XDP: the Distro View

Jiri Benc
Jesper Dangaard Brouer
Toke Høiland-Jørgensen

Linux Plumbers Conference
Lisbon, Sep 2019

Outline

In this talk we will give a distro view on XDP, and touch on related general eBPF topics.

- Enabling XDP: kernel config, required packages
- Supportability, bug reports handling
- Security considerations and hardening
- User experience and pain points
- Managing user expectations

Then we will look at some of the problems in depth.

Enabling XDP

Kernel config, required packages, testing.

Enabling XDP: kernel side

Straigthforward: CONFIG_BPF_SYSCALL=y

- XDP is always enabled
- Enable AF_XDP: CONFIG_XDP_SOCKETS=y
- Consider other networking BPF options:
  - CONFIG_CGROUP_BPF=y
  - CONFIG_NET_ACT_BPF=m
  - CONFIG_NET_CLS_BPF=m
  - CONFIG_BPF_STREAM_PARSER=y
  - CONFIG_LWTUNNEL_BPF=y

Enabling XDP: packages (1/2)

- Newest iproute2
- bpftool
  - Part of the kernel source code
  - But mostly independent
  - Similar to iproute2: no need for a dependency to a particular kernel version
- clang/llvm with bpf backend
  - BTF support is highly desirable
- pahole
  - Overloaded with BTF conversion code

Enabling XDP: packages (2/2)

- libbpf
  - Part of the kernel source code
  - Packageable as a library since kernel v5.1
  - Not much practical experience, yet

Enabling XDP: testing

- CONFIG_TEST_BPF=m
- tools/testing/selftests/bpf
  - Cumbersome to build and install
- samples/bpf
  - Needs custom installation script
  - Some samples do not work out of kernel tree
  - Not really usable for testing overall

eBPF supportability

Tools, bug reports, audit trail.

eBPF supportability: tools

Introspection needed. bpftool provides that.

- Essential to be installed on all systems.
- But provides only the current state, not the history.

sosreport tool

- Calls bpftool since v3.5.1.
- https://github.com/sosreport/sos

crash tool

- Understands eBPF since v7.2.2.
- https://github.com/crash-utility/crash

eBPF supportability: bug reports

The kernel behaves differently with BPF programs loaded.

How hard is it to debug a misbehaving system with buggy XDP programs loaded?

Need to teach support engineers to look for BPF programs.

- But that’s the usual thing with any new technology.

Distros need to create cheat sheets for users:

- What to look for if packets are disappearing (XDP, tc, etc.)
  - WiP: Drop monitor support for XDP
- What to look for if XDP programs are not working as expected.
- etc.

eBPF supportability: audit trail

bpftool provides only the current state. The BPF program that caused issues (e.g. packet drops) may not be loaded anymore.

Possible solution: enhancing the audit subsystem.

- Patches currently stuck due to disagreement between bpf and audit maintainers.

eBPF security

Hardening, unprivileged BPF.

eBPF hardening

Two major areas of possible problems:

1. Spectre class of hardware bugs.
2. Verifier bugs.

Hardening

- CONFIG_BPF_JIT_ALWAYS_ON=y to secure against malicious VMs.
- Unprivileged users may load BPF programs. Is that a problem?

Unprivileged BPF (1/2)

Considerations

- Verifier bugs may be dangerous.
- BPF has been used to ease creation of exploits of hardware bugs.
- BPF developers are considering switching off unprivileged BPF as default.

Turning off unprivileged BPF

- kernel.unprivileged_bpf_disabled=1
- No way to set this by default in upstream kernel.
- Needs to be set in a bootloader. Or use a distro specific patch.

Unprivileged BPF (2/2)

Problems

- Daemons manipulating maps need to be privileged.
- Even when only reading maps.
- Want to limit access to maps owned by other services.

Possible solution: access rights for maps?

- Proposed by Andy Lutomirski

XDP pain points

User, developer, distro problems; expectations and best practices.

User experience problems

- No readily available XDP solution packaged in distros.
  - “What? Do I need to be a programmer to use XDP?”
- tcpdump does not see all packets anymore.
  - XDP_DROP etc.
  - There’s no tcpdump-like feature for XDP.
- Interface statistics do not count all packets anymore.
  - “It must be something on the wire!”
- XDP programs do not reach the expected speed.
  - Because generic XDP is used.

