Programmable Dataplane

THE NEXT STEP IN SDN ?

SIMON JOUET – SIMON.JOUET@GLASGOW.AC.UK

HTTP://NETLAB.DCS.GLA.AC.UK

Motivation (I)

In only a few years OpenFlow revolutionised Networking

- Decouple the **control plane** from the **data plane**
- Centrally manage the control plane in software
- Open the control logic to the users
  - *Just program you network behaviour in Java/Python …*
- Abstract packet forwarding logic from particular hardware
  - *No more vendor lock-in, same software can be used on any OpenFlow switch*
- Access to network wide information
  - *Topology: links, switches, ports, bandwidth, latency …*
  - *Globally informed (possibly optimal) decision can be made*

~16 900 publications in less than 8 years

- *Traffic Engineering, Routing Protocols, Policy enforcement, Software Design, Performance evaluation, Architecture verification, Debugging …*

Motivation (II)

OpenFlow just the first step in SDN

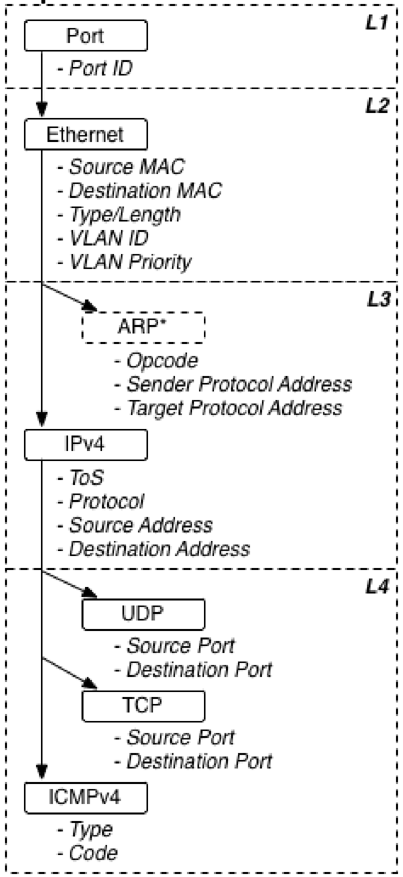
- OF was necessary to show the benefits SDN can provide

- However, limited functionality and purpose
  - Limited set of fields to match on (3.6 times more fields in 1.5 than 1.0)
  - What about new protocols? And custom protocols?
  - What about inequality or range matching?
  - What about statistics other than Packet, Byte count, Flow duration?
  - What about stateful matching or forwarding logic?
  - What about line-rate packet processing? Telemetry? Anomaly detection?

To achieve the next step in SDN we need data plane programmability

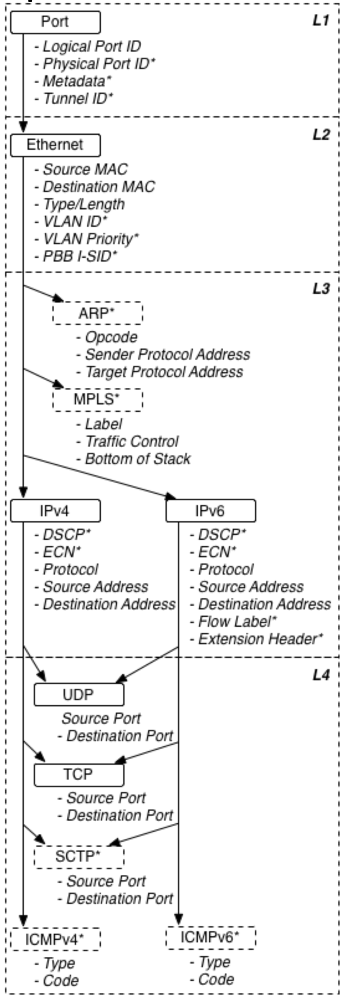
Motivation (II)

