To set the context

Let’s look into deployment options for programmable switches
Whitebox Deployment

- Maximum flexibility
- Maximum disruption/risk/work

NOS (e.g., Cumulus)

Remote controller/NOS (e.g., ONOS)

Platform vendor (Cisco)
Chip vendor (Barefoot)
Customer/open source

Programmable chip
Turn-key Deployment

- Deployment as usual
  - Familiar features and interfaces
- Resource optimization
- Future proof
- Feature agility
- Streaming telemetry

NetOS

Profiles

profile1.p4
profile2.p4
profile3.p4

Programmable chip

Platform vendor (Cisco)
Chip vendor (Barefoot)
Customer/open source

• No flexibility
  • No custom feature and protocol support
Hybrid Deployment

• Best of breed
• Deployment as usual
  • Familiar features and interfaces
• Minimum development effort
  • Leverage existing functions in building new features

Minimize disruption and risk!
Challenges

Do not break what works
• Vendor data plane code is well tested
• … and we don’t want to need regression testing

Don’t want to show, don’t want to see
• Vendor code and custom code may be confidential
• Not practical to familiarize with a lot of vendor code to just write a few lines

Resource availability
• Still “limited” on current chips

Data/control plane dependence
• Net OS should keep working
• Net OS should not be aware of custom data plane functions
In a nutshell

P4 and its ecosystem were not designed for incremental programming

Single programmer

Single source code

Single control plane
We need to explicitly support Incremental Programming
How can we address these challenges?

**Challenges**

- **Do not break what works**
  - Vendor data plane code is well tested
  - …and we don’t want to need regression testing

- **Don’t want to show, don’t want to see**
  - Vendor code and custom code may be confidential
  - Not practical to familiarize with a lot of vendor code to just write a few lines

- **Resource availability**
  - Still “limited” on current chips

- **Data/control plane dependence**
  - NXOS should keep working
  - NXOS should not be aware of custom data plane functions

**Identify constraints on new code**

**Enforce those constraints on custom code**
Do’s and Don’t’s

- Do add new
  - Headers, parsers, tables, actions
- Do not modify existing
  - Headers, tables, actions
- Modify in a controlled way parsers and control flow

HW pipelined architecture

Isolation from existing P4 program

Compiler mapping of tables on HW

No API changes

NetOS unaffected
Incremental Programming Data Plane Environment daPIPE
Support developers and streamline their task (while enforcing constraints)
Components of the Solution

- daPIPE Graphical User Interface
- daPIPE build environment
- Control program
- Nexus 34180YC
- Nexus 3464C

```c
#define FLOW_PORT_HASH_WID
#define EMPTY_FLOW_PORT_EN

header_type metadata_t {
    set_active_port
    modify_field(md.ingress_port, ig_intr_md.ingress_port);
}
```
Sample Usecase
Fox Networks Advanced Technology Group

0-20 sec

20-45 sec

45-60 sec

https://github.com/FOXNEOAdvancedTechnology/ts_switching_P4
Specification

- A switch shall forward packets based on the RTP timestamp they contain.

- If sent to 239.1.1.1, change destination address to 239.3.3.3 when RTP timestamp is
  - Between 0 and 2
  - Between from 5 and F

- If sent to 239.2.2.2, change destination address to 239.3.3.3 when RTP timestamp is
  - Between 3 and 4
Incremental Programming Unique Advantage

- Leverage existing features
  - Protocol parsing up to UDP messages
  - Layer 2-3 forwarding, including multicast packet forwarding
  - Multicast routing (offered by the operating system)

- Focus on new feature
  - Write just a few lines of P4 code and control code

**daPIPE bonus feature:**
no need to deal with the complexity of pre-existing code
Development Workflow

• Browse available (stock) metadata
• Define custom headers and metadata
• Specify parser(s) and their hook(s) in existing (stock) parsers
• Define custom tables and actions
• Specify control flow
• Compile and load on chip
• Develop control plane functionalities
Existing header view
Adding RTP header
Adding RTP parser
Resulting Parsing Code

```c
header_type ethernet_t {
  fields {
    dstAddr : 48;
    srcAddr : 48;
    etherType : 16;
  }
}
header ethernet_t ethernet;

header_type rtp_t {
  fields {
    version : 2;
    padding : 1;
    sequence_number : 16;
    timestamp : 32;
    SSRC : 32;
  }
}
header rtp_t rtp;
```

```c
parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType)
  {
    ETHERTYPE_IPV4 : parse_ipv4;
    default : ingress;
  }
}

parser parse_udp {
  extract(udp);
  return parse_rtp;
}

parser parse_rtp {
  extract(rtp);
  return ingress;
}
```
Add action
Adding a table
Define control flow
Compile and upload to switch
Control Plane and NetOS Support

- Cisco Apps
  - BGP
  - OSPF
- Customer Apps
  - Cfg
  - Ctrl plane

- Infrastructure
- HAL
- Cisco.p4
- Cu.p4

SW (mostly) control plane

HW data plane

Guest Shell (container)

NXOS

Controlled data plane API access

APIs generated by compiling P4 Programmable ASIC
Open Challenges

• On the customer side
  • Debugging
  • Access to the right level of knowledge on the stock P4 program

• On switching system vendor side
  • Support model
  • Troubleshooting issues
    • Identify whether related to stock code of customer code

• On programmable ASIC vendor side
  • Offer technical support directly to the end customer for chip/compiler related problems