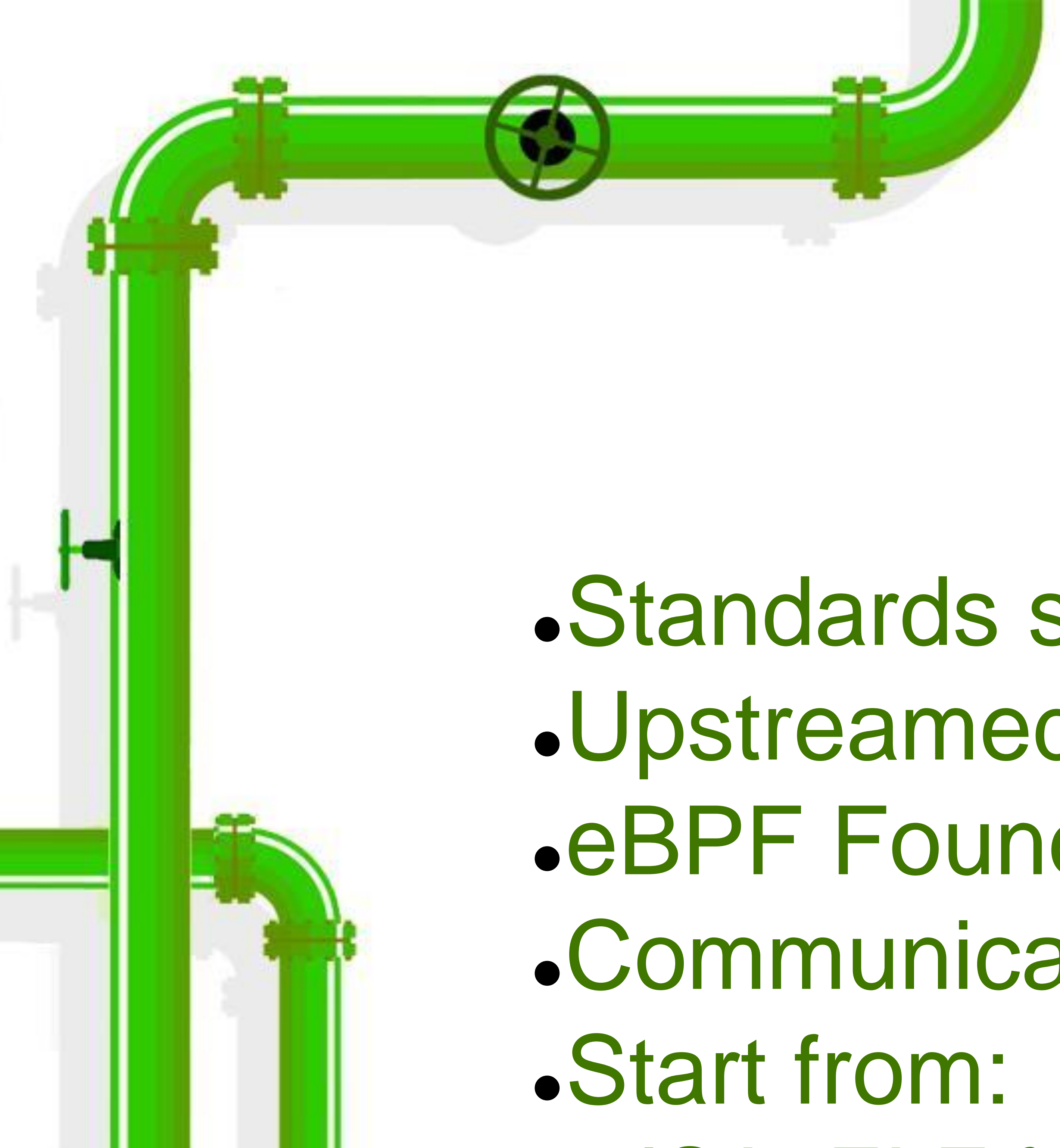
Linux
Plumbers
Conference
Dublin, Ireland September 12-14, 2022

eBPF Standardization

Dave Thaler <dthaler@microsoft.com>



- Standards should apply across platforms/implementations
- Upstreamed to Linux kernel tree
- eBPF Foundation should publish as PDF with version numbers
- Communication on bpf mailing list
- Start from:
  - ISA, ELF format, BTF (load time vs debug/metadata info), some verifier expectations in terms of what ISA covers vs not, compiler expectations
- Ok to point to other docs for things out of scope for now
  - Prog types, map types, verifier expectations (e.g., halting test)