OpenFlow 1.0

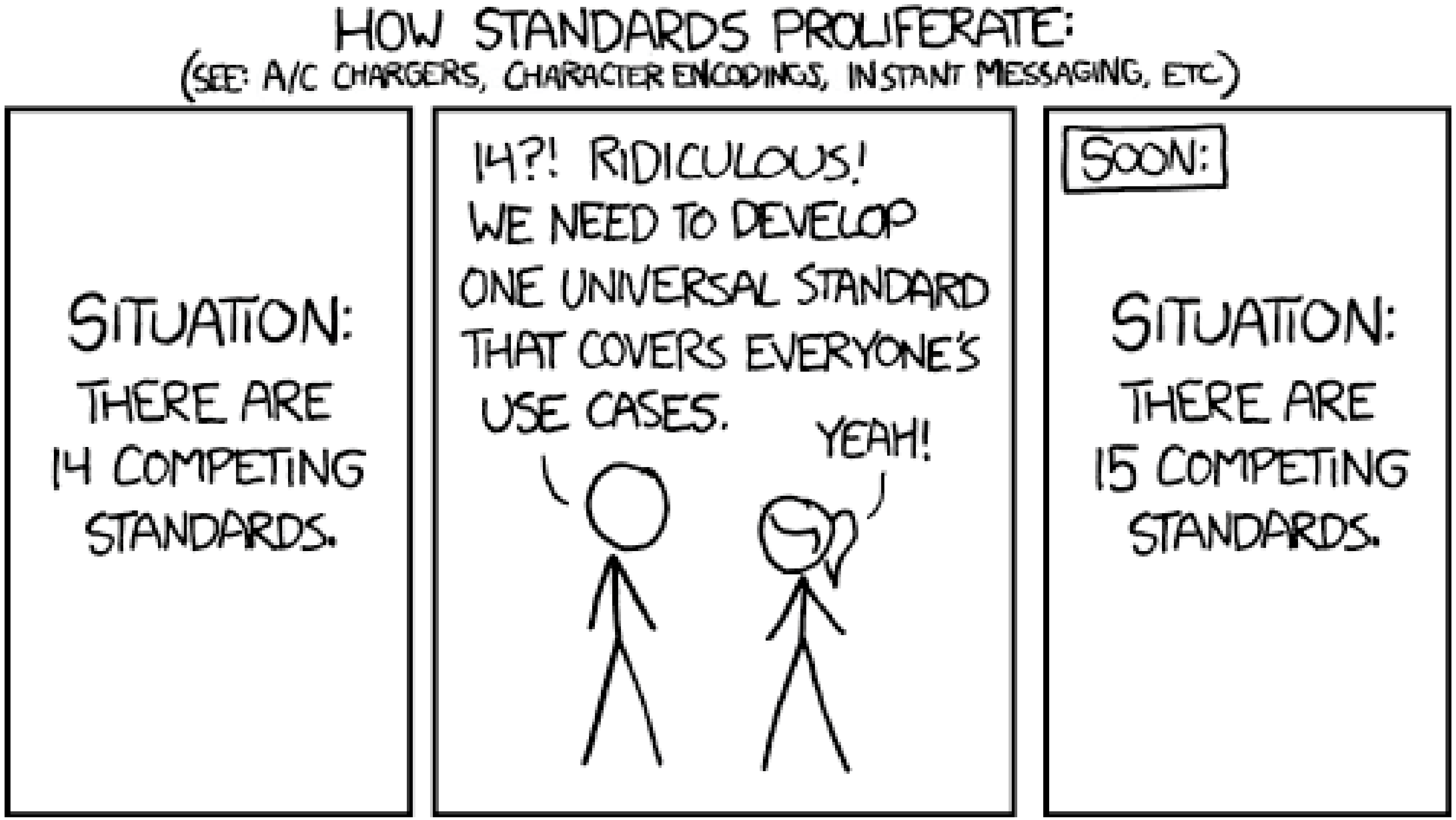


OF Version	Release Date	Match fields	Depth	Size
1.0	Dec 2009	12	12	264
1.1	Feb 2011	15	15	320
1.2	Dec 2011	36	9 — 18	603
1.3	Jun 2012	40	9 — 22	701
1.4	Oct 2013	41	9 — 23	709
1.5	Dec 2014	44	10 — 26	773

OpenFlow 1.5



Challenges



https://xkcd.com/927/

Challenges

Goal:

- Program the data plane to achieve arbitrary matching and processing
  - *Not limited to the fields, action, and processing OpenFlow provide*
  - *Do not rely on yearly protocol specification update for new features*
- Protocol Independence
  - *No knowledge of Ethernet, TCP, UDP ...*
  - *Work with existing and future protocols*
- Target Independence
  - *No specific target hardware*
- Line-rate processing

Do not try to support every existing protocol header fields:

- Provide an instruction set suitable to match arbitrary protocols and fields
- Execution of the instruction set is an implementation detail
  - Interpreter, Just-in-time compiler, FPGAs, ASICS, NPU …

The BPF instruction set (I)

No point reinventing the wheel, 1992 McCane and Jacobson BPF

- Designed specifically for packet matching and processing
- Designed as a platform (target) independent bytecode
- Designed as a protocol independent instruction set
- Widely used by the Linux kernel
- Widely used by networking tools: TCPdump, Wireshark, libpcap, winpcap …
- Extended BPF (eBPF) + JIT added in Linux kernel 3.18

```
simonj@haggis:~$ sudo tcpdump -d 'ether proto 0x800 and tcp src port not 22'
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 10
(002) ldb      [23]
(003) jeq      #0x6            jt 4    jf 9
(004) ldh      [20]
(005) jset     #0x1fff         jt 9    jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 14]
(008) jeq      #0x16           jt 10   jf 9
(009) ret      #65535
(010) ret      #0
```

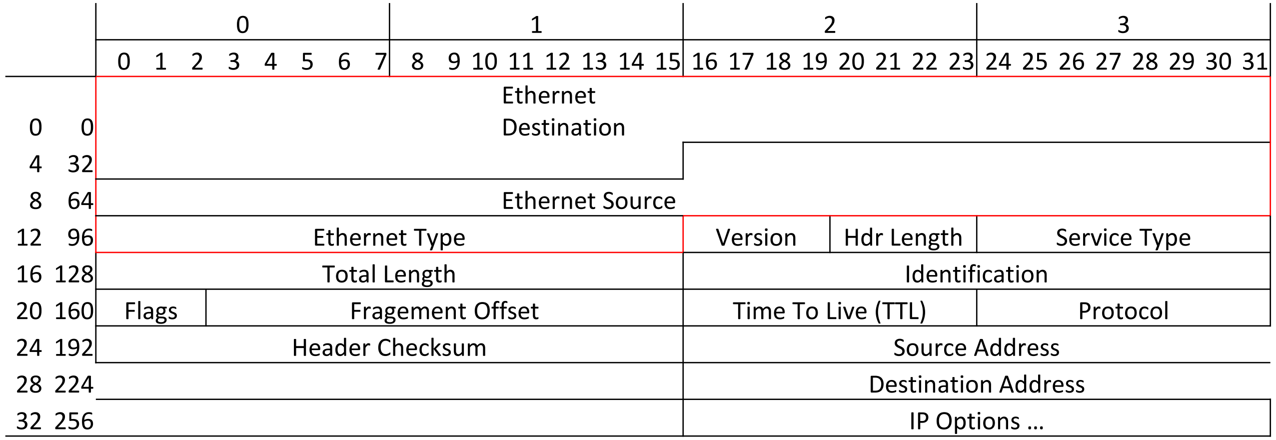
Match

- IPv4 packets
- Not port 22

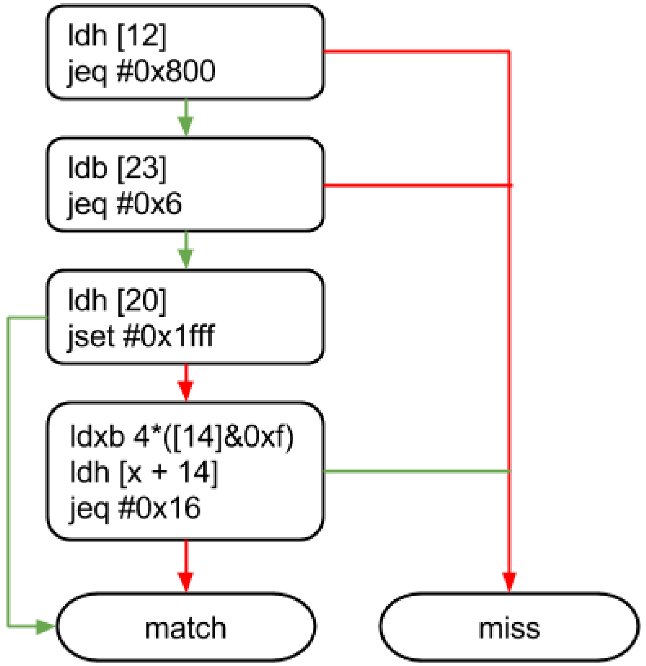
The BPF instruction set (II)

