Software-Defined Networking and Advanced Network Control Programming

7 SDN Programming

Kai Gao
kaigao@scu.edu.cn

School of Cyber Science and Engineering
Sichuan University

Recap

Network Programming Language

Network programming language in this course refer to programming languages in the SDN architecture

- Compile high-level intents to low-level device configurations (flow rule, BGP configuration, etc.)
- Maintain policy compliance under network events (topology change, traffic change, policy change, etc.)

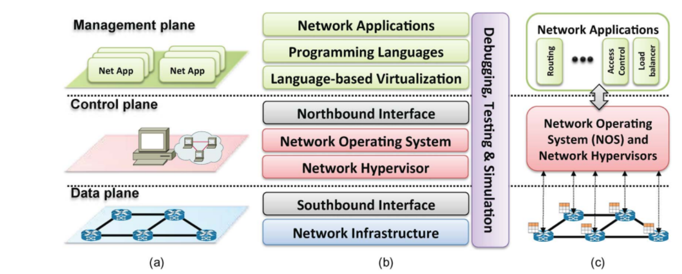


Fig. 6. *Software-Defined Networks in (a) planes, (b) layers, and (c) system design architecture.*

D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76

Frenetic

Problems

- Error-prone interaction between individual components

Solutions:

Program 1: Simple switching

```
def switch_join(switch):
  repeater(switch)
def repeater(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  install(switch,pat1,DEFAULT,None,[output(2)])
  install(switch,pat2,DEFAULT,None,[output(1)])
```

Program 2: Traffic monitoring

```
def monitor(switch):
  pat = {in_port:2,tp_src:80}
  install(switch,pat,DEFAULT,None,[])
  query_stats(switch,pat)
def stats_in(switch,xid,pattern,packets,bytes):
  print bytes
  sleep(30)
  query_stats(switch,pattern)
```

Frenetic

Problems

- Error-prone interaction between individual components
- Complex rule construction using low-level API

```
def repeater_monitor_noserver(switch):
  pat1 = {in_port:1}
  pat2 = {in_port:2}
  pat2web = {in_port:2,tp_src:80}
  pat2srv = {in_port:2,nw_dst:10.0.0.9,tp_src:80}
  install(switch,pat1,DEFAULT,None,[output(2)])
  install(switch,pat2srv,HIGH,None,[output(1)])
  install(switch,pat2web,MEDIUM,None,[output(1)])
  install(switch,pat2,DEFAULT,None,[output(1)])
  query_stats(switch,pat2web)
```

Solutions:

Frenetic

Problems

- Error-prone interaction between individual components
- Complex rule construction using low-level API
- Race conditions when programming both control and data plane

Solutions:

Program

```
def repeater_monitor_hosts(switch):
  pat = {in_port:1}
  install(switch,pat,DEFAULT,None,[output(2)])
def packet_in(switch,inport,packet):
  if inport == 2:
     mac = dstmac(packet)
     pat = {in_port:2,dl_dst:mac}
     install(switch,pat,DEFAULT,None,[output(1)])
     query_stats(switch,pat)
```

Packets in arriving order:

in_port: 1, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 80
in_port: 1, dl_dst: C5:85:2D:D6:B6:8B, tp_dst: 8080

Flow table:

match	*priority*	*action*
in_port: 1, dl_dst: C5:85:2D:D6:B6:8B	DEFAULT	output: 2
in_port: 1, dl_dst: C5:85:2D:D6:B6:8B	DEFAULT	output: 2

Frenetic

Problems

- Error-prone interaction between individual components
- Complex rule construction using low-level API
- Race conditions when programming both control and data plane

Solutions:

- Network query language

```
Queries      q ::= Select(a) *
                   Where(fp) *
                   GroupBy([qh_1, ..., qh_n]) *
                   SplitWhen([qh_1, ..., qh_n]) *
                   Every(n) *
                   Limit(n)
Aggregates   a ::= packets | sizes | counts
Headers     qh ::= inport | srcmac | dstmac | ethtype |
                   vlan | srcip | dstip | protocol |
                   srcport | dstport | switch
Patterns    fp ::= true_fp() | qh_fp(n) |
                   and_fp([fp_1, ..., fp_n]) |
                   or_fp([fp_1, ..., fp_n]) |
                   diff_fp(fp_1, fp_2) | not_fp(fp)
```

Figure 3. Frenetic query syntax

Frenetic

Problems

- Error-prone interaction between individual components
- Complex rule construction using low-level API
- Race conditions when programming both control and data plane

Solutions:

- Network query language
- Functional reactive network policy library

Events

`Seconds` $\in$ int E
`SwitchJoin` $\in$ switch E
`SwitchExit` $\in$ switch E
`PortChange` $\in$ (switch $\times$ int $\times$ bool) E
`Once` $\in \alpha \rightarrow \alpha$ E

Basic Event Functions

`>>` $\in \alpha$ E $\rightarrow \alpha\ \beta$ EF $\rightarrow \beta$ E
Lift $\in (a \rightarrow \beta) \rightarrow \alpha\ \beta$ EF
`>>` $\in \alpha\ \beta$ EF $\rightarrow \beta\ \gamma$ EF $\rightarrow \alpha\ \gamma$ EF
`ApplyFst` $\in \alpha\ \beta$ EF $\rightarrow (\alpha \times \gamma)\ (\beta \times \gamma)$ EF
`ApplySnd` $\in \alpha\ \beta$ EF $\rightarrow (\gamma \times \alpha)\ (\gamma \times \beta)$ EF
`Merge` $\in (\alpha$ E $\times\ \beta$ E$) \rightarrow (\alpha$ option $\times\ \beta$ option$)$ E
`BlendLeft` $\in \alpha \times \alpha$ E $\times\ \beta$ E $\rightarrow (\alpha \times \beta)$ E
`BlendRight` $\in \beta \times \alpha$ E $\times\ \beta$ E $\rightarrow (\alpha \times \beta)$ E
`Accum` $\in (\gamma \times (\alpha \times \gamma \rightarrow \gamma) \rightarrow \alpha\ \gamma$ EF
`Filter` $\in (\alpha \rightarrow$ bool$) \rightarrow \alpha\ \alpha$ EF

Listeners

`>>` $\in \alpha$ E $\rightarrow \alpha$ L $\rightarrow$ unit
`Print` $\in \alpha$ L
`Register` $\in$ policy L
`Send` $\in$ (switch $\times$ packet $\times$ action) L

Rules and Policies

`Rule` $\in$ pattern $\times$ action list $\rightarrow$ rule
`MakeForwardRules` $\in$ (switch $\times$ port $\times$ packet) policy EF
`AddRules` $\in$ policy policy EF

Figure 4. Selected Frenetic Operators.

Pyretic

Goal:

- Modular network policy

Solutions:

Monitor
srcip=5.6.7.8 → count

Route
dstip=10.0.0.1 → fwd(1)
dstip=10.0.0.2 → fwd(2)

Load-balance
srcip=0*,dstip=1.2.3.4 → dstip=10.0.0.1
srcip=1*,dstip=1.2.3.4 → dstip=10.0.0.2

Compiled Prioritized Rule Set for "Monitor | Route"
srcip=5.6.7.8,dstip=10.0.0.1 → count,fwd(1)
srcip=5.6.7.8,dstip=10.0.0.2 → count,fwd(2)
srcip=5.6.7.8 → count
dstip=10.0.0.1 → fwd(1)
dstip=10.0.0.2 → fwd(2)

Compiled Prioritized Rule Set for "Load-balance >> Route"
srcip=0*,dstip=1.2.3.4 → dstip=10.0.0.1,fwd(1)
srcip=1*,dstip=1.2.3.4 → dstip=10.0.0.2,fwd(2)

Pyretic

Goal:

- Modular network policy
- Programming on virtualized networks

Solutions:

Many-to-one Mapping:

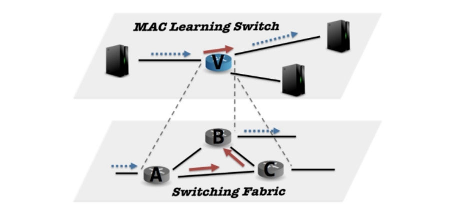


Figure 2: Many physical switches to one virtual.

One-to-many Mapping:

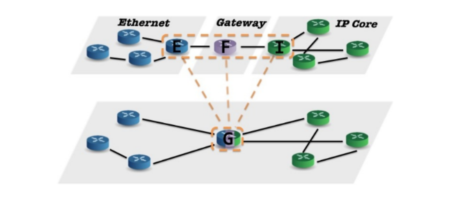


Figure 3: One physical switch to many virtual.