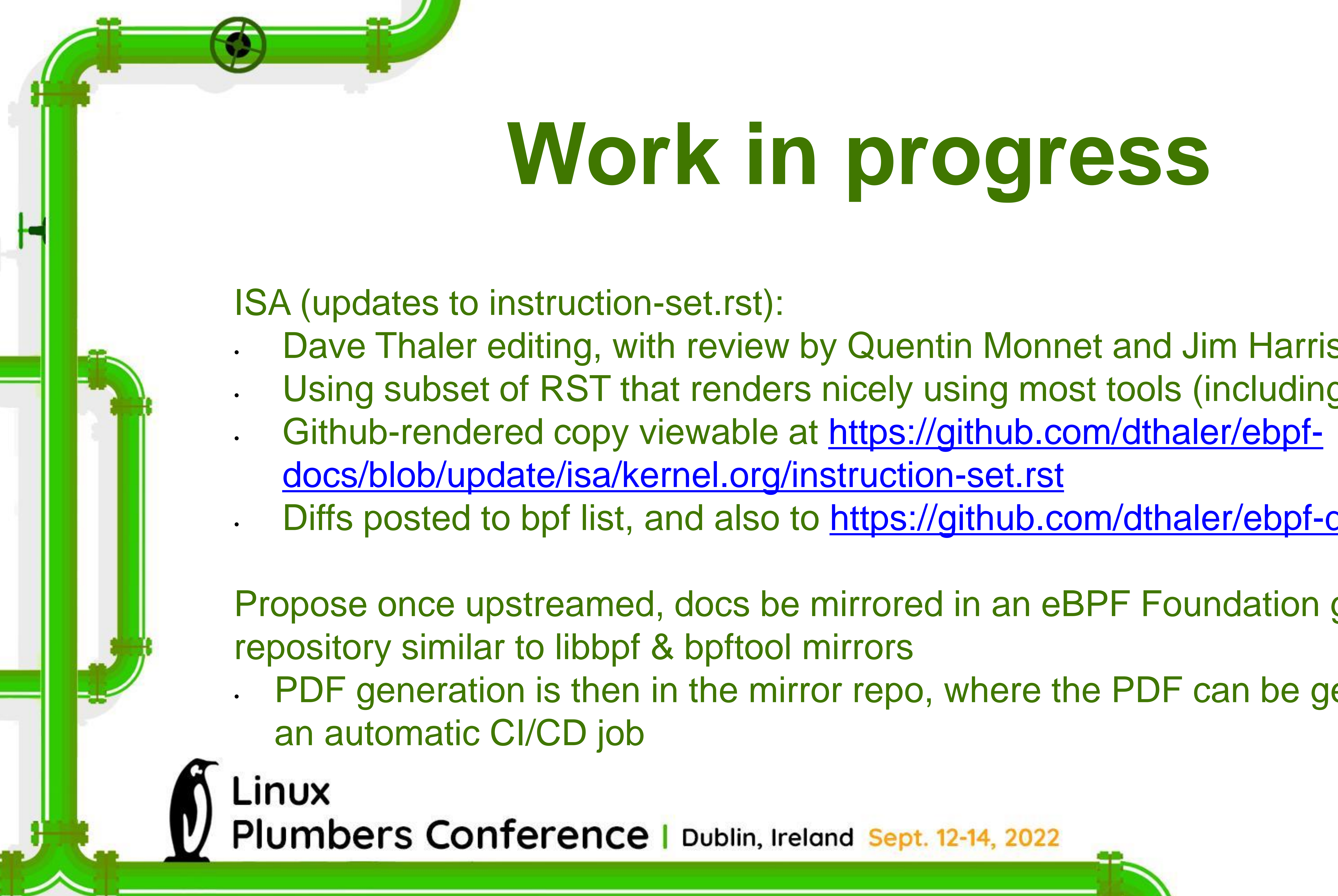
Work in progress

ISA (updates to instruction-set.rst):