Developer experience problems

- Packets can be silently dropped with XDP programs that are accepted as correct.
  - Because of using unimplemented features.
  - What are the available XDP features on the interface?
- XDP is not powerful enough.
  - Can’t send or duplicate packets.
  - “Where is a repository with XDP libraries I can use?”
  - “Okay, let’s use AF_XDP...” (later) “performance gotchas!”
- Verifier not smart enough.
  - It has gotten better, but may still reject valid programs

Distro experience problems (1/3)

- User wants to install these two packages. But both are using XDP!
  - Or user is using XDP for custom filtering. And distro is using XDP, too.
  - But only one XDP program per interface is supported.
- Great part of features untested on non-x86_64.
- Lack of community consensus on common libraries, build and devel environment.
  - Risk of too much fragmentation, unpolished user experience.
  - Example: iproute2 has its own bpf support code.
    - incompatible ELF map format
    - WIP: conversion to libbpf
  - Promote libbpf as the preferred solution?

Distro experience problems (2/3)

- libbpf
  - API in flux, including functions removal.
  - When built from the kernel, the package has the kernel version.
  - How much can be relied on libbpf repo on GitHub?
  - Distros need to link to the system version. “Vendoring” makes that hard.
- BTF and pahole
  - perhaps the BTF functionality should be split into a different tool?
  - kernel build and BTF: gcc should generate BTF

Distro experience problems (3/3)

- virtio_net supports XDP but the performance is limited.
  - Can we have XDP passthrough?
  - Can we have XDP offloading from VM to NIC?
  - What about VM migration?

User expectation

XDP has strong marketing. Everyone wants to use it.

- There are no ready to use solutions.
- Not enough features when trying to implement a custom solution.
- Turning to AF_XDP (because it is "XDP", isn't it?) and resulting disappointment.

Distros need to focus on developers and encourage them to develop XDP based solutions.

- Need more examples.
- Need best practices.
- Need education about limitations.

Examples and best practices

Kernel selftests/bpf and samples/bpf: bad starting point

XDP tutorial

- https://github.com/xdp-project/xdp-tutorial
- Easy build and devel environment.
- Easy to try out: uses veth and network name spaces.
- How to best package it in a distro?

XDP tools (planned)

- https://github.com/xdp-project/xdp-tools
- Best practices like the tutorial, but easier to re-use
- Shippable tools, usable out of the box; please contribute!
  - E.g., xdpdump, simple packet filter

Dive in: Multiple XDP programs on a single interface

Can we agree on a common way to do this?

Supporting multiple programs on one interface

XDP currently only supports one program per interface.

- So how to support multiple functions in sequence?
- Driving factors:
  - Debugging: Enable XDP and still be able to handle the support calls
  - Composability: User-defined XDP programs combined with packaged ones
    - E.g.: Run custom filtering, then XDP-enabled Suricata
- Today, multiple programs only possible through cooperative tail calls
  - Implemented differently across projects

Let’s look at a couple of examples of how this is done today...

Prior art #1: Katran xdp_root

Facebook's Katran LB has a mechanism for multi-program loading

- Each program cooperatively (tail) calls remaining progs in array

```
int xdp_root(struct xdp_md *ctx) { // installed on interface
  for (__u32 i = 0; i < ROOT_ARRAY_SIZE; i++) {
    bpf_tail_call(ctx, &root_array, i); // doesn't return when it succeeds
  }
  return XDP_PASS;
}
int xdp_prog_idx0(struct xdp_md *ctx) { // in root_array with idx=0
  for (__u32 i = 1; i < ROOT_ARRAY_SIZE; i++) { // start at 1!
    bpf_tail_call(ctx, &root_array, i); // doesn't return when it succeeds
  }
  return XDP_PASS;
}
```

Pros: Supports multiple programs with one map

Cons: Programs need to know their place in the sequence, no per-action hooks

Prior art #2: Cloudflare xdpdump

Cloudflare posted a xdpcap utility that can run after other XDP programs:

- Instrument your XDP return with tail-call per XDP 'action' code

```
struct bpf_map_def xdpcap_hook =  {
        .type = BPF_MAP_TYPE_PROG_ARRAY,
        .key_size = sizeof(int), .value_size = sizeof(int),
        .max_entries = 5 // one entry for each XDP action
};
int xdpcap_exit(struct xdp_md *ctx, void *hook_map, enum xdp_action action) {
    bpf_tail_call(ctx, hook_map, action); // doesn't return if it succeeds
    return action; // reached only if above tail-call failed (no prog installed)
}

int xdp_main(struct xdp_md *ctx) {  // program installed on interface
        return xdpcap_exit(ctx, &xdpcap_hook, XDP_PASS);
}
```

