Goodbye Scapy, Hello ***snappi***

snappi

Dataplane testing from *DC to Daylight* Using the Open Traffic Generator API

Chris Sommers, SW Architect
Ankur Sheth, Director of Engineering
Keysight Technologies

Speaker Biographies

Chris Sommers is a software architect with decades of experience in the design of hardware and software for wireless and wireline networking. Chris works at Keysight Networking Labs where he investigates emerging technologies. Chris also participates in several P4.org working groups. He holds a Bachelor's degree in Electronic Engineering from Cal Poly San Luis Obispo, CA.

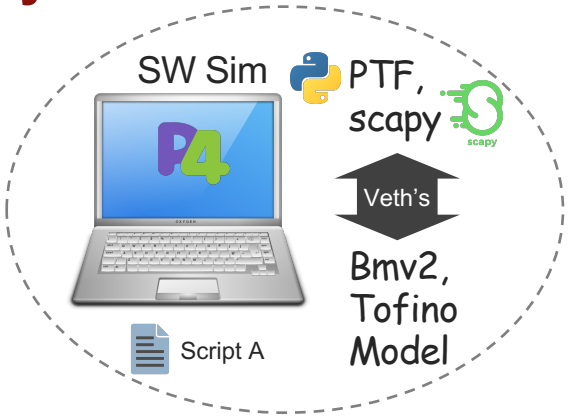
Ankur Sheth is a member of Product Strategy Group at Keysight where he works on NPIs and also leads Keysight's Network Test Virtualization products. Over the years Ankur has led numerous teams that have built high quality breakthrough products in wired and wireless networking space. Ankur holds a Master's degree in Electrical Engineering (Computer Networks) from University of Southern California, CA.

ixia

Chris and Ankur have helped develop the well-known Ixia line of packet testers manufactured by Keysight Technologies.

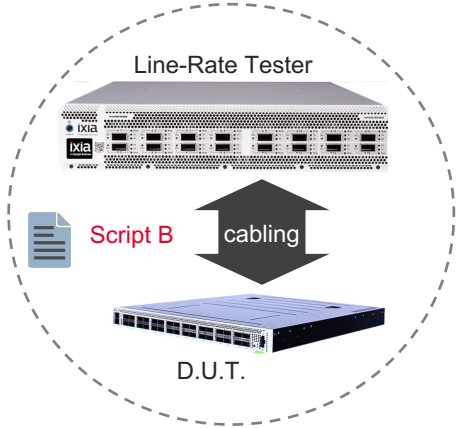
I'm a typical P4 Dataplane Developer – What's my workflow ?

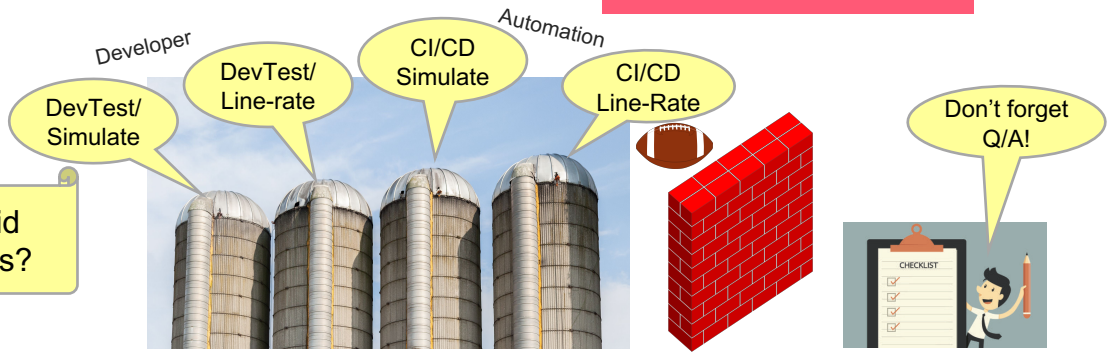
- Write some really cool P4 code
- Run it in bmv2 or a vendor P4 simulator (e.g. Tofino Model)
- Throw packets at it with Scapy/PTF, Trex, custom SW tools, etc.



- Test it on *real* P4 switching device (Device Under Test, or DUT)
- *Use a hardware-based* traffic generator for precision & performance
- Problem – can't re-use your scripts! Scapy/PTF doesn't natively support true line-rate testers
  - Write custom libraries/wrappers for PTF?
  - Rewrite scripts from scratch?

- Lots of extra work!
- Maintenance headache!
- May not work the same!
- Vendor lock-in!