Represent the BPF code as a tree:

- → jt (jump if condition is true)
- → jf (jump if condition is false)



Ethernet + IPv4 Headers



Acyclic Flow Graph Representation of a BPF program

The BPF instruction set (III)

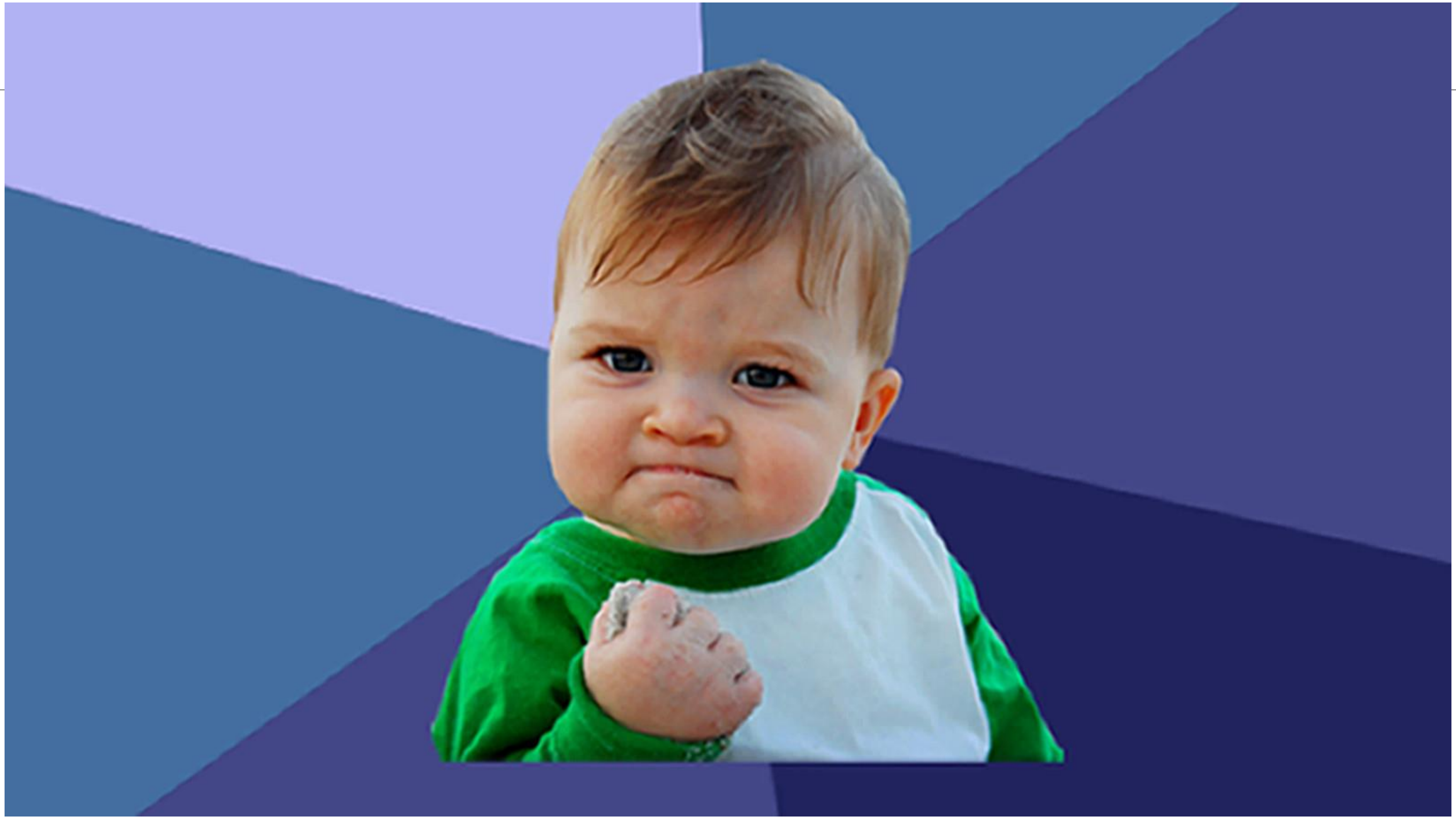
One of requirement is line-rate processing

