Open Programmable Networks

Spiros Eliopoulos (Cornell)
Nate Foster (Cornell)
Arjun Guha (UMass Amherst)

ONUG Fall 2014
We are at the start of a revolution!
In Congress, July 4, 1776.

The unanimous Declaration of the thirteen united States of America.

When in the Course of human Events, it becomes necessary for one People to dissolve the political Bands which have connected them with another; and to judiciously pursue the objects of perfection, subjected to them by the Laws of Nature, and of Nature's God; or the consent of the People, a decent respect to the Opinions of mankind requires that they should declare the causes which impel them to this resolution.

We hold these Truths to be self-evident, that all Men are created equal; that they are endowed by their Creator with certain inalienable Rights; that among these are Life, Liberty, and the pursuit of Happiness. That to secure these Rights, Governments are instituted among Men, deriving their just Powers from the Consent of the Governed. That whenever any Form of Government becomes destructive of these Ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its Foundation on such Principles, and organizing its Powers in such Form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictate that Governments long established should not be changed for light and transient Causes; and accordingly will roll the present System of Government.

He has refused his Assent to Laws, the most wholesome and necessary for the public good.

He has refused to pass other Laws for the accommodation of large districts of people, unless those people would relinquish the right of Representation in the Legislature, as the original Right of free Subjects. He has refused to pass such Laws, for reducing unsuitable, occupying too much of the public service. He has refused to pass such Laws; and, by deploying的力量, to secure the submission of these Disaffected people. He has called a Assembly, and assembled the** four out of every five people; and, when so assembled, has utterly refused to attend to the Petitions of the People; or, of the People.

He has kept the Law for the accommodation of large districts of people, unless those people would relinquish the right of Representation in the legislature, as the original Right of free Subjects. He has refused to pass such Laws; and, by deploying the force of the Community, to secure the submission of these Disaffected People.

He has called a Special Assembly, and assembled the** four out of every five people; and, when so assembled, has utterly refused to attend to the Petitions of the People; or, of the People.

He has kept the Law for the accommodation of large districts of people, unless those people would relinquish the right of Representation in the legislature, as the original Right of free Subjects. He has refused to pass such Laws; and, by deploying the force of the Community, to secure the submission of these Disaffected People.

He has called a Special Assembly, and assembled the** four out of every five people; and, when so assembled, has utterly refused to attend to the Petitions of the People; or, of the People.
Networks have been opened up... giving programmers the freedom to write software that tailors their behavior to suit specific applications!
Open Networking Successes

Data Center Virtualization
• Write programs against virtual topologies
• Controller maps virtual programs to physical network

Traffic Monitoring
• Declare continuous traffic queries
• Controller polls counters and aggregates results

Verification and Debugging
• Specify behavior using high-level properties
• Controller generates code to enforce key invariants

Traffic Engineering
• Optimize bandwidth according to natural criteria
• Controller provisions paths using constraint solver
Open Networking Architecture

Application  Application  Application

Northbound Controller

Southbound Controller

Application  Application  Application
Southbound Interfaces

There are now many ways to manage network device configurations programmatically:

- NetConf
- OpenFlow
- OVS
- P4
- SNMP
- YANG
- etc.

These interfaces, which are rapidly maturing, provide a solid foundation for network programming.
But on the northbound side... the situation is bleak.

Current controllers provide a variety of abstractions:

- Device abstraction layers
- Isolated slices
- Virtual networks
- QoS provisioning
- NFV service chaining
- Custom services (discovery, firewall, etc.)