- Dave Thaler editing, with review by Quentin Monnet and Jim Harris
- Using subset of RST that renders nicely using most tools (including github)
- Github-rendered copy viewable at https://github.com/dthaler/ebpf-docs/blob/update/isa/kernel.org/instruction-set.rst
- Diffs posted to bpf list, and also to https://github.com/dthaler/ebpf-docs/pull/4

Propose once upstreamed, docs be mirrored in an eBPF Foundation github repository similar to libbpf & bpftool mirrors

- PDF generation is then in the mirror repo, where the PDF can be generated via an automatic CI/CD job



ISA Table of Contents

- 1 eBPF Instruction Set
  - 1.1 Versions
  - 1.2 Registers and calling convention
  - 1.3 Instruction encoding
    - 1.3.1 Instruction classes
  - 1.4 Arithmetic and jump instructions
    - 1.4.1 Arithmetic instructions
      - 1.4.1.1 Byte swap instructions
    - 1.4.2 Jump instructions
      - 1.4.2.1 Helper functions
  - 1.5 Load and store instructions
    - 1.5.1 Regular load and store operations
    - 1.5.2 Atomic operations
    - 1.5.3 64-bit immediate instructions
    - 1.5.4 Legacy BPF Packet access instructions
  - 1.6 Appendix

Arithmetic operations

Proposed text:

> Underflow and overflow are allowed during arithmetic operations, meaning the 64-bit or 32-bit value will wrap.
>
> BPF_DIV has an implicit program exit condition as well. If eBPF program execution would result in division by zero, program execution must be gracefully aborted.

But meaning of “gracefully aborted” is not yet defined. Similarly previous doc had

> When an eBPF program is trying to access the data beyond the packet boundary, the program execution will be aborted.

1.6 Appendix

For reference, the following table lists opcodes in order by value.

opcode	imm	description	reference
0x04	any	dst = (uint32_t)(dst + imm)	Arithmetic instructions
0x05	0x00	goto +offset	Jump instructions
0x07	any	dst += imm	Arithmetic instructions
0x0c	0x00	dst = (uint32_t)(dst + src)	Arithmetic instructions
0x0f	0x00	dst += src	Arithmetic instructions
0x14	any	dst = (uint32_t)(dst - imm)	Arithmetic instructions
0x15	any	if dst == imm goto +offset	Jump instructions
0x16	any	if (uint32_t)dst == imm goto +offset	Jump instructions
0x17	any	dst -= imm	Arithmetic instructions
0x18	any	dst = imm	Load and store instructions

...

Open questions

Ok to include Linux & Clang specific implementation notes?
(current draft does, propose yes)

> ⓘ Note
>
> *Linux implementation*: In the Linux kernel, uint32_t is expressed as u32, uint64_t is expressed as u64, etc. This document uses the standard C terminology as the cross-platform specification.

Support for `BPF_ALU` is required in ISA version 3, and optional in earlier versions.

> ⓘ Note
>
> *Clang implementation*: For ISA versions prior to 3, Clang v7.0 and later can enable `BPF_ALU` support with `-Xclang -target-feature -Xclang +alu32`.

of program input arguments

Upon entering execution of an eBPF program, registers R1 - R5 initially can contain the input arguments for the program (similar to the argc/argv pair for a typical C program). The actual number of registers used, and their meaning, is defined by the program type; for example, a networking program might have an argument that includes network packet data and/or metadata.

> Note
>
> *Linux implementation*: In the Linux kernel, all program types only use R1 which contains the "context", which is typically a structure containing all the inputs needed.