Pyretic

Goal:

- Modular network policy
- Programming on virtualized networks

Solutions:

- Composition of network policies

Primitive Actions:

A ::= `drop` | `passthrough` | `fwd(port)` | `flood` | `push(h=v)` | `pop(h)` | `move(h1=h2)`

Predicates:

P ::= `all_packets` | `no_packets` | `match(h=v)` | `ingress` | `egress` | P & P | (P | P) | ~P

Query Policies:

Q ::= `packets(limit,[h])` | `counts(every,[h])`

Policies:

C ::= A | Q | $P[C]$ | (C | C) | C >> C | `if_`(P,C,C)

Figure 4: Summary of static NetCore syntax.

Pyretic

```
def virtualize(ingress_policy,
               egress_policy,
               fabric_policy,
               derived_policy):
```

Goal:

- Modular network policy
- Programming on virtualized networks

Solutions:

- Composition of network policies
- Network objects and policy transformations

Maple

Goal:

- Algorithmic policy
- Automatic & correct rule generation

Solution:

- Trace tree
- Barrier rule

```python
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Maple

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

match	*priority*	*action* (on s1)
srcip=192.168.0.0/24, srcport=22	MEDIUM	output(2)
srcip=192.168.0.0/24	DEFAULT	output(2)

Goal:

- Algorithmic policy
- Automatic & correct rule generation

Solution:

- Trace tree
- Barrier rule

Maple

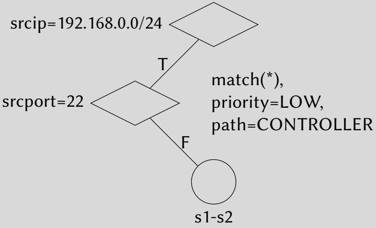
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Goal:

- Algorithmic policy
- Automatic & correct rule generation

Solution:

- Trace tree
- Barrier rule



Maple

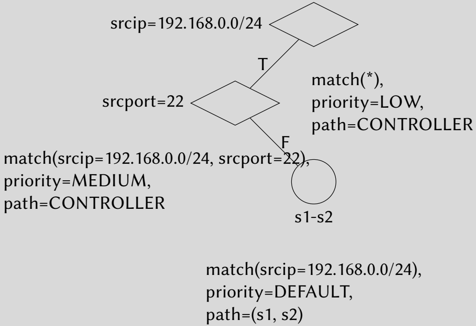
```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```

Goal:

- Algorithmic policy
- Automatic & correct rule generation

Solution:

- Trace tree
- Barrier rule



Magellan

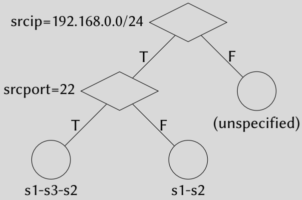
Goal:

- *Proactively* build rules for algorithmic policies

Solutions:

- Data flow analysis
- DFG partition and cost analysis

```
def outgoing_policy(pkt):
    if pkt.srcip in 192.168.0.0/24:
        if pkt.srcport == 22:
            use_path(s1, s3, s2)
        else:
            use_path(s1, s2)
```



Magellan

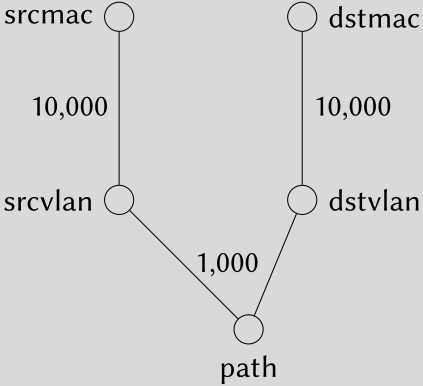
Goal:

- *Proactively* build rules for algorithmic policies

Solutions:

- Data flow analysis
- DFG partition and cost analysis

```
def policy(pkt):
    srcvlan = mac2vlan[pkt.srcmac]
    dstvlan = mac2vlan[pkt.dstmac]
    use_path(vlan2path[srcvlan, dstvlan])
```



Magellan

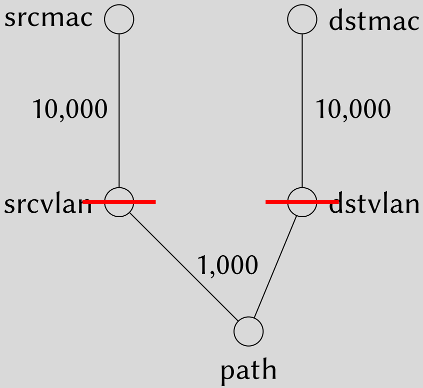
Goal:

- *Proactively* build rules for algorithmic policies

Solutions:

- Data flow analysis
- DFG partition and cost analysis

```
def policy(pkt):
    srcvlan = mac2vlan[pkt.srcmac]
    dstvlan = mac2vlan[pkt.dstmac]
    use_path(vlan2path[srcvlan, dstvlan])
```



Summary

In the coming lectures, we cover the following topics on network control programming

- SDN-based policy routing
- distributed enforcement of centralized policy
- programming stateful networks

Summary

In the coming lectures, we cover the following topics on network control programming

- SDN-based policy routing
- distributed enforcement of centralized policy
- programming stateful networks

In this lecture, you should

- learn network programming languages Merlin, Propane and SNAP

Summary

In the coming lectures, we cover the following topics on network control programming

- SDN-based policy routing
- distributed enforcement of centralized policy
- programming stateful networks

In this lecture, you should

- learn network programming languages Merlin, Propane and SNAP
- learn path constraints expressed in regular expressions

Summary

In the coming lectures, we cover the following topics on network control programming

- SDN-based policy routing
- distributed enforcement of centralized policy
- programming stateful networks

In this lecture, you should

- learn network programming languages Merlin, Propane and SNAP
- learn path constraints expressed in regular expressions
- learn how to use product graph to find policy compliant paths

Summary

In the coming lectures, we cover the following topics on network control programming

- SDN-based policy routing
- distributed enforcement of centralized policy
- programming stateful networks

In this lecture, you should

- learn network programming languages Merlin, Propane and SNAP
- learn path constraints expressed in regular expressions
- learn how to use product graph to find policy compliant paths
- learn how to use ILP to solve state placement problem

Merlin

Overview

Merlin: A Language for Provisioning Network Resources

Robert Soulé[*] Shrutarshi Basu[†] Parisa Jalili Marandi[*] Fernando Pedone[*]
Robert Kleinberg[†] Emin Gün Sirer[†] Nate Foster[†]
[*]University of Lugano [†]Cornell University

Robert Soulé et al. “Merlin: A Language for Provisioning Network Resources”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies.* CoNEXT ’14. New York, NY, USA: ACM, 2014, pp. 213–226. URL: http://doi.acm.org/10.1145/2674005.2674989

Goal

Support resource provisioning in SDN programming languages

There are two types of resource constraints:

- **Guarantee**: the minimum bandwidth the traffic can reach
- **Constraint**: the maximum bandwidth the traffic can reach

Example: Assume there are two flows f_1 and f_2 traversing a link with 5 Mbps

- rate of f_1: r_1, rate of f_2: r_2
- bandwidth of f_1: b_1, bandwidth of f_2: b_2
- guarantee of f_1: 2 Mbps
- constraint of f_1: 4 Mbps

Goal

Support resource provisioning in SDN programming languages

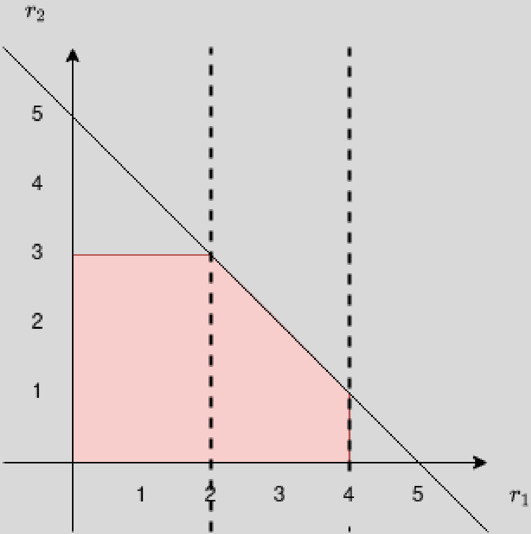
There are two types of resource constraints:

- **Guarantee**: the minimum bandwidth the traffic can reach
- **Constraint**: the maximum bandwidth the traffic can reach

Example: Assume there are two flows f_1 and f_2 traversing a link with 5 Mbps

- rate of f_1: r_1, rate of f_2: r_2
- bandwidth of f_1: b_1, bandwidth of f_2: b_2
- guarantee of f_1: 2 Mbps
- constraint of f_1: 4 Mbps

Feasible region of rate:



Goal

Support resource provisioning in SDN programming languages

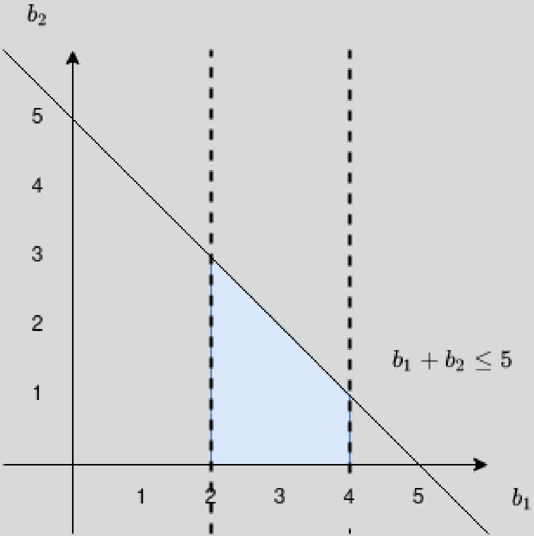
There are two types of resource constraints:

- **Guarantee**: the minimum bandwidth the traffic can reach
- **Constraint**: the maximum bandwidth the traffic can reach

Example: Assume there are two flows f_1 and f_2 traversing a link with 5 Mbps

- rate of f_1: r_1, rate of f_2: r_2
- bandwidth of f_1: b_1, bandwidth of f_2: b_2
- guarantee of f_1: 2 Mbps
- constraint of f_1: 4 Mbps

Feasible region of bandwidth:



Merlin Language

Merlin allows programmers to specify policies with resource requirements

$$
\begin{array}{rcll}
loc & \in & \textit{Locations} & \\
t & \in & \textit{Packet-processing functions} & \\
h & \in & \textit{Packet headers} & \\
f & \in & \textit{Header fields} & \\
v & \in & \textit{Header field values} & \\
id & \in & \textit{Identifiers} & \\
n & \in & \mathbb{N} & \\
pol & ::= & [s_1; \ldots; s_n], \phi & \text{Policies} \\
s & ::= & id : p \rightarrow r & \text{Statements} \\
\phi & ::= & \mathsf{max}(e, n) \mid \mathsf{min}(e, n) & \text{Presburger Formulas} \\
 & \mid & \phi_1 \text{ and } \phi_2 \mid \phi_1 \text{ or } \phi_2 \mid ! \, \phi_1 & \\
e & ::= & n \mid id \mid e + e & \text{Bandwidth Terms} \\
a & ::= & . \mid c \mid a \, a \mid a | a \mid a^* \mid ! \, a & \text{Path Expression} \\
p & ::= & p_1 \text{ and } p_2 \mid p_1 \text{ or } p_2 \mid ! \, p_1 & \text{Predicates} \\
 & \mid & h.f = v \mid \mathsf{true} \mid \mathsf{false} & \\
c & ::= & loc \mid t & \text{Path Element}
\end{array}
$$

Figure 1: Merlin abstract syntax.

Robert Soulé et al. “Merlin: A Language for Provisioning Network Resources”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’14. New York, NY, USA: ACM, 2014, pp. 213–226. URL: http://doi.acm.org/10.1145/2674005.2674989

Example

```
[ x : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 20) -> .* dpi .* ;
  y : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 21) -> .* ;
  z : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 80) -> .* dpi .* nat .* ],
  max(x + y, 50MB/s) and min(z, 100MB/s)
```

- This policy defines 3 flows: x, y and z
- Total bandwidth of flows x and y must not exceed 50 MB/s
- Bandwidth of flow z has a guarantee of 100 MB/s

A Closer Look at Flow Specification

```
x : (ip.src = 192.168.1.1 and
     ip.dst = 192.168.1.2 and
     tcp.dst = 20) -> .* dpi .* ;
```

- x: identifier
- `ip.src = 192.168.1.1 and ip.dst = 192.168.1.2 and tcp.dst = 20`: predicate
- `.* dpi .*`: path expression

$$
\begin{array}{rcll}
loc & \in & \textit{Locations} & \\
t & \in & \textit{Packet-processing functions} & \\
h & \in & \textit{Packet headers} & \\
f & \in & \textit{Header fields} & \\
v & \in & \textit{Header field values} & \\
id & \in & \textit{Identifiers} & \\
n & \in & \mathbb{N} & \\
pol & ::= & [s_1; \ldots; s_n], \phi & \text{Policies} \\
s & ::= & id : p \rightarrow r & \text{Statements} \\
\phi & ::= & \mathsf{max}(e, n) \mid \mathsf{min}(e, n) & \text{Presburger Formulas} \\
 & \mid & \phi_1 \text{ and } \phi_2 \mid \phi_1 \text{ or } \phi_2 \mid ! \; \phi_1 & \\
e & ::= & n \mid id \mid e + e & \text{Bandwidth Terms} \\
a & ::= & . \mid c \mid a\, a \mid a \,|\, a \mid a^* \mid ! \; a & \text{Path Expression} \\
p & ::= & p_1 \text{ and } p_2 \mid p_1 \text{ or } p_2 \mid ! \; p_1 & \text{Predicates} \\
 & \mid & h.f = v \mid \mathsf{true} \mid \mathsf{false} & \\
c & ::= & loc \mid t & \text{Path Element}
\end{array}
$$

Figure 1: Merlin abstract syntax.

Path Expression

Merlin uses path expression to specify the traversal constraint

Merlin path expression is a regular language and can be represented as a finite state automaton

Example

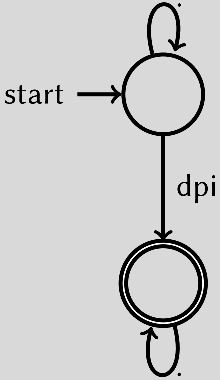
Path Expression

Merlin uses path expression to specify the traversal constraint

Merlin path expression is a regular language and can be represented as a finite state automaton

Example

- `.* dpi .*`



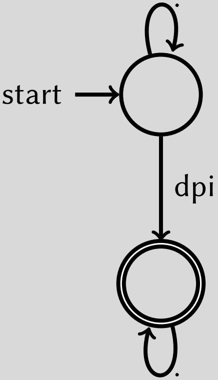
Path Expression

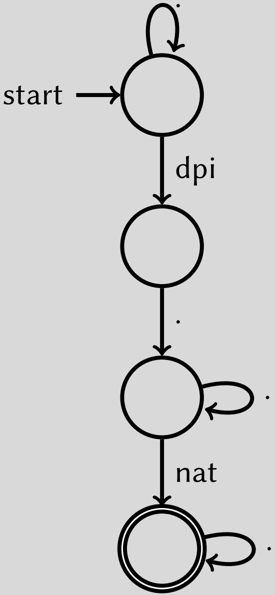
Merlin uses path expression to specify the traversal constraint

Merlin path expression is a regular language and can be represented as a finite state automaton

Example

- `.* dpi .*`
- `.* dpi .* nat .*`





Logical Topology

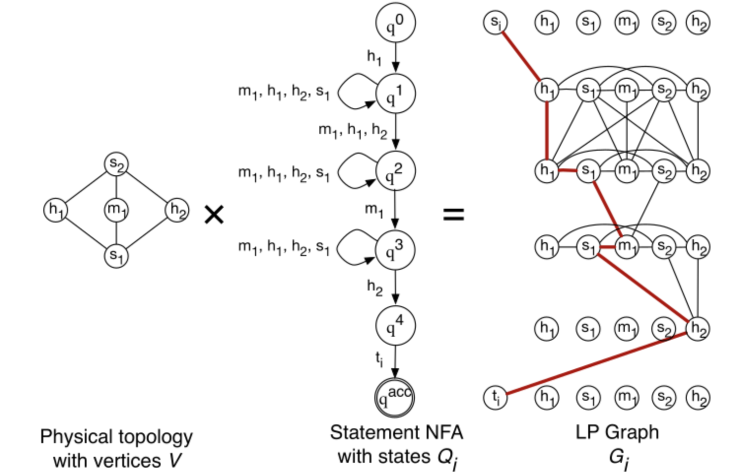


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

`h1 .* dpi .* nat .* h2`

Locations:

- `dpi`: `h1`, `h2`, or `m1`
- `nat`: `m1`

Logical Topology

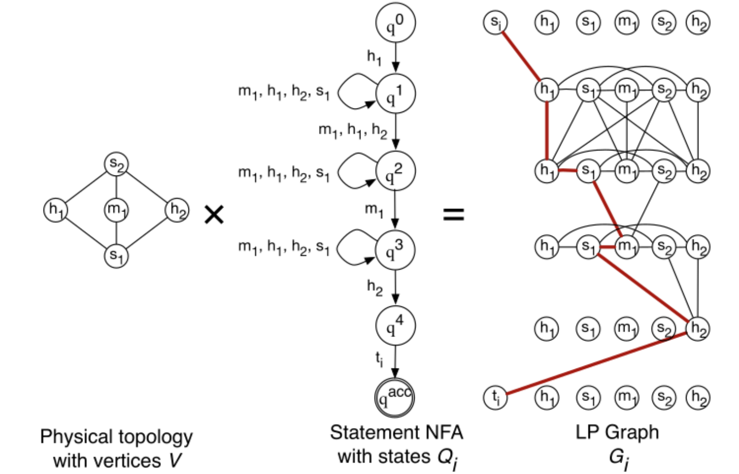


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

`h1 .* dpi .* nat .* h2`

Locations:

- `dpi`: `h1`, `h2`, or `m1`
- `nat`: `m1`

Path:

`h1`

Logical Topology

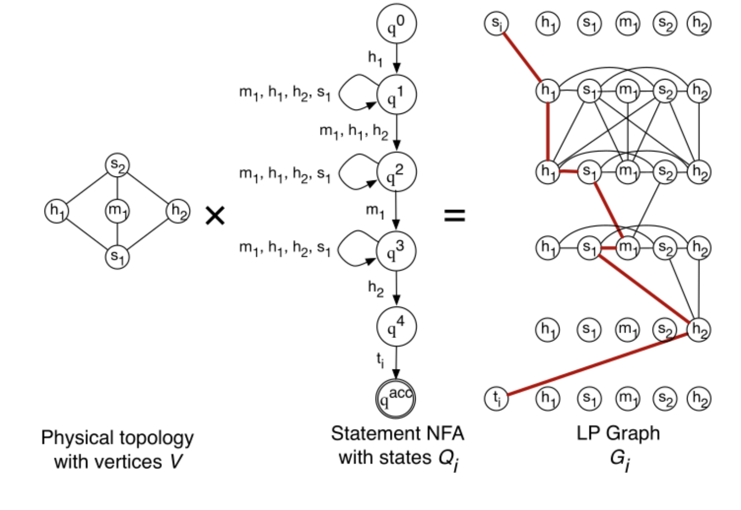


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

`h1 .* dpi .* nat .* h2`

Locations:

- `dpi`: `h1`, `h2`, or `m1`
- `nat`: `m1`

Path:

`h1`

Logical Topology

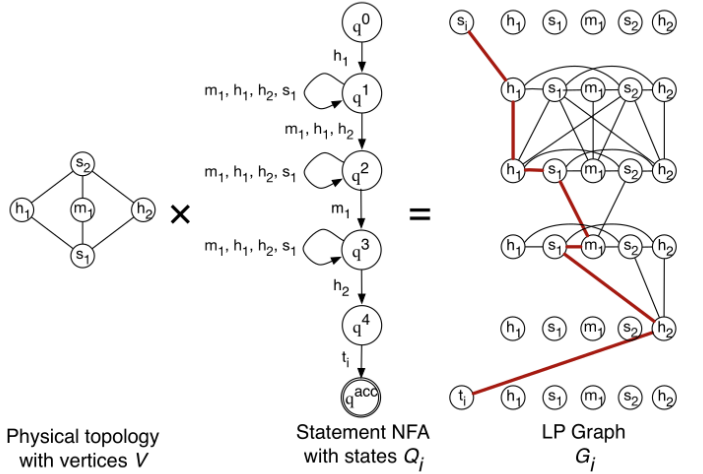


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

h1 .* dpi .* nat .* h2

Locations:

- dpi: h1, h2, or m1
- nat: m1

Path:

h1

Logical Topology

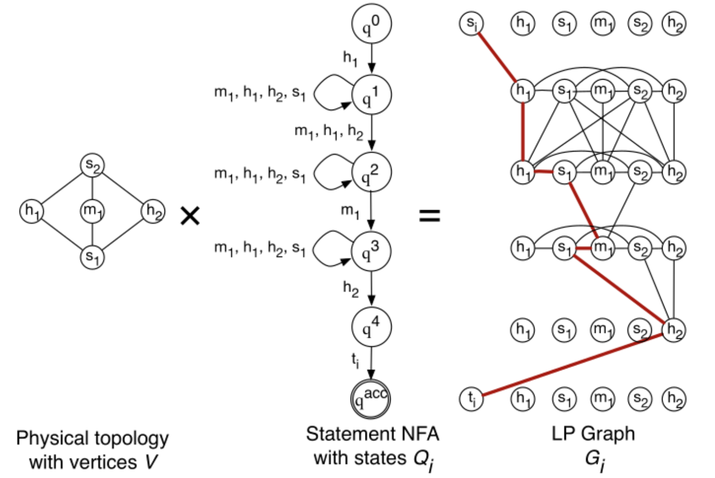


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

`h1 .* dpi .* nat .* h2`

Locations:

- `dpi`: `h1`, `h2`, or `m1`
- `nat`: `m1`

Path:

`h1 s1`

Logical Topology

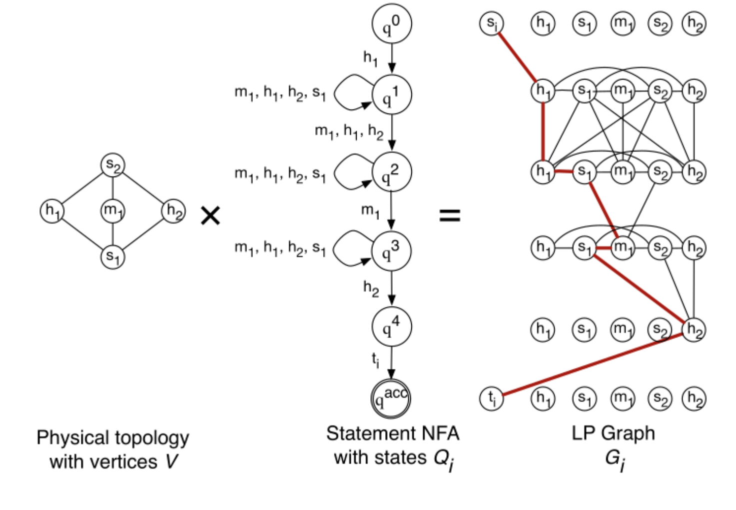


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

`h1 .* dpi .* nat .* h2`

Locations:

- `dpi`: `h1`, `h2`, or `m1`
- `nat`: `m1`

Path:

`h1 s1 m1`

Logical Topology

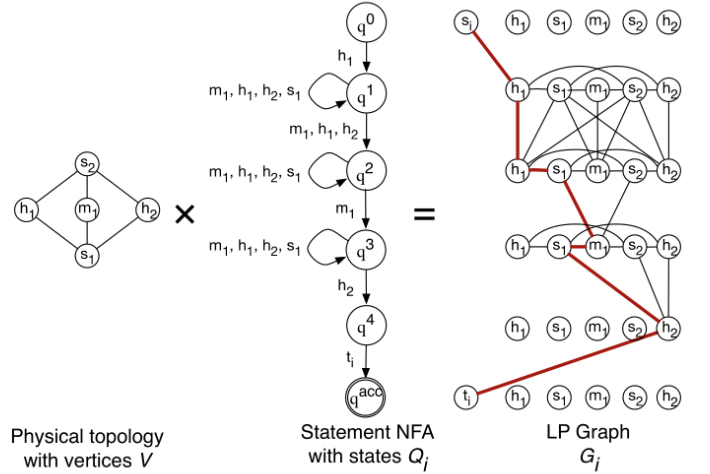


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

`h1 .* dpi .* nat .* h2`

Locations:

- `dpi`: `h1`, `h2`, or `m1`
- `nat`: `m1`

Path:

`h1 s1 m1 s1`

Logical Topology

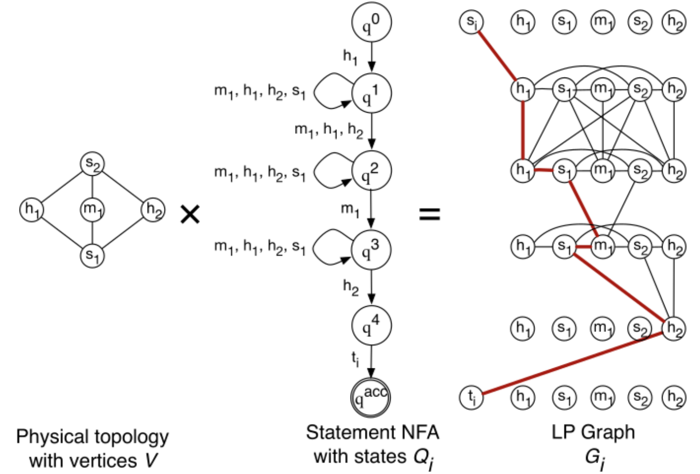


Figure 2: Logical topology for the example policy. The thick red path illustrates a solution.