But the development of these abstractions has been ad hoc, driven more by the needs of particular applications than by fundamental principles.
High-level abstractions

Northbound Interface Design

Good performance

Resource allocation

Modularity
Modular Composition

(Route + Monitor) ; Firewall

Northbound Controller

Southbound Controller
Modular Composition

Monolithic application

(Route + Monitor) ; Firewall

Northbound Controller

Southbound Controller
Modular Composition

This style of programming complicates:
- Writing, testing, and debugging programs
- Reusing code across applications
- Porting applications to new platforms
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Forward 2</td>
</tr>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
</tbody>
</table>

**Route** + **Monitor**
### Route

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

### Monitor

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
</tbody>
</table>

### Route + Monitor

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.1</td>
<td>Forward 1, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.2</td>
<td>Forward 2, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
<tr>
<td>dstip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>
## Route + Monitor

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.1</td>
<td>Forward 1, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.2</td>
<td>Forward 2, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
<tr>
<td>dstip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>
### Route + Monitor

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.1</td>
<td>Forward 1, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4, dstip=10.0.0.2</td>
<td>Forward 2, Count</td>
</tr>
<tr>
<td>srcip=1.2.3.4</td>
<td>Count</td>
</tr>
<tr>
<td>dstip=10.0.0.1</td>
<td>Forward 1</td>
</tr>
<tr>
<td>dstip=10.0.0.2</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

### Firewall

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcpdst = 22</td>
<td>Drop</td>
</tr>
<tr>
<td>*</td>
<td>Fwd ?</td>
</tr>
</tbody>
</table>
Route + Monitor

Firewall

(Route + Monitor)

Firewall
Machine Languages

Current APIs are derived from the underlying machine languages.

Programmers must work in terms of low-level concepts such as:

- Flow tables
- Matches
- Priorities
- Timeouts
- Events

This approach complicates programs and reasoning.
Better would be to have APIs based on higher-level, more intuitive abstractions. Then, programmers could work in terms of natural concepts such as:

- Logical predicates
- Mathematical functions
- Network-wide paths
- Policy combinators
- Atomic transactions

which would streamline many programs and simplify reasoning.
Programming Languages

Better would be to have APIs based on higher-level, more intuitive abstractions. Then, programmers could work in terms of natural concepts such as:

- Logical predicates
- Mathematical functions
- Network-wide paths
- Policy combinators
- Atomic transactions

which would streamline many programs and simplify reasoning.
Our Vision

• Write network programs in a high-level language
• Generate efficient low-level code using a compiler
• Reason about network properties automatically

Main Results

• Designed language based on an elegant and compositional foundation: packet-processing functions
• Built compilers and run-time systems that implement these languages efficiently on OpenFlow hardware
• Showed how to encode other high-level abstractions using functions (e.g., slicing, virtualization, QoS, etc.)
Frenetic Architecture

Application

NetKAT Language

Run-Time System

→ ─── ─── ─── ←
Frenetic Architecture

Application

NetKAT Language

Run-Time System

High-level application logic

Often expressed as a finite-state machine on network events (topology changes, new connections, etc.)
Frenetic Architecture

Programs describe packet-processing functions

Compile to a collection of forwarding tables, one per switch in the network
Frenetic Architecture

Translation configuration updates into sequences of OpenFlow instructions

Code that manages the rules installed on switches

Application

NetKAT Language

Run-Time System
Frenetic Architecture

Application

NetKAT Language

Run-Time System

Forwarding elements that implement packet-processing functionality efficiently in hardware
NetKAT Language
Semantic Design
Semantic Design

Packets

A record comprising the location of the packet and a collection of header values

\{\text{switch}=n_1, \text{port}=n_2, \text{ethSrc}=n_3, \ldots\}
Semantic Design

Packets
A record comprising the location of the packet and a collection of header values
\{\text{switch}=n_1, \text{port}=n_2, \text{ethSrc}=n_3, \ldots\}

Policies
Denotes total functions on packets (really packet histories)
Semantic Design

Packets
A record comprising the location of the packet and a collection of header values

\{\text{switch}=n_1, \text{port}=n_2, \text{ethSrc}=n_3, \ldots\}

Policies
Denotes total functions on packets (really packet histories)

Predicates
Restrict the behavior of a program to a particular set of packets using predicates
Semantic Design

Packets
A record comprising the location of the packet and a collection of header values
\{switch=n_1, \text{port}=n_2, \text{ethSrc}=n_3, \ldots\}

Policies
Denotes total functions on packets (really packet histories)

Predicates
Restrict the behavior of a program to a particular set of packets using predicates

Combinators
Combine smaller programs into bigger ones via natural mathematical operations
Semantic Design

**Packets**
A record comprising the location of the packet and a collection of header values

\{\text{switch}=n_1, \text{port}=n_2, \text{ethSrc}=n_3, \ldots\}

**Policies**
Denotes total functions on packets (really packet histories)

**Predicates**
Restrict the behavior of a program to a particular set of packets using predicates

**Combinators**
Combine smaller programs into bigger ones via natural mathematical operations

Functional “see every packet” abstraction
Primitive Design

What constructs should an SDN language provide?
What constructs should an SDN language provide?