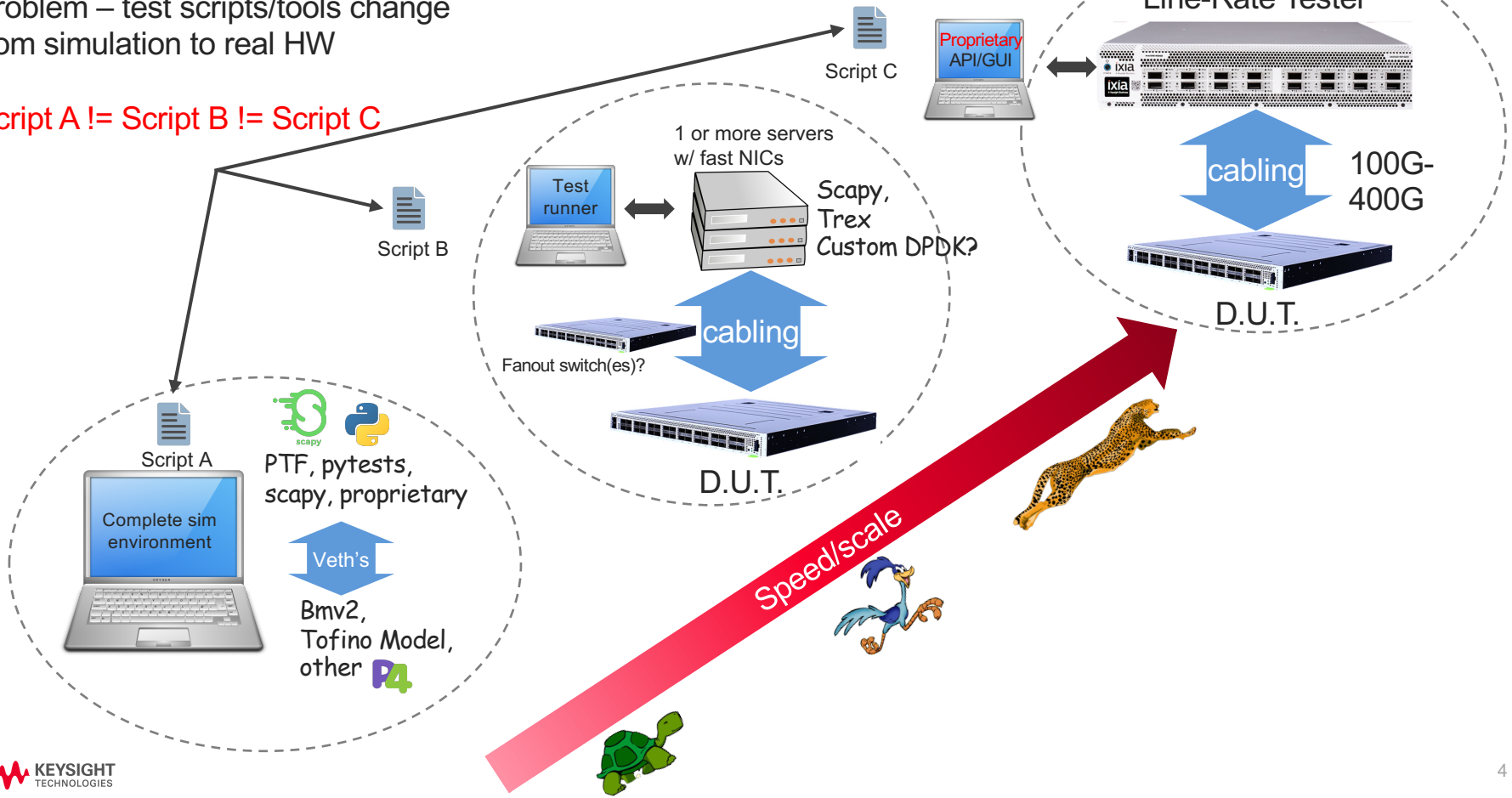
How to avoid scripting silos?

KEYSIGHT TECHNOLOGIES

Traditional Test Environments vs. Speed/Scale

Problem – test scripts/tools change from simulation to real HW

Script A != Script B != Script C



Wish-List for Packet Testing

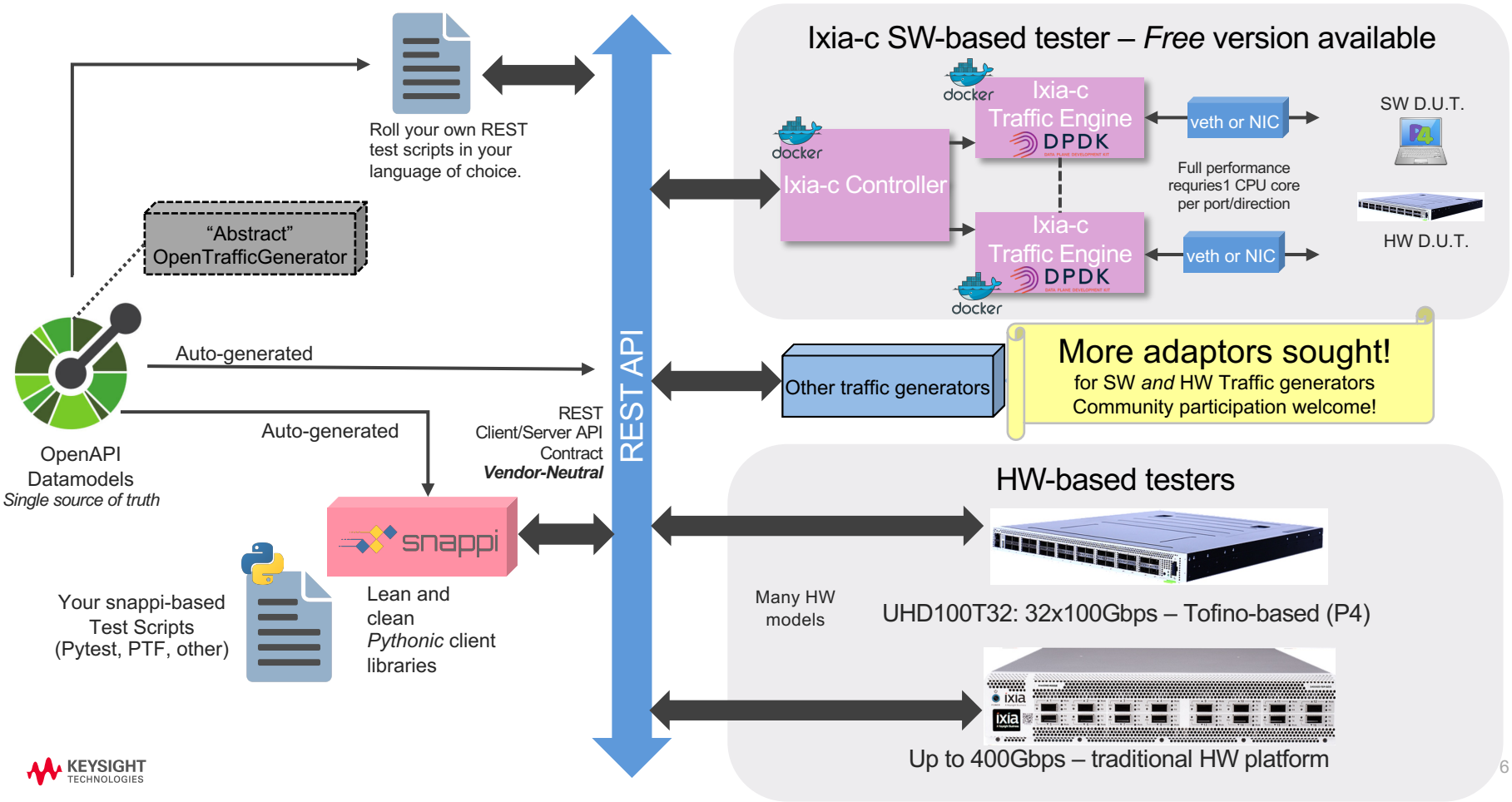
As a developer, I want it all, and I want it now:

- Write my test script once, run it everywhere
- Speeds from slow simulators (“DC”) up to line rate (“Daylight”)
- Model-based, declarative API – config as a “document”
- Open standard – vendor-agnostic, community ecosystem
- Free, software-based solutions
- Commercial, full-performance solutions
- Easy onramp – download and go in minutes

Introducing…Open Traffic Generator, snappi & Ixia-c

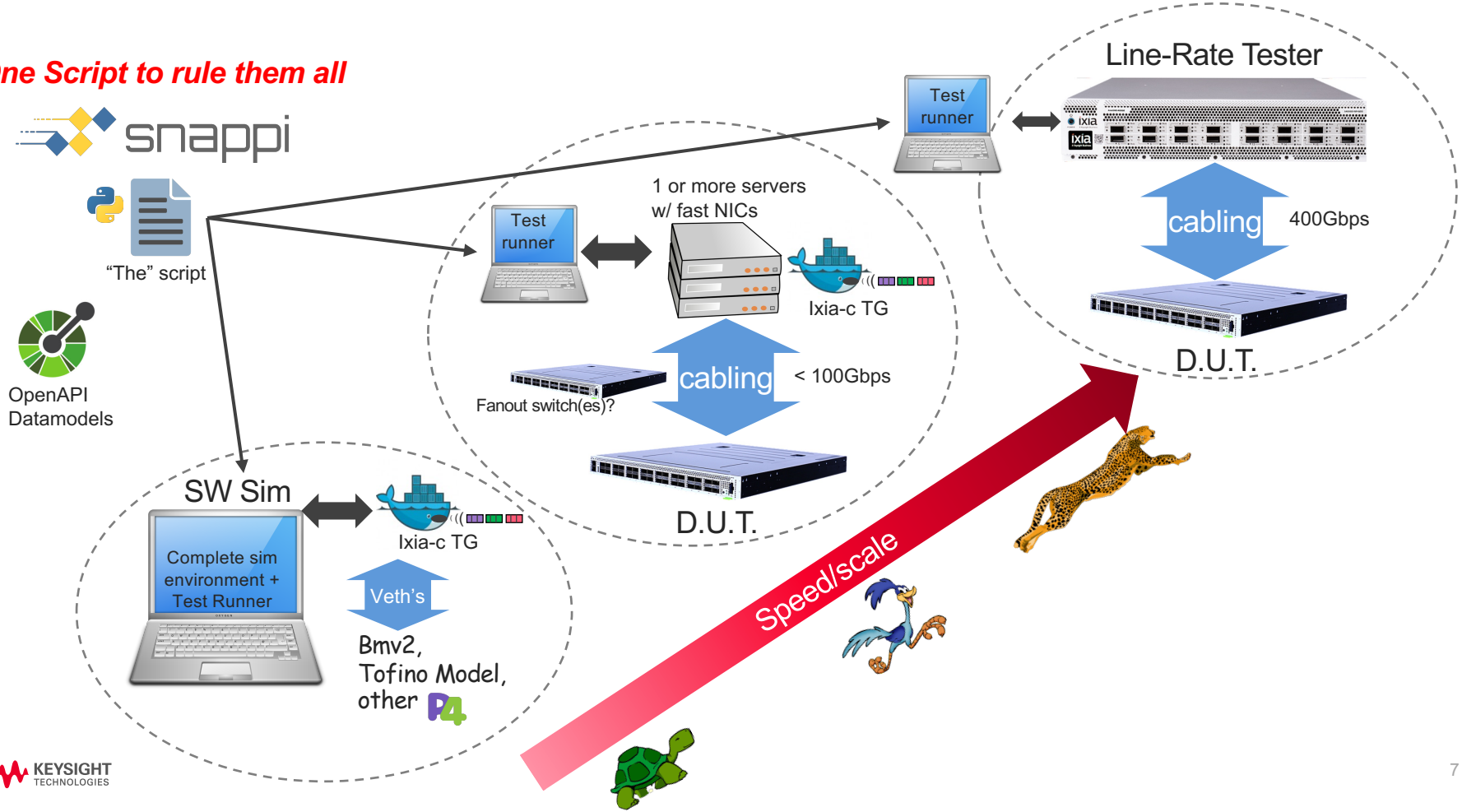
https://github.com/open-traffic-generator

Open Traffic Generator – Model-based Packet Testing!