- BPF does not include backward jumps, the execution can only move forward
- The Control Flow Graph (CFG) is therefore acyclic (not Turing Complete)

Results in nice properties for High Performance Packet Processing

- CFG can be reordered to only parse each layer once
  - *Reduce the number of memory accesses, speed up the execution*
- Nodes can be reordered to be executed in the order of the layers
  - Pass-through switching: execute the BPF program as the packet is received
- Worst case execution time can be calculated
  - Maximum program execution time is the time it take to execute longest path in the graph
- Can be mapped to a **match+action** pipeline

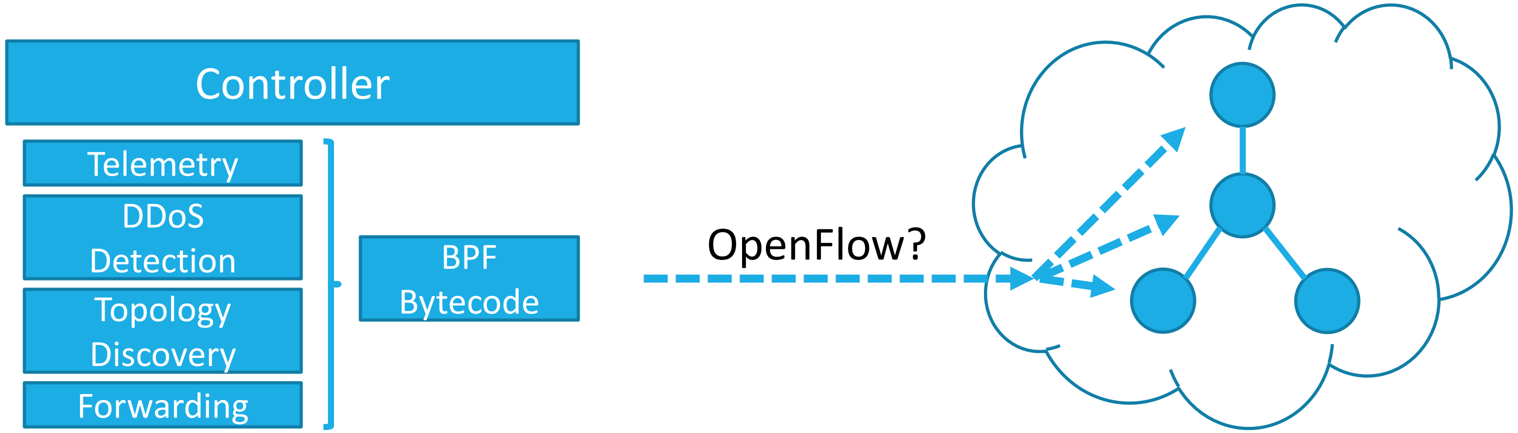
Achieve Platform/Protocol Independence and provide bound for realtime execution



Architecture

Intelligent vs Complex

- Example of “intelligence” is a learning switch
- Complex processing doesn’t imply “intelligence”
- The central controller provide the intelligence the nodes provide the processing
  - *If you don’t know you ask the controller*



Implementation

Proof of concept software-switch implementation

- Less than 500 lines of Go code
- Simple packet forwarding between NICs
  - Use BPF return code as the **output port**
- Complete BPF bytecode interpreter
  - 50 instructions, 2 registers, scratch memory
- We should release the code “soon”