- Packet predicates
- Packet transformations
Primitive Design

What constructs should an SDN language provide?

- Packet predicates
- Packet transformations
- Path construction
Primitive Design

What constructs should an SDN language provide?

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
What constructs should an SDN language provide?

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union
What constructs should an SDN language provide?

- Packet predicates
- Packet transformations
- Path construction
- Path concatenation
- Path union
- Path iteration
NetKAT Language

Syntax

\[
pol ::= \text{false} \\
    \text{true} \\
    \text{field} = \text{val} \\
    \text{field} ::= \text{val} \\
    pol_1 + pol_2 \\
    pol_1 ; pol_2 \\
    \neg pol \\
    pol^* \\
    \text{dup}
\]

Semantics

Functions from packet histories to sets of packet histories

Syntactic Sugar

\[
\text{if} \ pol \ \text{then} \ pol_1 \ \text{else} \ pol_2 \triangleq (pol; pol_1) + (\neg pol; pol_2)
\]
NetKAT Language

**Syntax**

\[
pol ::= \text{false} \mid \text{true} \mid \text{field} = \text{val} \mid \text{!pol} \mid \text{pol}^* \mid \text{dup}
\]

**Semantics**

NetKAT can encode switch configurations, network-wide paths, and even topologies.

\[
\text{if } pol \text{ then } pol_1 \text{ else } pol_2 \triangleq
\]
\[
(pol; pol_1) + (!pol; pol_2)
\]

Functions from packet histories
false drops its input
true copies its input
field = val copies its input if pk.field = val or drops it if not

field = val
field = val copies its input if pk.field = val or drops it if not when pk.field ≠ val.
field := val sets the input’s field component to val
pol₁ + pol₁ duplicates the input, sends one copy to each sub-policy, and takes the union of their outputs.
\( \text{pol} := \text{false} \mid \text{true} \mid \text{field} = \text{val} \mid \text{field} := \text{val} \mid \text{pol_1} + \text{pol_2} \mid \text{pol_1} ; \text{pol_2} \mid !\text{pol} \mid \text{pol_1}^* \mid \text{dup} \)

\( \langle \text{pk},.. \rangle \) runs the input through \( \text{pol_1} \) and then runs every output produced by \( \text{pol_1} \) through \( \text{pol_2} \).
\texttt{pol} drops the input if \texttt{pol} produces any output and copies it otherwise.
pol* repeatedly runs packets through pol to a fixpoint
dup duplicates the head packet of the input
Example

Topology
Example

**Topology**

**Specification**
- Forward packets to hosts 1-4
- Monitor traffic to unknown hosts
- Flood broadcast traffic to all hosts
- Disallow SSH traffic from hosts 1-2

**Flow Table**

```
{pattern={ethSrc=00:00:00:00:00:01,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethSrc=00:00:00:00:00:02,ethTyp=0x800,ipProto=0x06, tcpDstPort=22},action=[]}
{pattern={ethDst=00:00:00:00:00:01},action=[Output(1)]}
{pattern={ethDst=00:00:00:00:00:02},action=[Output(2)]}
{pattern={ethDst=00:00:00:00:00:03},action=[Output(3)]}
{pattern={ethDst=00:00:00:00:00:04},action=[Output(4)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff, port=1}, action=[Output(4), Output(3), Output(2)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff, port=2}, action=[Output(4), Output(3), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff, port=3}, action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff, port=4}, action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff}, action=[]}
{pattern={}, action=[Controller]}
```
Example: Forward

```plaintext
let forward =
  if ethDst = 00:00:00:00:00:01 then
    port := 1
  else if ethDst = 00:00:00:00:00:02 then
    port := 2
  else if ethDst = 00:00:00:00:00:03 then
    port := 3
  else if ethDst = 00:00:00:00:00:04 then
    port := 4
  else
    false
```
let flood =
    if port = 1 then
        port := 2 + port := 3 + port := 4
    else if port = 2 then
        port := 1 + port := 3 + port := 4
    else if port = 3 then
        port := 1 + port := 2 + port := 4
    else if port = 4 then
        port := 1 + port := 2 + port := 3
    else
        false