Wish-list answered! snappi + OpenTrafficGenerator

One Script to rule them all

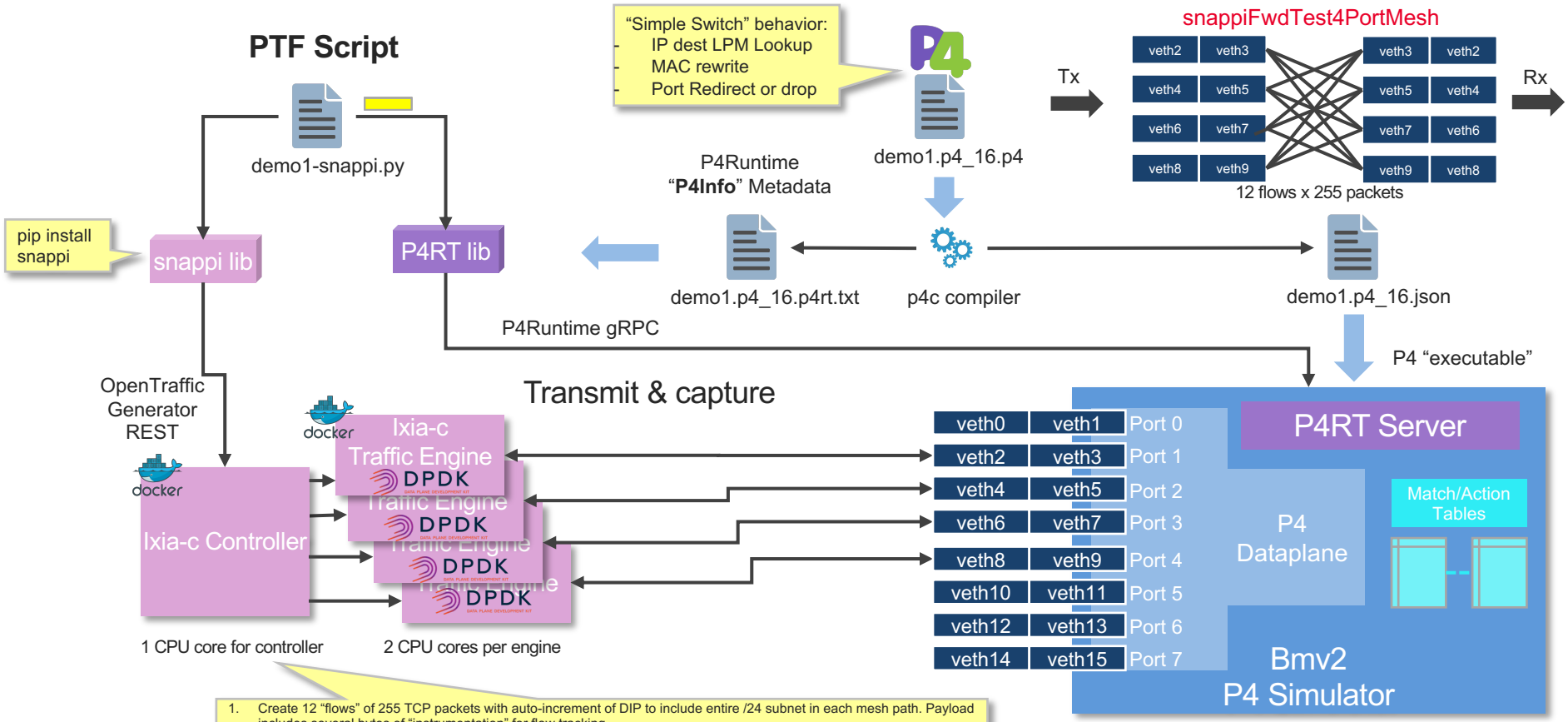


How to demonstrate Snappi?

- As an exercise, I converted an existing PTF test "**demo1**" from Andy Fingerhut's well-known p4-guide repo: https://github.com/jafingerhut/p4-guide
- I forked the repo: https://github.com/chrispsommers/p4-guide/tree/snappi-tests2 and made a new test directory demo1-snappi
- I modified one of the existing tests ***FwdTest*** and replaced Scapy with snappi
- I added several other tests to illustrate other snappi techniques and features
- Today's demo will showcase the added snappiFwdTest4PortMesh test which does some cool things:
  - Sets up 4 Ixia-c traffic engines + one Ixia-c controller (5 Docker containers)
  - Configures 12 flows to establish a 4x4 port mesh.
    - Each flow includes auto-increment in the IP dest address to cover a /24 subnet
    - Send 100-byte packets at a precise 50 PPS using the built-in Ixia-c scheduler
  - Retrieve all per-port and per-flow Tx and Rx statistics and compare to expected values
  - Retrieve all captured packets, examine the IP addresses and confirm the complete port-mesh was received.

Note this same test could easily be enhanced to control line-rate testers, at full speed and scale, by changing a few parameters

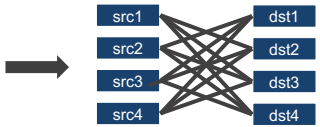
Demo Setup: PTF Test using bmv2, snappi+Ixia-c Tgen



1. Create 12 "flows" of 255 TCP packets with auto-increment of DIP to include entire /24 subnet in each mesh path. Payload includes several bytes of "instrumentation" for flow tracking.
2. Send and capture forwarded packets and count the stats per-port and per-flow
3. Verify per-port and per-flow packet counts, also verify the entire IP address mesh was received.

snappi code snippet – creating 12 flows

```
i = 0
for src in self.port_ndxs:
    for dst in self.port_ndxs:
        if src == dst:
            continue # no hairpin switching
```




```
        print("Configuring flow[%d]: %s => %s" % (i, ports[src].name, ports[dst].name))
        flow = self.cfg.flows.flow(name='port%d-%d' %(src+1, dst+1))[-1]
        # flow endpoints
        flow.tx_rx.port.tx_name = ports[src].name
        flow.tx_rx.port.rx_name = ports[dst].name
        # configure rate, size, frame count
        flow.size.fixed = 100
        flow.rate.pps = 50
        flow.duration.fixed_packets.packets = self.tx_count
        # configure protocol headers with defaults fields
        flow.packet.ethernet().ipv4().tcp()

        eth = flow.packet[0]
        eth.src.value = host_macs[src]
        eth.dst.value = host_macs[dst]

        ipv4 = flow.packet[1]
        ipv4.dst.increment.start = ip_hosts[dst]
        ipv4.dst.increment.step = '0.0.0.1'
        ipv4.dst.increment.count = self.tx_count
        ipv4.src.value = ip_hosts[src]
        ipv4.time_to_live.value = 64

        tcp = flow.packet[2]
        tcp.src_port.value = 1234
        tcp.dst_port.value = 80

        i+=1
```

Instantiate a flow

Precision scheduler: PPS, BPS or % line rate

Scapy Equivalent: Ether()/TCP()/IP()

Auto-increment a packet header– built-in to Snappi!