Policy:

`h1 .* dpi .* nat .* h2`

Locations:

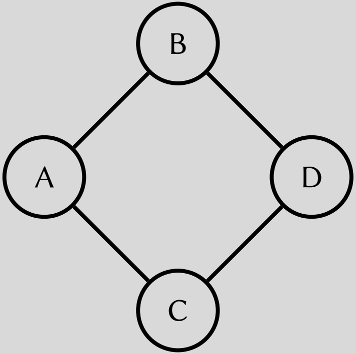
- `dpi`: `h1`, `h2`, or `m1`
- `nat`: `m1`

Path:

`h1 s1 m1 s1 h2`

Policy Localization

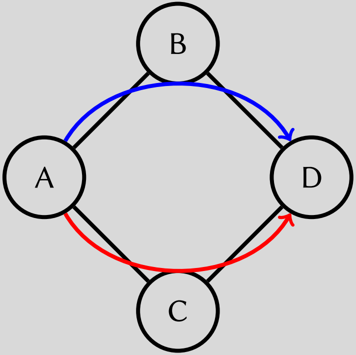
Bandwidth terms such as `max(x + y, 50 MB/s)` cannot be realized on hardware if they take different paths



Policy Localization

Bandwidth terms such as `max(x + y, 50 MB/s)` cannot be realized on hardware if they take different paths

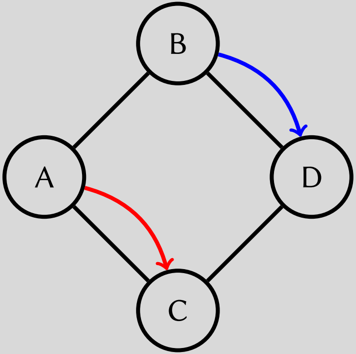
- `a->b->d`, `a->c->d`: can be enforced at `a` or `d`



Policy Localization

Bandwidth terms such as `max(x + y, 50 MB/s)` cannot be realized on hardware if they take different paths

- `a->b->d`, `a->c->d`: can be enforced at `a` or `d`
- `a->c`, `b->d`: cannot be enforced together



Policy Localization

Merlin transforms original policy into localized policy

```
[ x : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 20) -> .* dpi .* ;
  y : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 21) -> .* ;
  z : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 80) -> .* dpi .* nat .* ],
  max(x + y, 50MB/s) and min(z, 100MB/s)
```

```
[ x : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 20) -> .* dpi .* ;
  y : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 21) -> .* ;
  z : (ip.src = 192.168.1.1 and
       ip.dst = 192.168.1.2 and
       tcp.dst = 80) -> .* dpi .* nat .* ],
  max(x, 25MB/s) and max(y, 25MB/s) and min(z,
```

Global constraint `max(x + y, 50MB/s)` is transformed to localized constraint `max(x, 25MB/s) and max(y, 25MB/s)`

Policy Localization

- Constraints on multiple flows are transformed to constraints on single flows
- Localized constraints must still satisfy the global constraints

Example:

`min(x + y, 100 MB/s) and min(y + z, 60 MB/s) and max(y, 40 MB/s)`

× `min(x, 50MB/s) and min(y, 50MB/s) and min(y, 30MB/s) and min(z, 30MB/s)`

Policy Localization

- Constraints on multiple flows are transformed to constraints on single flows
- Localized constraints must still satisfy the global constraints

Example:

`min(x + y, 100 MB/s) and min(y + z, 60 MB/s) and max(y, 40 MB/s)`

× `min(x, 50MB/s) and min(y, 50MB/s) and min(y, 30MB/s) and min(z, 30MB/s)`

√ `min(x, 70MB/s) and min(y, 30MB/s) and min(z, 30MB/s)`

Policy Localization

- Constraints on multiple flows are transformed to constraints on single flows
- Localized constraints must still satisfy the global constraints

Example:

`min(x + y, 100 MB/s) and min(y + z, 60 MB/s) and max(y, 40 MB/s)`

× `min(x, 50MB/s) and min(y, 50MB/s) and min(y, 30MB/s) and min(z, 30MB/s)`

√ `min(x, 70MB/s) and min(y, 30MB/s) and min(z, 30MB/s)`

√ `min(x, 60MB/s) and min(y, 40MB/s) and min(z, 20MB/s)`

Find Solution using (Mixed) Integer Linear Programming

Decision variables:

- $x_e \in \{0, 1\}$ - whether an edge in the logical topology will be selected
- $r_{uv} \in [0, 1]$ - link utilization on link (u, v)
- $r_{max} \in [0, 1]$ - maximum link utilization
- $R_{max} \in [0, C]$ - maximum link rate

Flow conservation constraints:

$$\forall v \in \mathcal{G}, \sum_{e \in \delta^+(v)} x_e - \sum_{e \in \delta^-(v)} x_e = \begin{cases} 1 & \text{if } v = s_i \\ -1 & \text{if } v = t_i \\ 0 & \text{otherwise} \end{cases}$$

Resource constraints:

$$\forall (u, v), r_{uv} c_{uv} = \sum_i \sum_{e \in E_i(u,v)} r^i_{min} x_e \qquad \text{(definition of link utilization)}$$

$$\forall (u, v), r_{max} \geq r_{uv} \qquad \text{(definition of } r_{max}\text{)}$$

$$\forall (u, v), R_{max} \geq r_{uv} c_{uv} \qquad \text{(definition of } R_{max}\text{)}$$

$$r_{max} \leq 1 \qquad \text{(link capacity)}$$

Summary

- Merlin is an SDN programming language that enables programmers to specify resource constraints
- Merlin uses logical topology (cross product of topology and policy automaton) to find paths that satisfy waypoint traversal constraints
- Merlin transforms global constraints into localized constraints
- Merlin uses MILP to find feasible solution that satisfy the specified constraints

Propane

Overview

Don’t Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations

Ryan Beckett
Princeton

Ratul Mahajan
Microsoft

Todd Millstein
UCLA

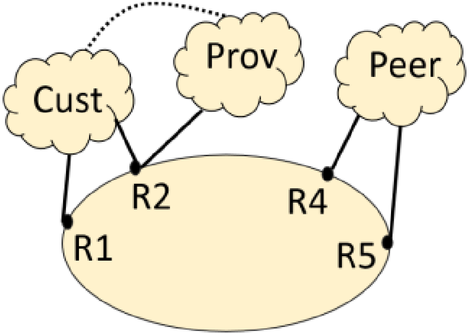
Jitendra Padhye
Microsoft

David Walker
Princeton

Ryan Beckett et al. “Don’t Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. New York, NY, USA: ACM, 2016, pp. 328–341. URL: http://doi.acm.org/10.1145/2934872.2934909

Problems

Complexity of using Decentralized Routing



Policy

P1. Prefer Cust > Peer > Prov

P2. Disallow traffic between Prov and Peer

P3. For Cust, prefer R1 > R2

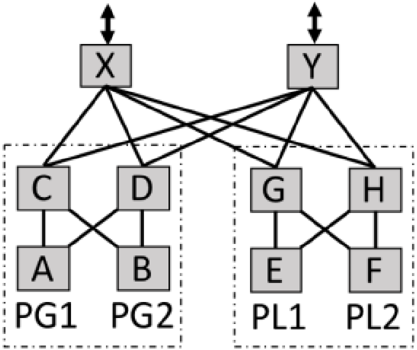
P4. Cust must be on path for its prefixes

P5. Cust must not be a transit to Prov

Figure 1: Creating router-level policies is difficult.

Problems

Complexity when Failures Happen



Policy

P1. Left cluster has global services with PG* prefixes, which should be announced externally as an aggregate PG

P2. Right cluster has local services with PL* prefixes, which should not be announced externally

Figure 2: Policy-compliance under failures is difficult.

Propane Example 1

```
define Prefs = exit(R1 » R2 » Peer » Prov)
```

(P1 and P3)

```
define Routing =
    {PCust => Prefs & end(Cust)
     true  => Prefs }
```

(P4, P1 and P3)

```
define transit(X,Y) = enter(X|Y) & exit(X|Y)
define cust-transit(X,Y) = later(X) & later(Y)
```

```
define NoTrans =
  {true => !transit(Peer,Prov) &
           !cust-transit(Cust,Prov)}
```

(P2, P5)

Final policy: `Routing & NoTrans`

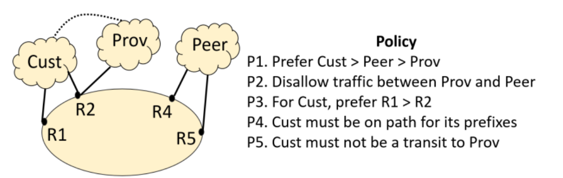


Figure 1: Creating router-level policies is difficult.