let broadcast =
    if ethDst = ff:ff:ff:ff:ff:ff:ff then
        flood
    else
        false
Example: Routing

```plaintext
let route = forward + broadcast
```
```ocaml
let monitor =
  if !(ethDst = 00:00:00:00:00:01 +
    ethDst = 00:00:00:00:00:02 +
    ethDst = 00:00:00:00:00:03 +
    ethDst = 00:00:00:00:00:04 +
    ethDst = ff:ff:ff:ff:ff:ff) then
    port := unknown
  else
    false
```
Example: Firewall

let firewall =
  if (ethSrc = 00:00:00:00:00:01 +
      ethSrc = 00:00:00:00:00:02) ;
  ethTyp = 0x800 ;
  ipProto = 0x06 ;
  tcpDstPort = 22 then
    false
  else
    true
Example: Main Policy

```plaintext
let main = (route + monitor); firewall

compiles to...

{pattern={ethSrc=00:00:00:00:00:01,ethTyp=0x800,ipProto=0x06/tcpDstPort=22},action=[]}
{pattern={ethSrc=00:00:00:00:00:02,ethTyp=0x800,ipProto=0x06/tcpDstPort=22},action=[]}
{pattern={ethDst=00:00:00:00:00:01},action=[Output(1)]}
{pattern={ethDst=00:00:00:00:00:02},action=[Output(2)]}
{pattern={ethDst=00:00:00:00:00:03},action=[Output(3)]}
{pattern={ethDst=00:00:00:00:00:04},action=[Output(4)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=1},action=[Output(4), Output(3), Output(2)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=2},action=[Output(4), Output(3), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=3},action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff,port=4},action=[Output(4), Output(2), Output(1)]}
{pattern={ethDst=ff:ff:ff:ff:ff:ff},action=[]}
{pattern={},action=[Controller]}
```
Demo
Demo Application

- Discovery + (Whitelist; Learning)
- NetKAT Language
- Run-Time System

Diagram showing a tree structure with servers.
Dynamic Applications
Dynamic Applications

Application

NetKAT Policy

NetKAT Language

Run-Time System
Dynamic Applications

Application

NetKAT Language

Run-Time System

Topology change

NetKAT Policy
Dynamic Applications

- Application
  - Topology change
  - Host change
- NetKAT Language
- Run-Time System
- NetKAT Policy
Dynamic Applications

Application

NetKAT Language

Run-Time System

Topography change
Host change
Traffic statistics

NetKAT Policy
Composing Applications

One module for each task

- Monitor
- Route
- Firewall
- Load Balance

NetKAT Language

Run-Time System

Benefits:
- Easier to write, test, and debug programs
- Can reuse modules across applications
- Possible to port applications to new platforms
Application Interface

Supports dynamic, stateful applications

```
type app
type event =
  | PacketIn of string * switchId * portId * payload * int
  | SwitchUp of switchId
  | SwitchDown of switchId
  | ...

type handler = event -> policy option
val create: policy -> handler -> app
val par: app -> app -> app
val seq: app -> app -> app
```

Concurrency via Jane Street’s `async` library

RESTful interface: write NetKAT applications in Python!
# switch state
table = {}

# helper functions
def learn(sw, pkt, pt):
    table[sw][get_ethernet(pkt).src] = pt

def switch_policy(sw):
    def f((known, unknown), mac):
        src = test("ethSrc", mac)
        dst = test("ethDst", mac)
        return (known | filter(dst) >> output(table[sw][mac]), unknown & ~src)
    (known_pol, unknown_pred) = reduce(f, table[sw].keys(), (drop(), true()))
    return known_pol | filter(unknown_pred) >> (controller() | flood(sw))

def policy():
    return union(switch_policy(sw) for sw in table.keys())