Test Outline - snappiFwdTest4PortMesh

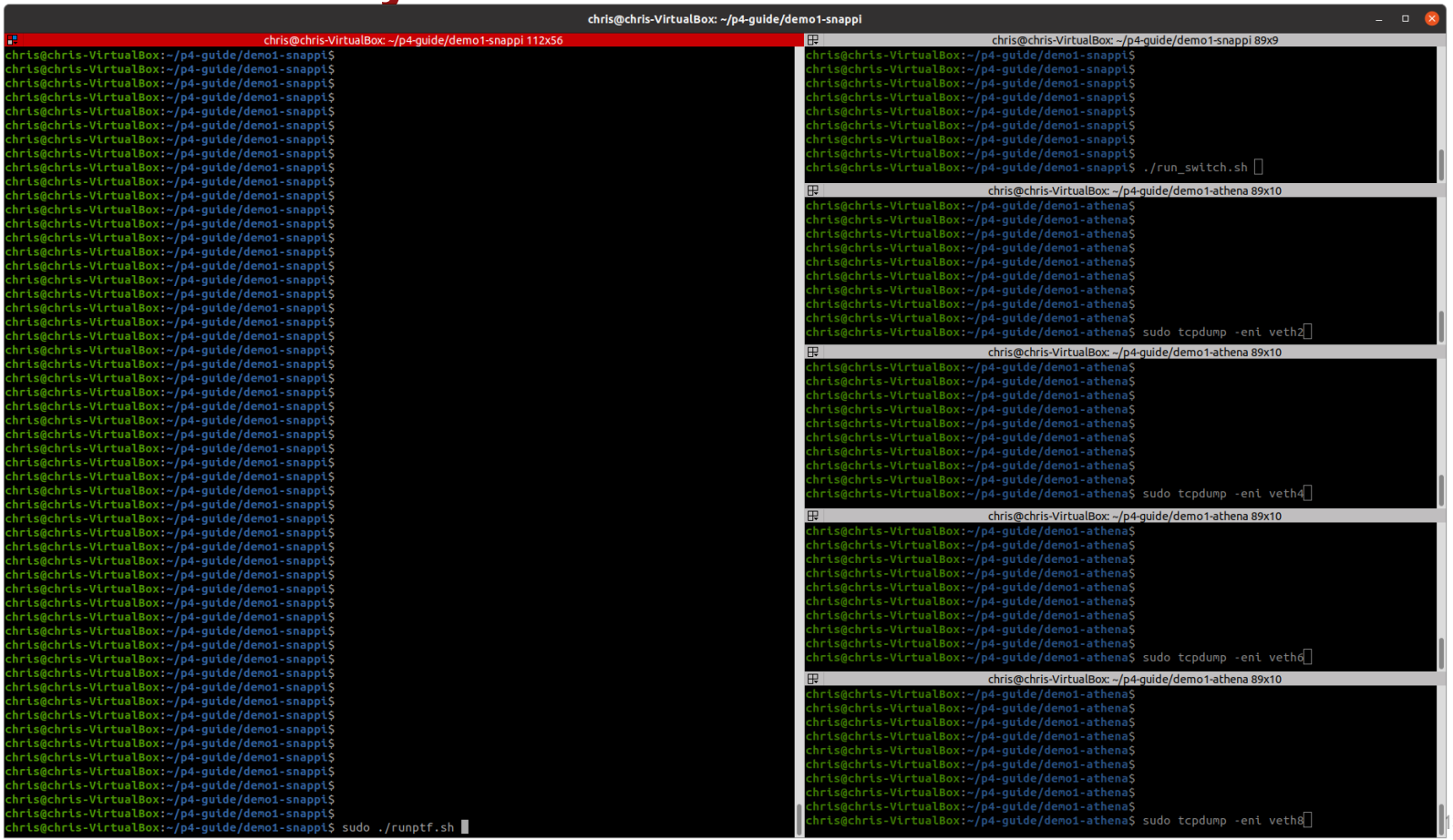
- Configure Ixia-c for 12 traffic flows, into veth2, veth4, veth6 and veth8 (dataplane ports 1-4 respectively in the P4 code). The 12 flows comprise a full-mesh, full-duplex test of port forwarding. Each flow will send 256 packets into its port, incrementing the last byte of the DIP from 1 to 256. Each flow will emit packets at 50 packets per second. We wait until the pipeline finishes processing all packets (or timeout while waiting).
- Configure Ixia-c to capture all the return traffic
- Start the traffic flows and capture the results
- Verify no packets were captured because the P4 dataplane forwarding tables have not been programmed: the default action is *drop*.
- Configure the P4 tables to match on the DIPs as configured in the traffic flows and forward to the correct egress ports, also performing MAC rewrite.
- Start traffic flow a second time and capture everything. We wait until the received packet counts match the expected values on all flows (or timeout waiting).
- Verify several expectations:
  - Each port transmits the correct count of packets (3*255, i.e. 255 packets to each of the other ports in the mesh)
  - Port Tx stats = port Rx stats = 3*255
  - The captured results has the correct number of packets for each flow (same as transmitted to each flow)
  - Examine each captured packet, extract IP src and dest address, and confirm exactly one packet was sent between each "host" and each of 255 "destinations" on the other port's subnets.