Pros: Different hook program per exit XDP 'action' code

Cons: Programs must include helper, needs one map per chain call

Limitations of current approaches

There are a couple of limitations we would like to overcome:

- Programs need to include tail call code
  - Needs cooperation from program authors
  - Incompatibility between approaches
  - Breaks if omitted by mistake (e.g., accidental return)
- Program order cannot be changed without recompilation
- Sysadmin cannot enforce policy
  - E.g., always run diagnostics program (such as xdpdump) first

Chain calling: design goals

High-level goal: execute multiple eBPF programs in a single XDP hook.

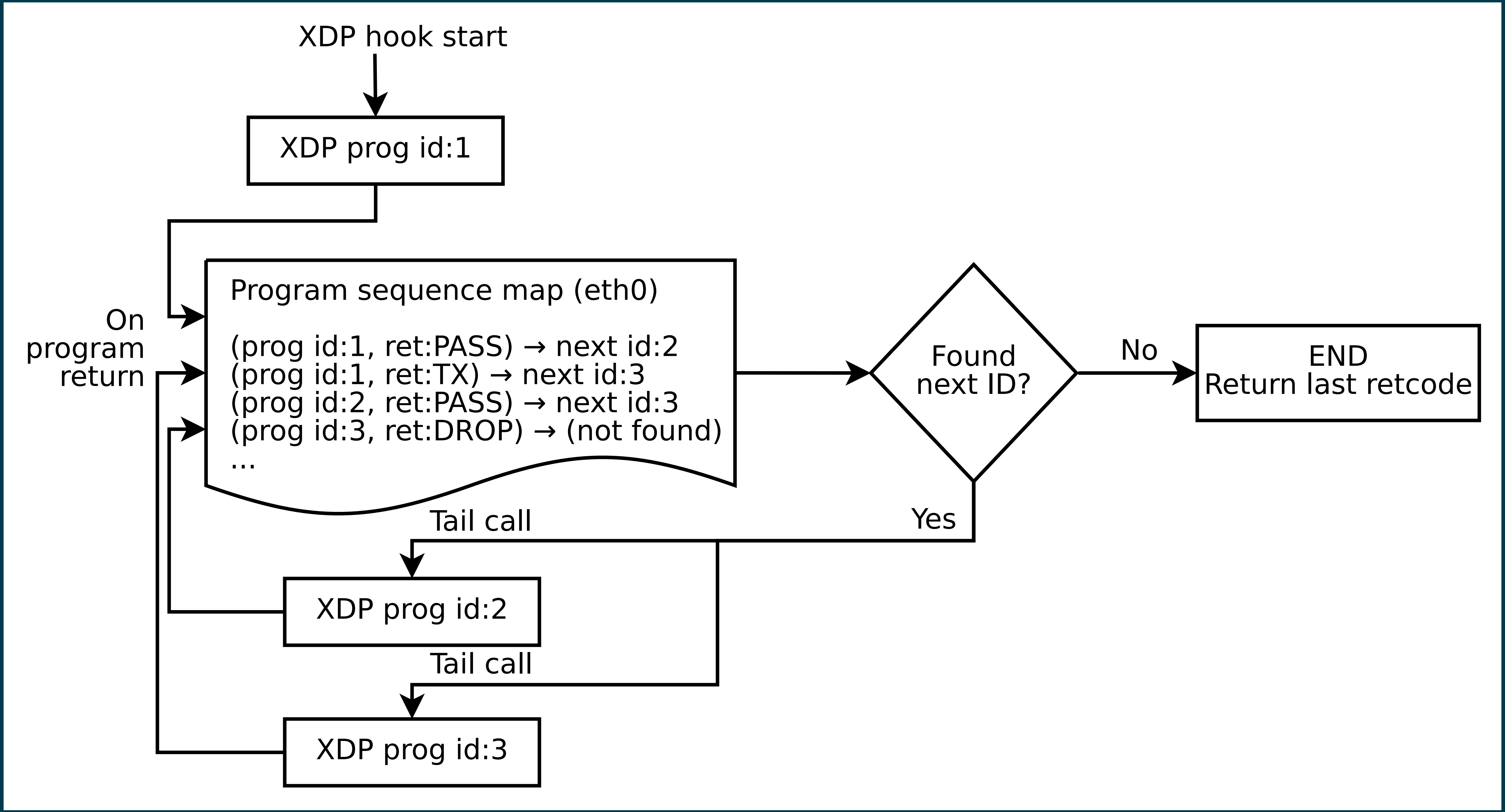
With the following features:

1. Arbitrary execution order
    - Must be possible to change the order dynamically
    - Execution chain can depend on program return code
2. Should work without modifying the programs themselves

Chain calling: Essential ideas

1. Per-interface data structure to define program sequence
   - Lookup current program ID and return code and get next program
   - Can be implemented with BPF maps
   - Similar to prior art #2, but one map for whole call chain
2. Add a hook at program return:
   - Either by rewriting program return instructions
   - Or by hooking into `bpf_prog_run_xdp()` in the kernel

Chain-calling: example execution flow


XDP hook start
XDP prog id:1
On program return
Program sequence map (eth0)
(prog id:1, ret:PASS) → next id:2
(prog id:1, ret:TX) → next id:3
(prog id:2, ret:PASS) → next id:3
(prog id:3, ret:DROP) → (not found)
...
Found next ID?
No
END
Return last retcode
Yes
Tail call
XDP prog id:2
Tail call
XDP prog id:3

Chain calling: Call sequence lookup helper

The chain call lookup could be implemented like this:

```
struct chain_call_lookup {
    unsigned int prog_id;
    unsigned int return_code;
};

int bpf_chain_call(ctx, retcode) {
  void *map = get_chain_call_map(ctx.ifindex);
  if (map) {
     struct chain_call_lookup key = {
       .prog_id = ctx.prog_id,
       .return_code = retcode
     };
     bpf_tail_call(ctx, map, &key); // doesn't return if successful
  }
  return retcode;
}
```

Chain calling: Call sequence lookup helper #2

The chain call lookup could also be implemented like this:

```
int bpf_chain_call(ctx, retcode) {
  void *map = get_chain_call_map(ctx.ifindex);
  if (map) {
     void *inner_map = bpf_map_lookup(map, &ctx.prog_id);
     if (inner_map)
       bpf_tail_call(ctx, inner_map, &retcode); // doesn't return if successful
  }
  return retcode;
}
```

Implementation option #1: userspace only

To do this in userspace (e.g., libbpf), the loader must:

1. Define `bpf_chain_call()` as bpf func
2. Create+pin outer map per ifindex
3. Populate map as XDP programs are loaded (key by prog tag?)
4. Rewrite RETURN instructions to call `bpf_chain_call()` before loading prog

Pros: No kernel support needed

Cons: Only enforceable if all loaders comply, lots of book-keeping, can’t swap map

Implementation option #2: Kernel verifier

In the kernel verifier:

1. Define `bpf_chain_call()` as BPF helper
2. Verifier rewrites return instructions to helper calls
3. Userspace populates per-ifindex call sequence map

Pros: Enforceable systemwide, uses existing tail call infrastructure

Cons: More code in already complex verifier

Implementation option #3: bpf_prog_run_xdp()

With kernel support in hook:

1. Make `bpf_chain_call()` a regular function
2. Call it before returning from `bpf_prog_run_xdp()`
3. Userspace populates per-ifindex call sequence map

Pros: Enforceable systemwide, no new verifier code

Cons: Multiple BPF invocations instead of tail calls, another check in fast path

Chain-calling: Updating the call sequence

- Simple updates: linked-list like operations (map stays the same)

```
# Insert after id 3
  --> id = load(prog.o);
  --> map_update(map, {3, PASS}, id) # atomic update
# Insert before id 2
  --> id = load(prog.o);
  --> map_update(map, {id, PASS}, 2); # no effect on chain sequence
  --> map_update(map, {1, PASS}, id); # atomic update
```

- More complex operations: replace the whole thing

```
# Replace ID 3 with new program
  --> id = load(prog.o); map = new_map();
  --> map_update(map, {1, PASS}, 2);
  --> map_update(map, {1, TX}, id);
  --> map_update(map, {2, PASS}, id);
  --> xdp_attach(eth0, 1, map, FORCE); # atomic replace
```

We want atomic updates; how to manage read-modify-update races?