Propane Example 2

```
define Ownership =
    {PG1 => end(A)
     PG2 => end(B)
     PL1 => end(E)
     PL2 => end(F)
     true => end(out)}
```

```
define Locality =
    {PL1 | PL2 => only(in)}
```

Final policy: `Ownership & Locality`

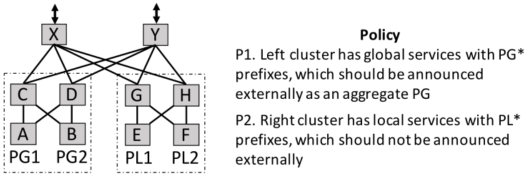


Figure 2: Policy-compliance under failures is difficult.

Propane Language

Syntax

pol	$::=$	$p_1, \ldots, p_n$	*policies*
p	$::=$	t => r_1» … »r_m $\mid$ cc	*constraints*
x	$::=$	$d.d.d.d/d$	*prefix*
t	$::=$	`true`	*true*
	$\mid$	$!t$	*negation*
	$\mid$	$t_1 \mid t_2$	*disjunction*
	$\mid$	$t_1 \& t_2$	*conjunction*
	$\mid$	$prefix = x$	*prefix test*
	$\mid$	$comm = d$	*community test*
r	$::=$	l	*location*
	$\mid$	$\emptyset$	*empty set*
	$\mid$	**in**	*internal loc*
	$\mid$	**out**	*external loc*
	$\mid$	$r_1 \cup r_2$	*union*
	$\mid$	$r_1 \cap r_2$	*intersection*
	$\mid$	$r_1 \cdot r_2$	*concatenation*
	$\mid$	$!r$	*path negation*
	$\mid$	r^*	*iteration*
ln	$::=$	$r_1 \rightarrow r_2$	*links*
cc	$::=$	$agg(x, ln) \mid tag(d, t, ln)$	*control constraints*

Propane Expansions

any	=	$\mathtt{out}^* \cdot \mathtt{in}^+ \cdot \mathtt{out}^*$
drop	=	$\emptyset$
internal	=	$\mathtt{in}^+$
only(X)	=	$\mathtt{any} \cap X^*$
never(X)	=	$\mathtt{any} \cap (!X)^*$
through(X)	=	$\mathtt{out}^* \cdot \mathtt{in}^* \cdot X \cdot \mathtt{in}^* \cdot \mathtt{out}^*$
later(X)	=	$\mathtt{out}^* \cdot (X \cap \mathtt{out}) \cdot \mathtt{out}^* \cdot \mathtt{in}^+ \cdot \mathtt{out}^*$
before(X)	=	$\mathtt{out}^* \cdot \mathtt{in}^+ \cdot \mathtt{out}^* \cdot (X \cap \mathtt{out}) \cdot \mathtt{out}^*$
end(X)	=	$\mathtt{any} \cap (\Sigma^* \cdot X)$
start(X)	=	$\mathtt{any} \cap (X \cdot \Sigma^*)$
exit(X)	=	$(\mathtt{out}^* \cdot \mathtt{in}^* \cdot (X \cap \mathtt{in}) \cdot \mathtt{out} \cdot \mathtt{out}^*) \cup$ $(\mathtt{out}^* \cdot \mathtt{in}^+ \cdot (X \cap \mathtt{out}) \cdot \mathtt{out}^*)$
enter(X)	=	$(\mathtt{out}^* \cdot \mathtt{out} \cdot (X \cap \mathtt{in}) \cdot \mathtt{in}^* \cdot \mathtt{out}^*) \cup$ $(\mathtt{out}^* \cdot (X \cap \mathtt{out}) \cdot \mathtt{in}^+ \cdot \mathtt{out}^*)$
link(X, Y)	=	$\mathtt{any} \cap (\Sigma^* \cdot X \cdot Y \cdot \Sigma^*)$
path$(\vec{X})$	=	$\mathtt{any} \cap (\Sigma^* \cdot X_1 \ldots X_n \cdot \Sigma^*)$
novalley$(\vec{X})$	=	$\mathtt{any} \cap$ $!\mathtt{path}(X_2, X_1, X_2) \cap \cdots \cap$ $!\mathtt{path}(X_n, X_{n-1}, X_n)$

Figure 4: Regular Intermediate Representation (RIR) syntax (left), and Propane language expansions (right).

Propane Compilation

We focus on the translations from FE to RIR and from RIR to PGIR

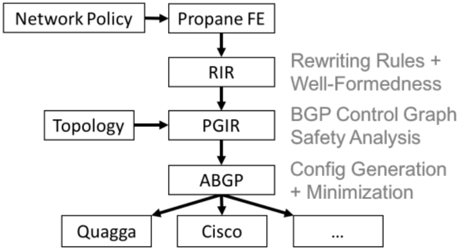


Figure 3: Compilation pipeline stages for Propane.

Regular Intermediate Representation (RIR)

- Each rule has the format `prefix => path`
- `prefix` is a set of IPv4 prefixes
- `path` denotes a set of path that satisfies the expression
- Policy composition is similar to OpenFlow policy composition
- After rule-based rewriting, the policy can be rewritten as one policy where each rule specifies the path for a single prefix

Example:

```
define Ownership =
    {PG1  => end(A)
     PG2  => end(B)
     PL1  => end(E)
     PL2  => end(F)
     true  => end(out)}
```

```
define Locality =
    {PL1 | PL2 => only(in)}
```

`Ownership & Locality`:

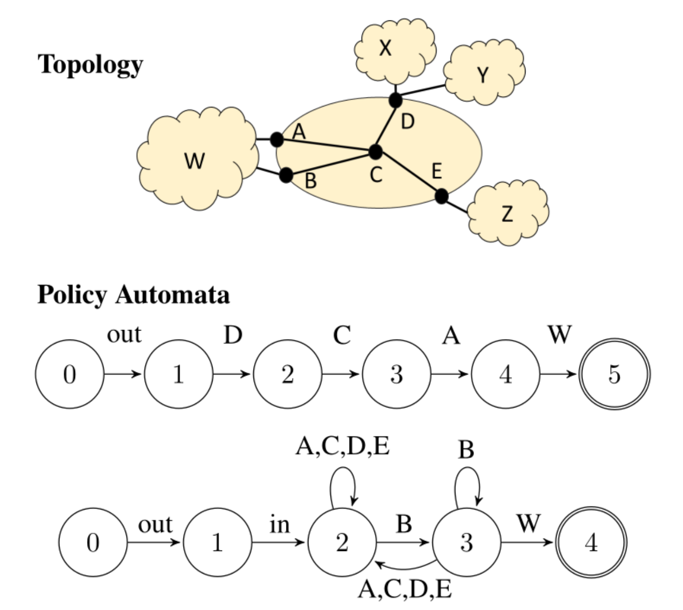
```
PG1  => end(A)
PG2  => end(B)
PL1  => only(in) ∩ end(E)
PL2  => only(in) ∩ end(F)
true  => exit(out)
```

Path Expression to Automaton

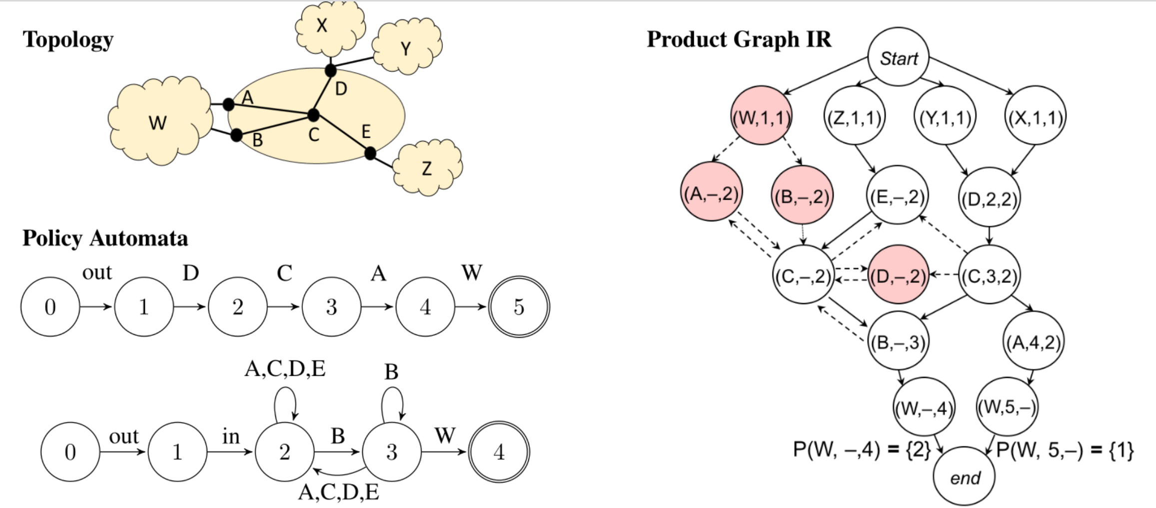
Propane path expression is a regular language and can be represented as a finite state automaton