Demo available at https://github.com/chrispsommers/p4-guide/tree/snappi-tests2

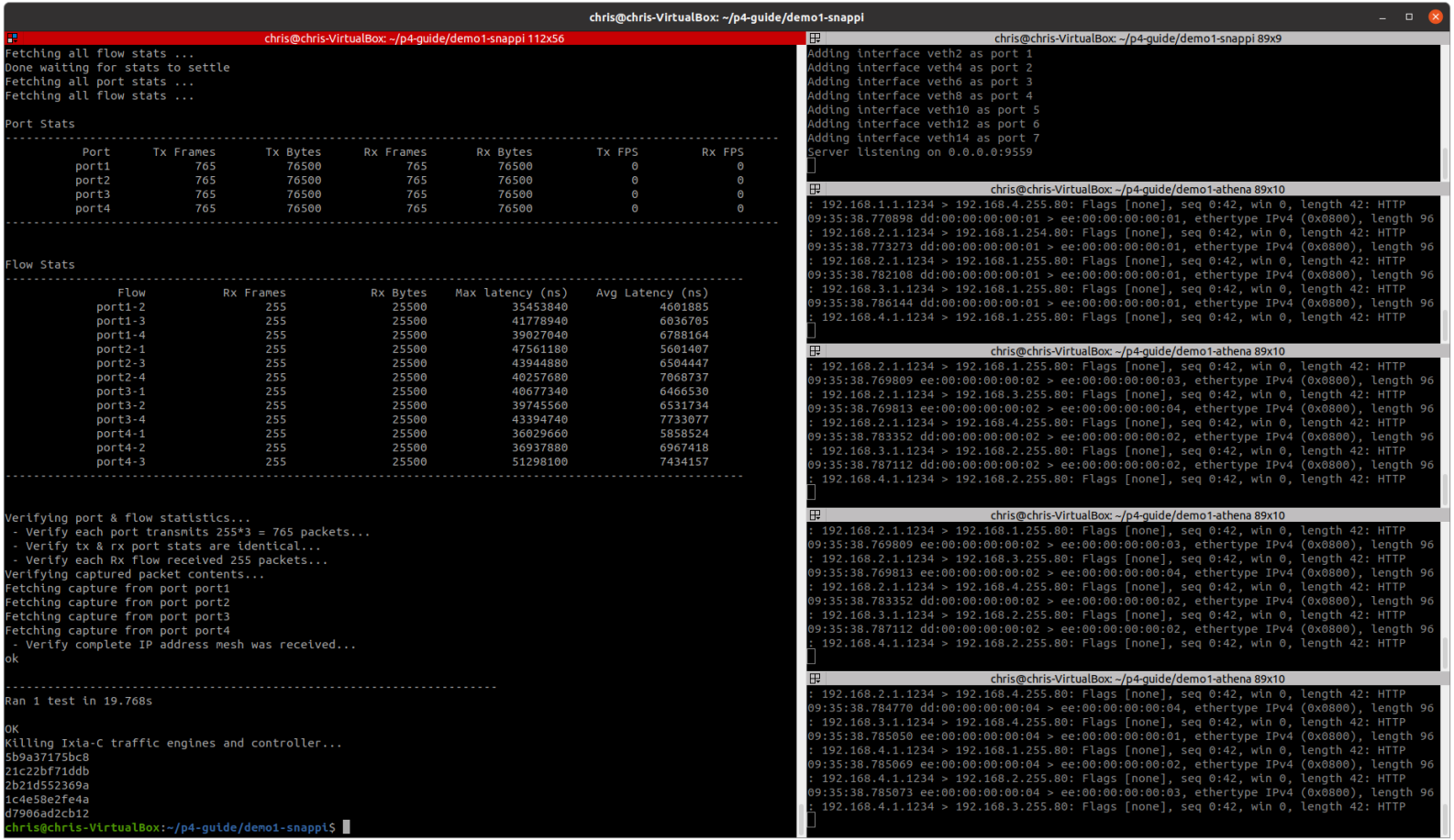
Thanks to Andy Fingerhut for his p4-guide repo at https://github.com/jafingerhut/p4-guide

Run the Demo!