These instructions are used to access packet data and can only be used when the program context contains a pointer to a networking packet. `BPF_ABS` accesses packet data at an absolute offset specified by the immediate data and `BPF_IND` access packet data at an offset that includes the value of a register in addition to the immediate data.

These instructions have seven implicit operands:

- Register R6 is an implicit input that must contain a pointer to a context structure with a packet data pointer.
- Register R0 is an implicit output which contains the data fetched from the packet.
- Registers R1-R5 are scratch registers that are clobbered by the instruction.

> ⓘ Note
>
> *Linux implementation*: In Linux, R6 references a struct sk_buff.

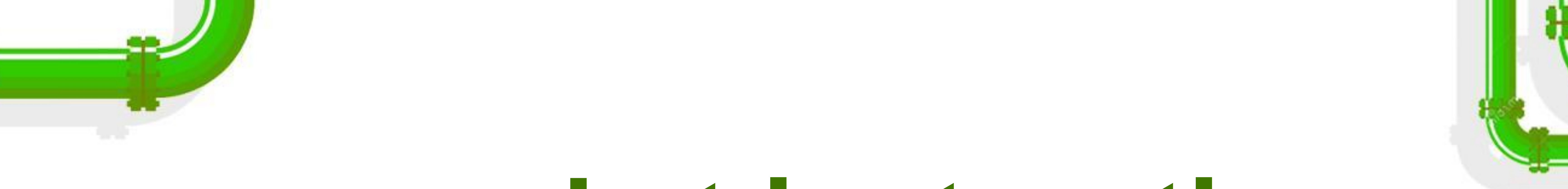
Should we just remove legacy packet instructions from the standard?

These instructions have an implicit program exit condition as well. If an eBPF program attempts access data beyond the packet boundary, the program execution must be gracefully aborted.

`BPF_ABS | BPF_W | BPF_LD` (0x20) means:

```
R0 = ntohl(*(uint32_t *) (R6->data + imm))
```

where `ntohl()` converts a 32-bit value from network byte order to host byte order.



Legacy packet instructions

These instructions are used to access packet data and can only be used when the program context contains a pointer to a networking packet. `BPF_ABS` accesses packet data at an absolute offset specified by the immediate data and `BPF_IND` access packet data at an offset that includes the value of a register in addition to the immediate data.

From: https://lore.kernel.org/bpf/8DA9E260-AE56-4B21-90BF-CF0049CFD04D@intel.com/

> > I think we need to document them as supported in the linux kernel, but deprecated in general.
> > The standard might say "implementation defined" meaning that different run-times don't have to support them.
>
> Yeah. If we do the extensions proposal above we could make these a specific extension as well.

y packet
rd?

```
R0 = ntohl(*(uint32_t *) (R6->data + imm))
```

where `ntohl()` converts a 32-bit value from network byte order to host byte order.

Other wide instructions

Quentin writes:

> The dest = imm (0x18) and call (0x85) instructions have a different semantic when their src register is set to a special flag. I think this is also part of the ISA and should be documented? See commits 2 to 7 of iovisor/bpf-docs#26 (and their description) for a quick reference.

The PR has:

0x18 (src == 0) | lddw dst, imm | dst = imm
0x18 (src == 1) | lddw dst, map | dst = imm with imm == map fd
0x18 (src == 2) | lddw dst, map value | dst = map[0] + insn[1].imm with insn[0] == map fd
0x18 (src == 3) | lddw dst, kernel var | dst = imm with imm == BTF id of var
0x18 (src == 4) | lddw dst, BPF func | dst = imm with imm == insn offset of BPF callback
0x18 (src == 5) | lddw dst, imm | dst = imm with imm == map index
0x18 (src == 6) | lddw dst, map value | dst = map[0] + insn[1].imm with insn[0] == map index

But the ISA does not currently define the existence or meaning of a "map fd" or a "BTF id of var" or a "map index" or a "BPF callback". Should it?

Linux
Plumbers
Conference
Dublin, Ireland September 12-14, 2022