Example:

- $\mathtt{W} \cdot \mathtt{A} \cdot \mathtt{C} \cdot \mathtt{D} \cdot \mathbf{out}$
- $\mathtt{W} \cdot \mathtt{B} \cdot \mathbf{in}^{+} \cdot \mathbf{out}$

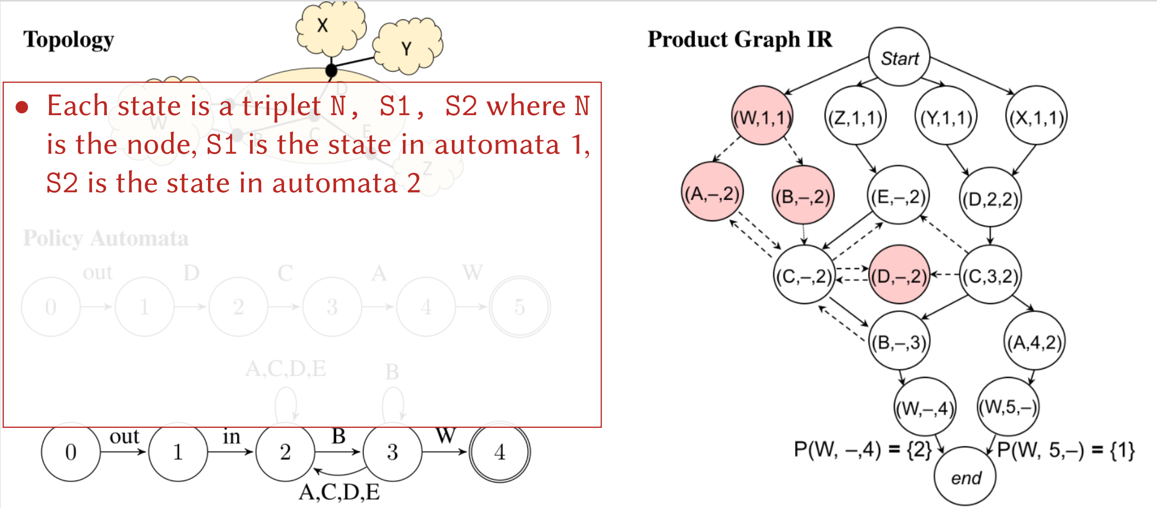


Product Graph IR



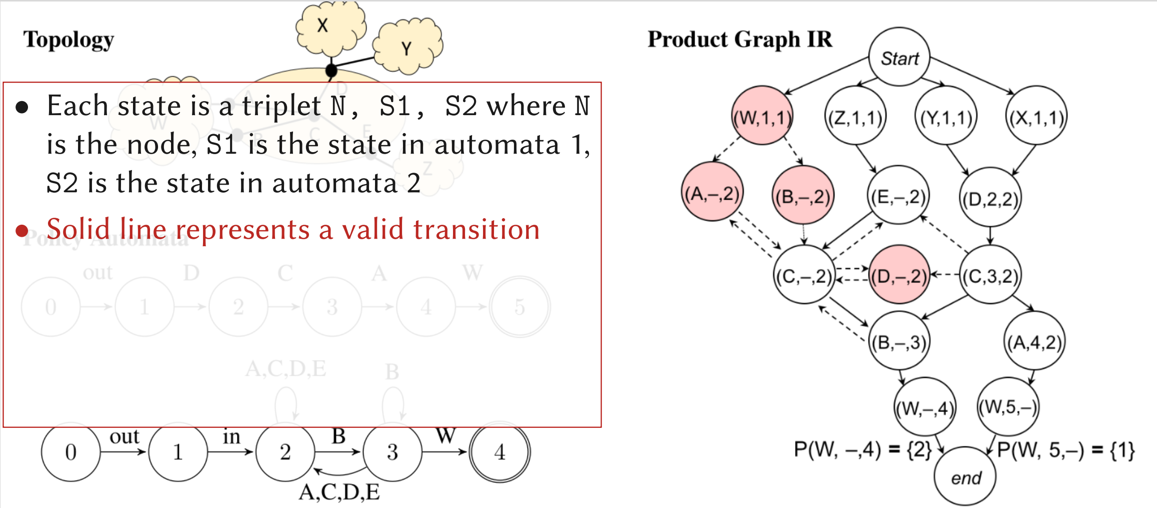
Product Graph IR

- Each state is a triplet `N`, `S1`, `S2` where `N` is the node, `S1` is the state in automata 1, `S2` is the state in automata 2



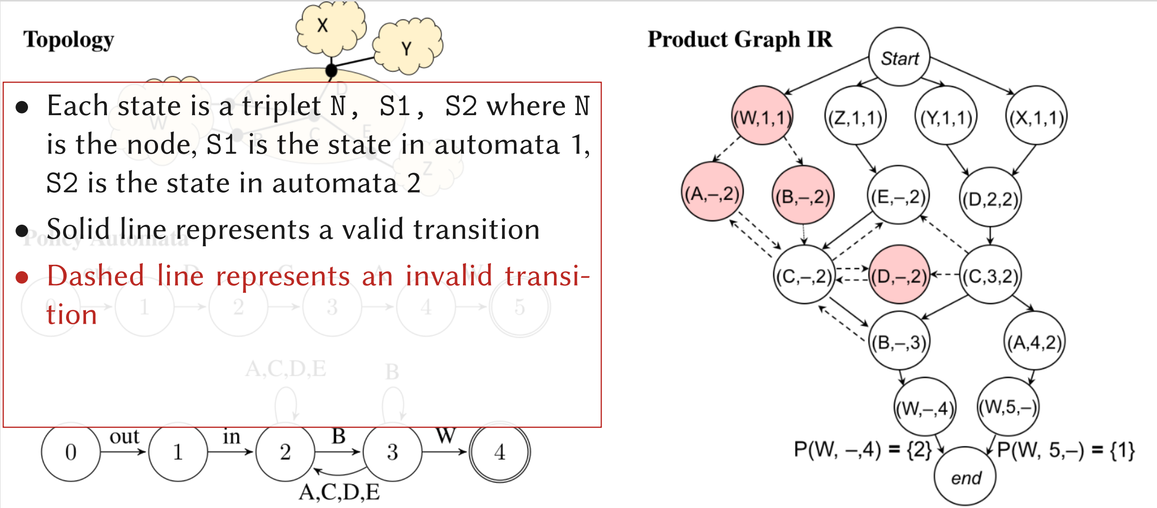
Product Graph IR

- Each state is a triplet N, S1, S2 where N is the node, S1 is the state in automata 1, S2 is the state in automata 2
- Solid line represents a valid transition



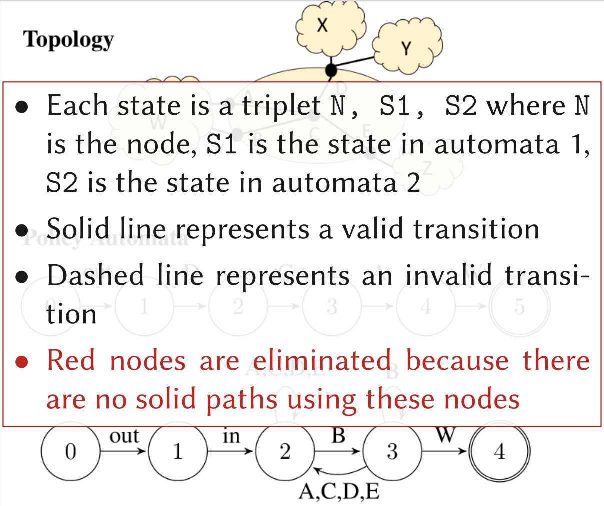
Product Graph IR

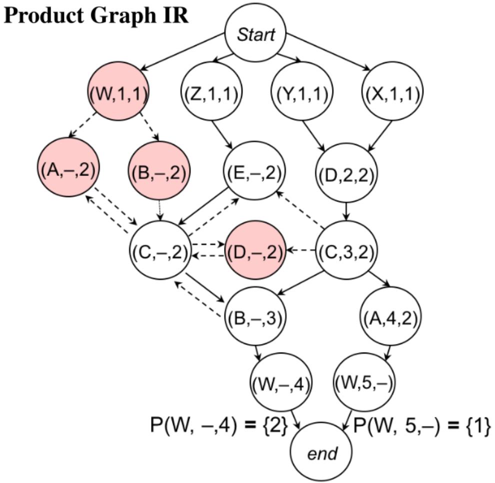
- Each state is a triplet `N, S1, S2` where `N` is the node, `S1` is the state in automata 1, `S2` is the state in automata 2
- Solid line represents a valid transition
- Dashed line represents an invalid transition



Product Graph IR

- Each state is a triplet `N, S1, S2` where `N` is the node, `S1` is the state in automata 1, `S2` is the state in automata 2
- Solid line represents a valid transition
- Dashed line represents an invalid transition
- Red nodes are eliminated because there are no solid paths using these nodes





What's Next

After obtaining the PGIR, Propane conducts preference inference with analysis on:

- **failure-safety**: whether the policy can be realized under network failures
- **aggregation-safety**: whether sub-prefixes can be aggregated into a shorter prefix

Then PGIR is translated into abstract BGP configuration

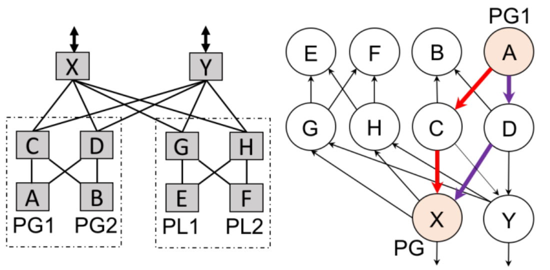


Figure 8: Aggregation safety for a datacenter.