Demo1 – Screen Layout and Commands



Demo1 – snappiFwdTest4PortMesh Results

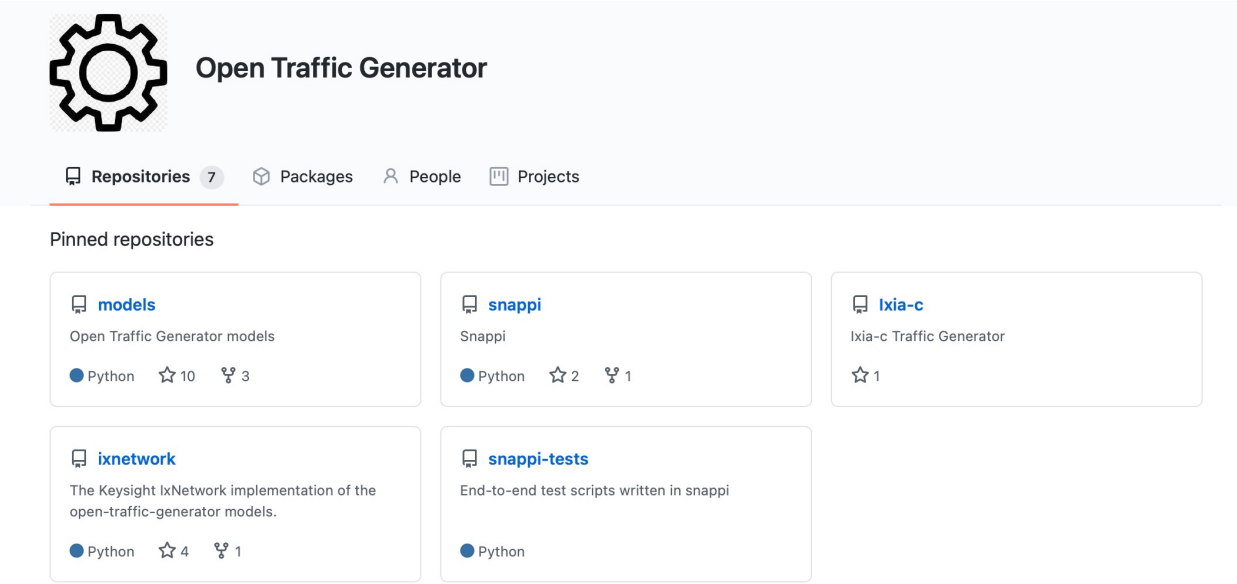


KEYSIGHT TECHNOLOGIES

snappi vs. Scapy

Item	snappi	Scapy
Portability	• **One script** to cover SW and HW traffic generators	• Only works for SW generator. Migration to HW = start over
Scalability (# ports)	• Add CPU cores (SW) or line cards/chassis (HW) to scale-out to any size • Controller and Traffic Engines/Line cards talk over sockets, easy to scale-out	• Limited by #NICs; can use fanout switches with additional ad-hoc automation – cumbersome!
Automation	• snappi and/or REST automation, designed for automation	• Pytest or PTF; framework and dataplane are synonymous
Performance/ Throughput	• HW Traffic-generators: 400Gbps port speeds • Ixia-c SW traffic generator: 10Gbps @ 64 byte frame size using one Xeon class core, approaching 100Gbps depending upon packet length; uses DPDK	• Limited by CPU speed – “slow-ish” • Not optimized for DPDK
CPU Resources (SW Traffic Gen)	• 1 core (shared) for controller • Hi-performance: 1 core per direction, per port • Best-Effort: 1 core per direction (shared among ports)	• Unclear, but scapy believed to run on one core. • Roll your own multi-core version for higher performance
Scheduler	• Precision scheduling of multiple flows, nsec precision in HW • Specified as packets-per-second, bits-per-second, % of max line rate • Burst Mode	• Roll-your-own scheduler – good luck!
Packet types	• Ethernet, VLAN, GRE, GTPv1, GTPv2, IPv4, IPv6, ICMP, ICMPv6, UDP, TCP, custom. More protocols are on the way.	• Essentially unlimited, easy to design new ones
Header patterns	• Built-in patterns like increment, decrement, list, etc. directly executed by the engines at speed, to generate millions of unique packets.	• Hand-build your own patterns in python, “manually” send them in sequence.
Flow tracking	• Built-in “instrumentation” header tracks up to 256 unique flows per port	• Not supported
Statistics	• Per-port and per-flow transmit and receive statistics • Latency measurements (min, max, avg) per flow	• Roll your own!
Capture	• Capture packets with filters, access via port or flow • Process in memory, write to Pcap file, send to tcpdump	• Capture packets per port
Statefulness/ Interactivity	• Not supported	• Good support, easy to perform interactive sessions

github.com/open-traffic-generator



Find us on GitHub!

- Use Ixia-c & snappi
- Report bugs
- Log feature requests
- Contribute to models
- Contribute other backends

Thank You

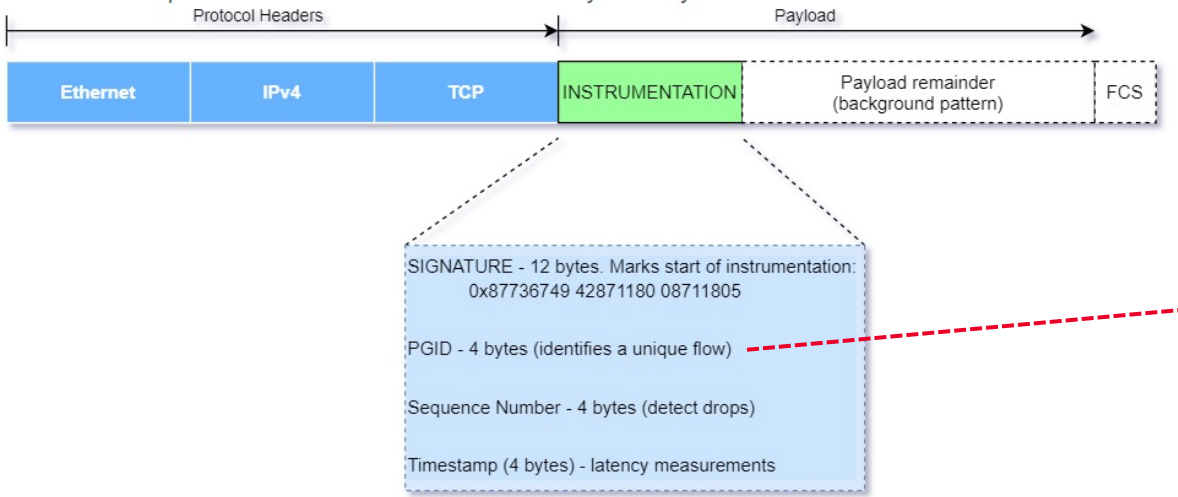
chris.sommers@keysight.com
ankur.sheth@keysight.com
https://github.com/open-traffic-generator