Dive in: Missing XDP feature detection

How do we ensure programs will work if loading succeeds?

Builtin versus drivers

XDP features dependent on driver support, which breaks BPF feature "system"

- BPF-core is always compiled-in
- BPF verifier will reject BPF prog
  - if using a feature that isn't available in BPF core

XDP challenges this concept.

The XDP available features issue

Today: Users cannot know if a device driver supports XDP or not

- This is the question asked most often
- And people will often use generic XDP without noticing,
  - and complain about performance... this is a support issue.

Real users requesting this:

- Suricata config want to query for XDP support, else fallback to BPF-TC
- VM migration want to query for XDP support, else need to abort migration

Original argument: Drivers MUST support all XDP features

- Thus, there is no reason to expose feature bits
- This was never true, and e.g. very few drivers support redirect

What is the real issue?!?

Simply exposing feature XDP to userspace, doesn't solve the real issue

- Real issue: too easy to misconfigure
- How to get users to check features before attach? (unlikely to happen)

Real issue: Kernel allows users to attach XDP program

- that uses features the driver doesn't implement
- causes silent drops (only way to debug is tracepoints)

Solution: Need something that can reject earlier

- at BPF load or XDP attach time
- BPF verifier rejects at BPF load time (doesn't see attach operation)
  - (if using a feature that isn't available in BPF core)

Tech road-block: BPF tail-calls vs attach-time

Solution #1: Do feature match/check at XDP driver attach time

- Reject attach, if prog uses unsupported features
- Not possible due to BPF tail-call maps

Essentially tail-call maps adds attach “hook” outside driver control

1. Driver XDP prog tail-calls into prog map
2. Tail-prog calls into another (2nd level) prog map
3. Later 2nd level map is updated
    - with new program using unsupported feature

How can driver reject this 2nd level map insert?!?

Solution #2: BPF load time with ifindex (1/2)

Solution#2: Do feature match/check at BPF load time

- Supply ifindex at BPF load time (like HW-offload already does!)

Issue-2A: what if ifindex bound XDP-prog uses tail-call map

- How to check features of programs inserted into tail-call map?
- Solution-2A: Bind tail-call map to ifindex
  - And on tail-call map insert, BPF prog must be ifindex bound too
  - Require: bound prog, must only use bound tail-map (same ifindex)
- Limitations: cannot share tail-call maps (any real users?)
- Opt-in interface via supplying ifindex
  - Have to support loading with no ifindex, due to backwards compatibility

Solution #2: BPF-load time with ifindex (2/2)

Issue-2B: Generic XDP

- At BPF load time, don't know if used for native or generic XDP

Generic XDP support should be independent of net device

- Still, some XDP features are not supported
  - e.g. cpumap redirect (silent drop)

Possible solutions

- Option(1) supply more info than ifindex?
  - Annoying for API perspective
- Option(2) let ifindex imply native XDP?
  - Force generic-XDP to implement all XDP features (with some fallback)

Discussion: Expressing XDP features

OK, let’s suppose we agree on how to check for feature support.

But how do we express the features themselves?

Can verifier detect XDP features?

Either need to supply features (more input than `ifindex`)

- Or verifier needs to be able to detect features

Verifier detection strategy, to deduce XDP features in use

- If XDP return code comes from register/map
  - then assume all XDP return codes in use
- Except: can remove XDP_REDIRECT if redirect helper isn't used
  - And assume remaining codes are in use

What kind of XDP features to express?

Obvious feature: XDP return codes in use

Some BPF helpers can depend on driver feature

- `bpf_xdp_adjust_meta()` depend on driver feature
  - Today fails at runtime (we can do better!)
- `bpf_xdp_adjust_tail()` relevant to know for multi-buffer support

Verifier can easily detect BPF helpers in use

How to expose XDP features to userspace?

Highly prefer verifier detect features

- Pros: Avoids defining UAPI, thus easier to extend
- Cons: Userspace cannot easily get XDP feature bits from NIC

Driver needs to express feature bits internally.

How do userspace see what NIC supports? Two options:

- (1) Expose driver feature bits (needs some kind of UAPI; ethtool?)
- (2) Do feature probing like bpftool

Questions, comments?

Or did we get through them all on the way?