# event handler
def handler(_, event):
    print event
    typ = event['type']
    if typ == 'packet_in':
        pkt = packet.Packet(base64.decode(event['payload']['buffer']))
        learn(event['switch_id'], pkt, event['port_id'])
    else:
        pass
    return PolicyResult(policy())
Run-Time System
Run-Time System

Application

NetKAT Language

Run-Time System

Code that manages the rules installed on switches
Translate configuration updates into sequences of OpenFlow instructions

```ml
let new_update_for t =
  let t = List.fold new_table (~init:([], max_priority)) (~f:(fun (acc, pri) x -> (x, pri) :: acc, pri + 1)) in

let new_table = List.rev new_table in

let del_table = List.rev (flowtable_diff old_table new_table) in

let to_flow_mod prio flow =
  M.FlowModMsg (SDN_OpenFlow0x01.from_flow/prio/flow with command=/AddStrictFlow) in

let to_flow_del prio flow =
  M.FlowModMsg (SDN_OpenFlow0x01.from_flow/prio/flow with command=/DeleteStrictFlow) in

let new_table, _ = Deferred.List.iter new_table (~f:(fun (flow, pri) ->
                          send t.ctl/c_id (0l, to_flow_mod/prio/flow))) in

let del_table, _ = Deferred.List.iter del_table (~f:(fun (flow, pri) ->
                          send t.ctl/c_id (0l, to_flow_del/prio/flow))) in

let new_table = SwitchMap.add t.edge/new_table in
```
Network Updates

**Question:** how can we gracefully transition the network from one configuration to another?
Question: how can we gracefully transition the network from one configuration to another?
**Question:** how can we gracefully transition the network from one configuration to another?

**Must reason about all possible packet interleavings!**

**Target Policy**
Approach: develop abstractions that appear to update all of the switches in the network at once

Consistency Property: every packet (or flow) in the network “sees” a single policy version

Implementations:
- Order updates
- Unobservable updates
- One-touch updates
- Compositions of consistent
- Two-phase update
Two-Phase Updates

**Versioning:** instrument the compiler so that all forwarding rules match on a policy version

**Unobservable Update:** install the rules for the new policy in the interior of the network

**One-Touch Updates:** install rules at the edge that stamp packets with new version

**Garbage Collect:** delete the rules for the old policy
Two-Phase Updates

**Versioning:** instrument the compiler so that all forwarding rules match on a policy version.

**Unobservable Update:** install the rules for the new policy in the interior of the network.

**Theorem:** Unobservable + One-Touch = Consistent

**One-Touch Updates:** install rules at the edge that stamp packets with new version.

**Garbage Collect:** delete the rules for the old policy.
Applications
Rich Applications with NetKAT

- Isolated Slices
- Virtual Networks
- Network Debugging
- Fault Tolerance
- Quality of Service Provisioning
- Network Function Virtualization
Isolated Slices

In many situations, multiple tenants must share the network…
…but we don’t want their traffic to interfere with each other!

\[
\begin{align*}
\text{[Diagram of tree structures representing isolated slices]} &\quad = \quad & \text{[Diagram of tree structures representing isolated slices]} + \text{[Diagram of tree structures representing isolated slices]}
\end{align*}
\]
Isolated Slices

In many situations, multiple tenants must share the network… …but we don’t want their traffic to interfere with each other!

\[
\begin{align*}
\{ \text{in} \} \times : \text{pol} \{ \text{out} \}
\end{align*}
\]
Isolated Slices

In many situations, multiple tenants must share the network…
…but we don’t want their traffic to interfere with each other!

\[
\begin{align*}
\{ \text{in} \} x : \text{pol} & \{ \text{out} \} \\
\text{let } \text{pre} &= (\text{tag} = \text{none}; \text{in}; \text{tag} := x + \text{tag} = x) \text{ in} \\
\text{let } \text{post} &= (\text{out}; \text{tag} := \text{none} + !\text{out}) \text{ in} \\
&\text{(pre; pol; post)}
\end{align*}
\]
Virtualization