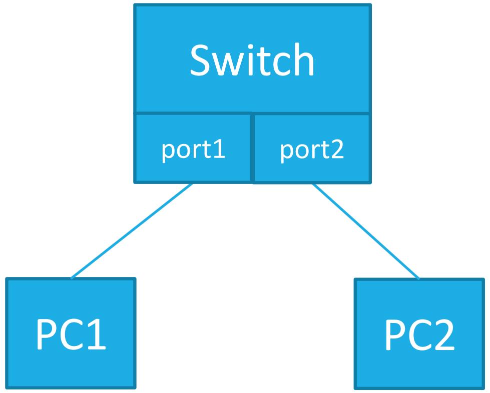
Working on a NetFPGA 10G implementation

- Show that line-rate (10G) can be achieved
- Evaluate the hardware complexity
  - Number of FPGA slices and macro cells

Example 1 - Forwarding

A really stupid switch

- If input_port is 1 send packet to port 2
- Else send packet to port 1

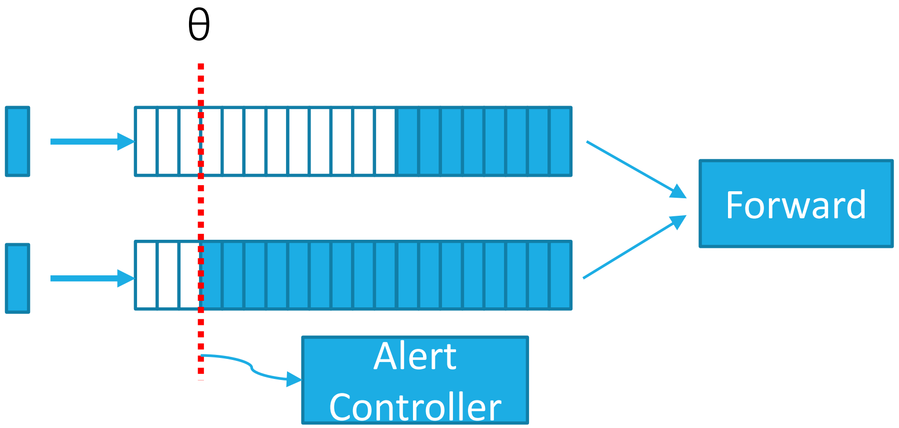



```
(01) {bpf.BPF_LD | bpf.BPF_ABS | bpf.BPF_W, 0, 0, 0},  // load the in_port
(02) {bpf.BPF_JMP | bpf.BPF_JEQ | bpf.BPF_K, 0, 1, 1}, // if in_port != 1 goto (04)
(03) {bpf.BPF_RET, 0, 0, 2},                           // output to port 2
(04) {bpf.BPF_RET, 0, 0, 1},                           // output to port 1
```

Example 2 - Telemetry

Alert controller on high buffer occupancy

- Check the buffer occupancy (θ)
- Alert controller if buffer occupancy > 100




```
{ bpf.BPF_LD | bpf.BPF_MEM, 0, 0, 0x20 }              // Load current buffer occupancy
{ bpf.BPF_JMP | bpf.BPF_JGT | bpf.BPF_K, 0, 1, 100} // if accumulator > 100
{ bpf.BPF_RET, 0, 0, 0xffff }                         // Alert the controller

// Jump here if buffer occupancy < 100
```

Example 3 – Anomaly Detection

SYN/FIN Denial of Service Anomaly Detection (21 instructions)

