 and 6to4 conversion functions, lack of flexibility for ipv6+frag <-> ipv4 translation.

Wishlist...

- We don’t like patches, we don’t mind kernel config options…
  Could we merge a few #ifdef ANDROID or IS_ENABLED(ANDROID) things?
- Easier entrance bar to LTS for very simple ‘features’ and not only bug fixes,
  We’ve had some 1-line trivial patches that would have ideally gone in via LTS,
  but they were features…

- Create/use netns (and uts, mnt?) via NET_ADMIN without SYS_ADMIN

- Need for more ‘mark’ bits, mark32 -> mark64 ?
- or some sort of unpriv user settable stream/app id (for example for 5G slicing)

- Fix IPv6 privacy addresses vs mac address generation randomness…
  (changing MAC address while link down does not result in new IPv6 address on device up)

Wishlist...

- Finer grained bpf attach/use privs… selinux? Right to attach X to Y?
  Bpf prog/map access via uid/gid linux privs… add bpf fs selinux?

- eBPF syscall filtering with argument inspection (more syscalls takes ptrs to flags)

- eBPF better performance, resizable/autosizing BPF maps

- eBPF support for finding/changing TCP header options (advmss clamp, rwin)

- Perhaps some sort of skb verifier? Are offsets, checksums, etc correct?

- Something smarter wrt. multithreaded BPF map traversal and updates.
  - Perhaps ability to write to only part of a key's value?

Wishlist… light weight XDP?

Want a highspeed, low power clat/forwarding engine…

Devices/drivers: *tons*: cellular, wifi, bluetooth, ethernet, rndis, ncm, usb dongle
Some of these are per chipset, since they're drivers…
Some of these are offload-free and *non*-gso aggregating (usb) -> memcpy...

- TC ACT? but skb_alloc/free costs, little benefit above core stack
- XDP eBPF? But too many drivers to update, for tethering want to forward packets on a different interface and driver than received on, shared memory model?

Lightweight way to run XDP on a ptr,len buffer -> return header + skipped bytes.
Light XDP xmit? dev->send(ptr,len,ptr,len)

Multiple XDP progs on an interface?

Resources

Code location in AOSP: ***(development is entirely in AOSP, including code review visible in AOSP's Gerrit)***

https://android.googlesource.com/kernel/common/ (android-mainline, android-4.9-q, ...)

https://android.googlesource.com/kernel/configs/ (master branch)

https://android.googlesource.com/kernel/tests/ *(network tests only a.t.m.)*

https://android.googlesource.com/platform/system/bpf/ *(support, loader)*

https://android.googlesource.com/platform/system/netd/ *(In particular /bpf_progs/...)*

https://android.googlesource.com/platform/external/android-clat/ *(daemon)*

Docs

https://source.android.com/devices/tech/datausage/ebpf-traffic-monitor

Questions?

Thank You