Often useful to programs against a simplified network topology

- **Information hiding**: limit what modules see
- **Protection**: limit what modules can do
- **Code reuse**: limit what dependencies modules have
Example: Gateway

- **Left:** learning switch on MAC addresses
- **Middle:** ARP on gateway, plus simple repeater
- **Right:** shortest-path forwarding on IP prefixes
Implementing Virtualization

Virtual Network

Physical Network
Implementing Virtualization

Virtual Network

Physical Network
Implementing Virtualization

Virtual Network

Physical Network
Implementing Virtualization

- Virtual Network
- Physical Network
Implementing Virtualization

Virtual Network

Physical Network
Implementing Virtualization

Virtual Network

Physical Network
Implementing Virtualization

Virtual Network

Physical Network
Implementing Virtualization

Virtual Network

Physical Network

This idiom can be implemented in NetKAT!

\[\text{ingress;}\]
\[(\text{raise;} \, \text{application;} \, \text{lower;} \, \text{fabric})^*;\]
\[\text{egress}\]
Network Debugging
Network Debugging

Often want to answer questions like:
• Does the network forward packets from A to B?
• Does the network block packets of type X?
• Does the network contain forwarding loops?
Network Debugging

We can encode entire networks as NetKAT policies and check these properties (and others) automatically.

Often want to answer questions like:

• Does the network forward packets from A to B?
• Does the network block packets of type X?
• Does the network contain forwarding loops?
Encoding Tables

Encoding switch forwarding tables is straightforward using NetKAT’s conditionals

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dstport=22</td>
<td>Drop</td>
</tr>
<tr>
<td>srcip=10.0.0.0/8</td>
<td>Forward 1</td>
</tr>
<tr>
<td>*</td>
<td>Forward 2</td>
</tr>
</tbody>
</table>

```
if dstport=22 then false
else if srcip=10.0.0.0/8 then
  port := 1
else
  port := 2
```
Encoding topologies is also straightforward using NetKAT’s tests, modifications, and union.

```
switch=A; port=1; switch:=B; port:=2 + switch=B; port=2; switch:=A; port:=1 + switch=B; port=1; switch:=C; port:=2 + switch=C; port=2; switch:=B; port:=1
```
A network can be encoded by alternating between policy and topology packet-processing steps.
A network can be encoded by alternating between policy and topology packet-processing steps.
A network can be encoded by alternating between policy and topology packet-processing steps.

\[
\text{true} + (\text{policy}; \text{topo})
\]
Encoding Networks

A network can be encoded by alternating between policy and topology packet-processing steps.

\[
\text{true} + (\text{policy}; \text{topo}) + (\text{policy}; \text{topo}; \text{policy}; \text{topo})
\]
Encoding Networks

A network can be encoded by alternating between policy and topology packet-processing steps.

\[
\text{true} \\
+ \\
(policy; \text{topo}) \\
+ \\
(policy; \text{topo}; policy; \text{topo}) \\
+ \\
(policy; \text{topo}; policy; \text{topo}; policy; \text{topo})
\]
A network can be encoded by alternating between policy and topology packet-processing steps.

```
true
+ 
(policy; topo)
+ 
(policy; topo; policy; topo)
+ 
(policy; topo; policy; topo; policy; topo)
   : 
+ 
(policy; topo)*
```
A network can be encoded by alternating between policy and topology packet-processing steps.

To check whether the network drops packets of type $X$, check if $\text{type}=X; (\text{policy}; \text{topo})^*$ is equivalent to $\text{false}$.

- $(\text{policy}; \text{topo})$
- $+$
- $(\text{policy}; \text{topo}; \text{policy}; \text{topo})$
- $+$
- $(\text{policy}; \text{topo}; \text{policy}; \text{topo}; \text{policy}; \text{topo})$
- $+$
- $(\text{policy}; \text{topo})^*$
Fault Tolerance

NetKAT Language

Run-Time System

Link down
Switch down
Fault Tolerance

- Red path or Blue path

NetKAT Language

Run-Time System

Path abstraction hides low-level faults

Link down

Switch down
Fault Tolerance

Run-Time System

- Red path
- Blue path

NetKAT Language

Path abstraction hides low-level faults

“Fast Failover” (OpenFlow 1.3+)

Link down
Switch down
Fault Tolerance

NetKAT’s path abstraction can also be used to build fault-tolerant network applications.