- Keep track of the number of packet with TCP SYN or FIN flag set
- If #SYN > 3*#FIN, alert the controller

```
// Check if it's an IP packet
{bpf.BPF_LD | bpf.BPF_ABS | bpf.BPF_H, 0, 0, 16},      // Load the ether.type
{bpf.BPF_JMP | bpf.BPF_JEQ | bpf.BPF_K, 0, 20, 0x800}, // Check if IPv4

// Check if it's a TCP packet
{bpf.BPF_LD | bpf.BPF_ABS | bpf.BPF_B, 0, 0, 27},      // Load the ip.protocol
{bpf.BPF_JMP | bpf.BPF_JEQ | bpf.BPF_K, 0, 18, 0x06},  // Check if TCP

// Checks that the IP fragment offset is 0 so we are sure that we have a TCP header
{bpf.BPF_LD | bpf.BPF_ABS | bpf.BPF_H, 0, 0, 24},        // Load the ip.offset
{bpf.BPF_JMP | bpf.BPF_JSET | bpf.BPF_K, 16, 0, 0x1fff}, // If ip.offset is not 0 return

// Get the length of the IP header into the index register
{bpf.BPF_LDX | bpf.BPF_B | bpf.BPF_MSH, 0, 0, 18},       // ip.header_length, multiply it by 4

// Check the state of the TCP flags
{bpf.BPF_LD | bpf.BPF_IND | bpf.BPF_B, 0, 0, 4 + 14 + 13}, // Load tcp.flags
{bpf.BPF_JMP | bpf.BPF_JSET | bpf.BPF_K, 1, 0, 0x02},      // Check tcp.flags.SYN
{bpf.BPF_JMP | bpf.BPF_JSET | bpf.BPF_K, 4, 12, 0x01},     // Check if tcp.flags.FIN is set

// If SYN is set, increment the counter, mem[0]++
{bpf.BPF_LD | bpf.BPF_MEM, 0, 0, 0},                    // Load memory 0 (SYN c
{bpf.BPF_ALU | bpf.BPF_ADD | bpf.BPF_K, 0, 0, 1}, // Increment the accumu
{bpf.BPF_ST | bpf.BPF_MEM | bpf.BPF_W, 0, 0, 0},  // Store the value from
{bpf.BPF_JMP | bpf.BPF_JA, 3, 0, 0},                    // Go check if somethin

// if FIN is set, increment the counter, mem[1]++
{bpf.BPF_LD | bpf.BPF_MEM, 0, 0, 1},                    // Load memory 1 (FIN c
{bpf.BPF_ALU | bpf.BPF_ADD | bpf.BPF_K, 0, 0, 1}, // Increment the accumu
{bpf.BPF_ST | bpf.BPF_MEM | bpf.BPF_W, 0, 0, 1},  // Store the value from

// if SYN count is more than 3 times greater than FIN count something is
{bpf.BPF_LD | bpf.BPF_MEM, 0, 0, 0},                    // Load SYN count in ac
{bpf.BPF_ALU | bpf.BPF_DIV | bpf.BPF_K, 0, 0, 3}, // divide SYN count by
{bpf.BPF_LDX | bpf.BPF_MEM | bpf.BPF_W, 0, 0, 1}, // Load FIN count in in
{bpf.BPF_JMP | bpf.BPF_JGT | bpf.BPF_X, 0, 1, 0}, // if #SYN/3 > #FIN

// Alert the controller
{bpf.BPF_RET, 0, 0, 0xffff},
```

GEANT Testbed Service

Could *a* next step be Data Plane programmability?

- Allow large scale forwarding, telemetry, anomaly detection experiments
  - Can this provide some solution to the outstanding GTS problems?
  - Use this approach for debugging, insert a BPF "probe"?
- How should that work in a multi-tenant network?
  - Need to make sure no interference between tenants
  - Isolation is harder when you can do whatever you want with the cables …
- What deployment steps could we envision?
  1. Have a BPF Switch Resource Type to create a virtual network between nodes
  2. Add NetFPGAs to the set of Resource Types of GTS
     - When defining a GTS testbed topology provide the FPGA bitfile?
     - Would allow large scale experiment on data plane processing
     - Though are NetFPGA suitable for this ? (what about a bad flash?)

Future Work

A language to describe the functions

- Writing the “SYN/FIN” ratio module took couple of hours …
- P4 is a perfect candidate (Sigcomm 2014)
- Currently working on a P4 to BPF compiler
  - *Was hoping to get it working before the workshop … Lexer and Parser are done*

Controller to Switch communication

- How do you send the BPF code to the switch?
- What do you send, the full program, just a diff of the update?
- Trade-off between switch complexity and data transferred

How to expose metadata

- Telemetry require buffer occupancy, current CPU load, memory utilisation …
- Most sampling processes require an accurate timestamp
- Go for the microcontroller approach, memory map the metadata?

Questions?

SIMON.JOUET@GLASGOW.AC.UK