BACKUP SLIDES

Flow-Tracking, Instrumentation Explained

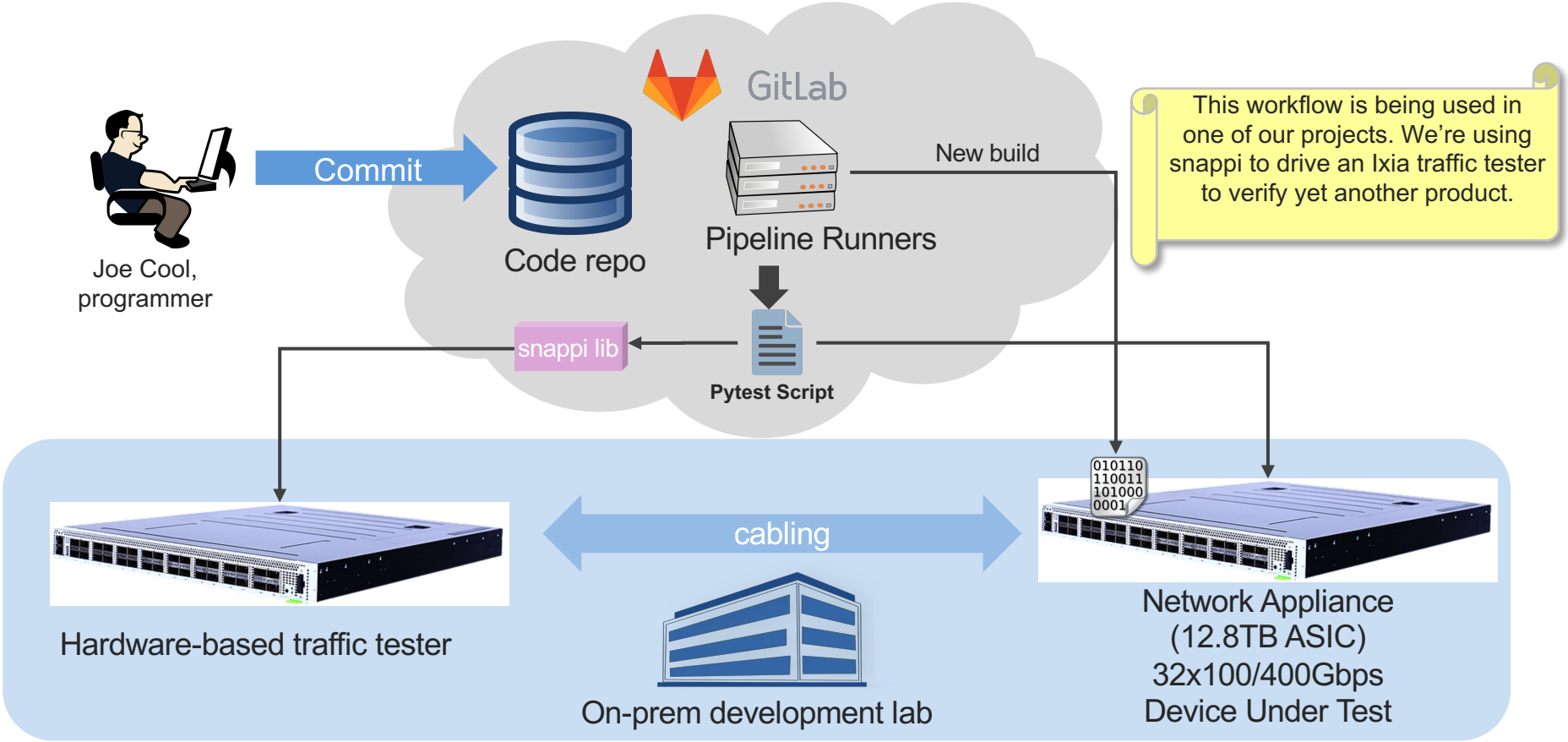
Ixia packet testers utilize a proprietary flow-tracking technique which involves inserting a special "instrumentation header" into the packet. It gets inserted after the last valid protocol header, i.e. it forms the first portion of "payload." This header, which is decribed in the ptf/scapy_contrib/ixia_scapy.py Scapy file, contains several interesting fields:

- 12-byte fixed "signature" which serves as a marker to indicate start of header
- 4-byte `PGID` or "port group ID" field; think of this as a flow ID
- 32-bit sequence number which can be used to detect packet drops
- 32-bit timestamp which can be used to measure latency or delay

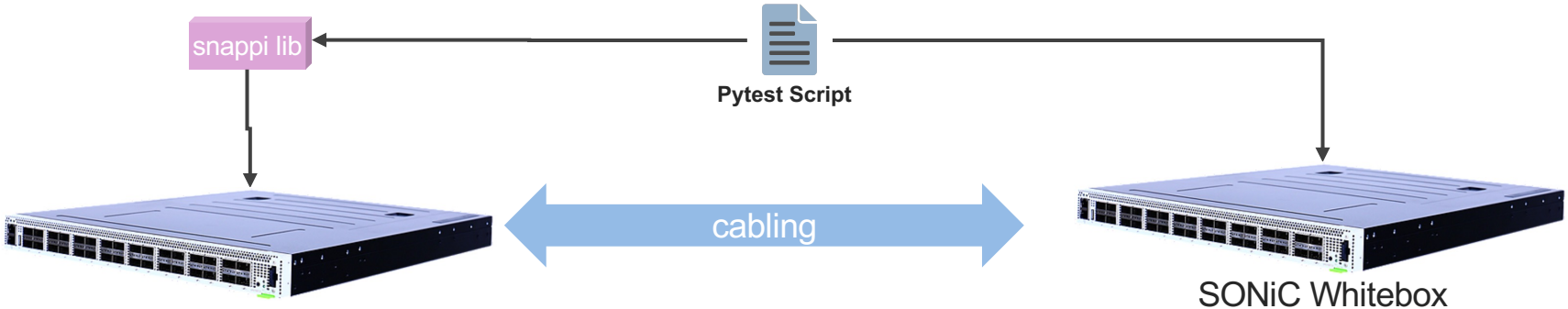


This technique was originally pioneered to enable hardware-based testers to perform real-time analysis of line-rate traffic prior to economically-viable protocol parsing engines (like P4 ASICs). The same technique can be done in CPUs (Athena) at lower speeds (approaching 100Gbps for larger packet sizes, limited by the packet-per-second rate).

Use-case: CI/CD Pipeline End-to-End Test using snappi



Use-case: SONiC PFC test using + snappi + Ixia tester



Test are being added to the SONiC testbed suite using snappi to control a traffic generator.