Path abstraction hides low-level faults

“Fast Failover” (OpenFlow 1.3+)

Run-Time System

Red path or Blue path
Quality of Service and NFV

- Give priority to VoIP over Web
- DPI on Web traffic
- Reserve bandwidth for Hadoop
- Block traffic from evil.com

Policy Constraint Solver
QoS (OVSDB)          Routing          Firewall

NetKAT Language
Run-Time System
3.1 Localization

Intuitively, a formula specifies the rate at which sources of various types of traffic may emit packets. Assume the universe of rates and that excluding multiplication ensures decidability.

Block traffic from evil.com by convention, policies without a rate clause are unconstrained—operators can be used to specify an aggregate cap on traffic, such as with

\[ \text{max}(x, 100\text{Mbps}) \]

\[ \text{min}(100\text{MB/s}) \]

Negation (\(\neg\)) or guarantee (\(\oplus\)) matches any single location.

\[ \neg \text{max}(x, 100\text{Mbps}) \]

\[ \text{max}(x, 100\text{Mbps}) \oplus \text{min}(100\text{MB/s}) \]

Wrong

Correct

To do this, the compiler takes as inputs the Merlin policy, a representation of the logical topology of the network, and an initial set of locations for each policy.

Given a policy, the compiler first parses it to create a set of logical nodes, each representing a packet-processing function or a bandwidth constraint. The compiler then constructs a statement NFA (NFAs are used to match sequences of network locations, including names of locations and functions) that represents the policy.

The NFA is then transformed into an LP graph, where each vertex represents a possible set of locations that can be matched by the policy. The LP graph is then used to find a feasible allocation of resources that satisfies the policy constraints.

For example, the following policy:

\[ \text{tcp.dst} = 80 \rightarrow \text{if} \neg \text{max}(x, 100\text{Mbps}) \text{then} \text{res.next} \rightarrow \text{hop} \text{else} \text{default} \text{end} \]

would result in the following LP graph:

```
+----------------+------------------+
|                |                  |
|                |                  |
|                |                  |
|                |                  |
|                |                  |
|                |                  |
|                |                  |
|                |                  |
+----------------+------------------+
```

The LP graph is then solved to find a feasible allocation of resources that satisfies the policy constraints.

Overall, Merlin's policy language enables direct expression of bandwidth constraints and packet-processing functions, while also allowing for the specification of aggregate guarantees—a natural and well-studied formalism for describing policies—using regular expressions.
Conclusion

• Programming languages have important role to play in the evolution of SDN programming interfaces
• NetKAT policy language provides a solid foundation for expressing and reasoning about packet-processing functions
• Many higher-level abstractions can be built on top of NetKAT
  - Isolated Slices
  - Virtual Networks
  - Network Debugging
  - Fault Tolerance
  - Quality of Service Provisioning
  - Network Function Virtualization
Thank you!

Collaborators

- Carolyn Anderson (Victoria)
- Shrutarshi Basu (Cornell)
- Marco Canini (UC Louvain)
- Andrew Ferguson (Google)
- Rodrigo Fonseca (Brown)
- Jean-Baptiste Jeannin (CMU)
- Dexter Kozen (Cornell)
- Robert Kleinberg (Cornell)
- Shriram Krishnamurthi (Brown)
- Chen Liang (Duke)
- Matthew Milano (Cornell)
- Jennifer Rexford (Princeton)
- Mark Reitblatt (Cornell)
- Cole Schlesinger (Princeton)
- Alexandra Silva (Nijmegen)
- Emin Gün Sirer (Cornell)
- Robert Soulé (Lugano)
- Laure Thompson (Cornell)
- Dave Walker (Princeton)

Papers, Code, etc.

http://frenetic-lang.org/