Summary

- Propane is a programming language that allows programmers to specify centralized prefix routing requirements
- Propane policies are organized as well-formed regular intermediate representation (RIR)
- RIR is compiled with the topology to generate product graph intermediate representation (PGIR)
- Propane infers the local preferences with safety analysis, and translates PGIR into abstract BGP

SNAP

Overview

SNAP: Stateful Network-Wide Abstractions for Packet Processing

Mina Tahmasbi Arashloo[1], Yaron Koral[1], Michael Greenberg[2], Jennifer Rexford[1], and David Walker[1]

[1]Princeton University , [2]Pomona College

Mina Tahmasbi Arashloo et al. “SNAP: Stateful Network-Wide Abstractions for Packet Processing”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 29–43. URL: https://doi.org/10.1145/2934872.2934892

Stateful Network Policies

Example: DNS tunnel detection

- Keep a counter for resolved but not used DNS requests for each host
- If a counter exceeds a threshold, block the traffic from the host

DNS-tunnel-detect

```
if dstip = 10.0.6.0/24 & srcport = 53 then
  orphan[dstip][dns.rdata] <- True;
  susp-client[dstip]++;
  if susp-client[dstip] = threshold then
    blacklist[dstip] <- True
  else id
else
  if srcip = 10.0.6.0/24 & orphan[srcip][dstip]
  then orphan[srcip][dstip] <- False;
       susp-client[srcip]--
  else id
```

Figure 1: SNAP implementation of DNS-tunnel-detect.

Stateful Network Policies

Example: DNS tunnel detection

- Keep a counter for resolved but not used DNS requests for each host
- If a counter exceeds a threshold, block the traffic from the host

```
DNS-tunnel-detect

if dstip = 10.0.6.0/24 & srcport = 53 then
  orphan[dstip][dns.rdata] <- True;
  susp-client[dstip]++;
  if susp-client[dstip] = threshold then
    blacklist[dstip] <- True
  else id
else
  if srcip = 10.0.6.0/24 & orphan[srcip][dstip]
  then orphan[srcip][dstip] <- False;
       susp-client[srcip]--
  else id
```

Figure 1: SNAP implementation of `DNS-tunnel-detect`.

- `orphan[dstip][dns.rdata]`: for each host (`dstip`), the request to `dns.rdata` is resolved but not used
- `susp-client[ip]`: the counter for host with `ip`

SNAP: Syntax

The syntax of SNAP:

- is based on the syntax of Pyretic
- with extensions to support stateful policies
- has no support for traversal order (one-big-switch abstraction)

$e \in \mathsf{Expr}$	$::=$	$v \mid f \mid \overrightarrow{e}$	
$x, y \in \mathsf{Pred}$	$::=$	id	Identity
	$\mid$	$drop$	Drop
	$\mid$	$f = v$	Test
	$\mid$	$\neg x$	Negation
	$\mid$	$x \vert y$	Disjunction
	$\mid$	$y \& x$	Conjunction
	$\mid$	$s[e] = e$	**State Test**
$p, q \in \mathsf{Pol}$	$::=$	x	Filter
	$\mid$	$f \leftarrow v$	Modification
	$\mid$	$p + q$	Parallel comp.
	$\mid$	$p; q$	Sequential comp.
	$\mid$	$s[e] \leftarrow e$	**State Modification**
	$\mid$	$s[e]$++	**Increment value**
	$\mid$	$s[e]$--	**Decrement value**
	$\mid$	if a then p else q	**Conditional**
	$\mid$	atomic(p)	**Atomic blocks**

Figure 4: SNAP's syntax. Highlighted items are not in NetCore.

SNAP: Compilation

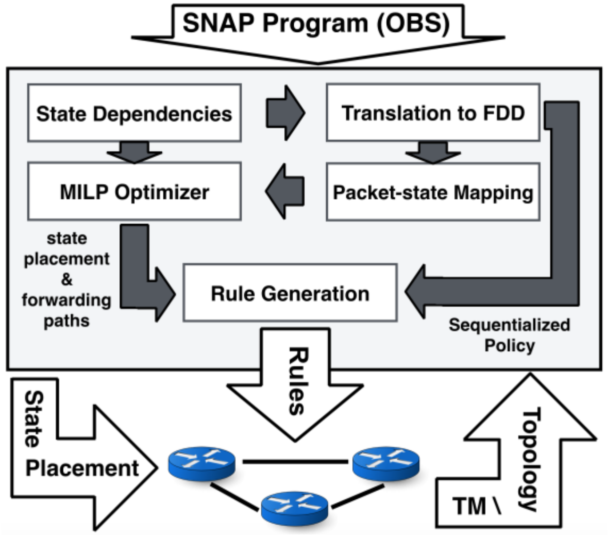


Figure 5: Overview of the compiler phases.

Main process:

- Translate SNAP program into xFDD
- Use xFDD to derive placement constraints for state variables
- Build MILP problem to solve the state placement problem
- Apply state placement and routing

xFDD

Extended Forwarding Decision Diagram (xFDD) is similar to *binary decision diagram*

It captures the data dependencies in a network policy

```
DNS-tunnel-detect

if dstip = 10.0.6.0/24 & srcport = 53 then
  orphan[dstip][dns.rdata] <- True;
  susp-client[dstip]++;
  if susp-client[dstip] = threshold then
    blacklist[dstip] <- True
  else id
else
  if srcip = 10.0.6.0/24 & orphan[srcip][dstip]
  then orphan[srcip][dstip] <- False;
      susp-client[srcip]--
  else id
```

Figure 1: SNAP implementation of DNS-tunnel-detect.

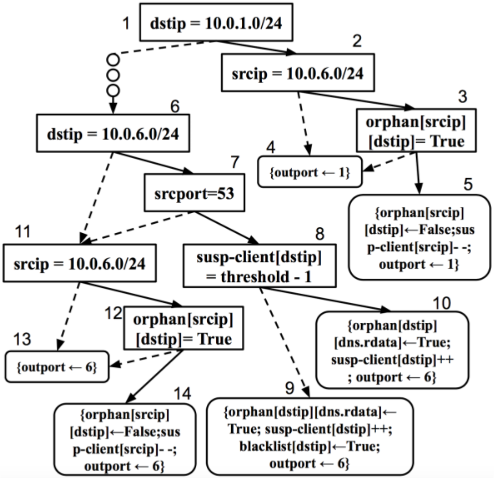


Figure 3: The equivalent xFDD for DNS-tunnel-detect; assign-egress

xFDD Translation

SNAP uses syntax-driven translation

$$
\begin{array}{rcll}
d & ::= & t \mathrel{?} d_1 : d_2 \mid \{as_1, \ldots, as_n\} & \text{xFDDs} \\
t & ::= & f = v \mid f_1 = f_2 \mid s[e_1] = e_2 & \text{tests} \\
as & ::= & a \mid a; a & \text{action sequences} \\
a & ::= & id \mid drop \mid f \leftarrow v \mid s[e_1] \leftarrow e_2 & \text{actions} \\
 & & \mid s[e_1]\texttt{++} \mid s[e_1]\texttt{--} &
\end{array}
$$

$$
\begin{array}{rcl}
\textsc{to-xfdd}(a) & = & \{a\} \\
\textsc{to-xfdd}(f = v) & = & f = v \mathrel{?} \{id\} : \{drop\} \\
\textsc{to-xfdd}(\neg x) & = & \ominus\textsc{to-xfdd}(x) \\
\textsc{to-xfdd}(s[e_1] = e_2) & = & s[e_1] = e_2 \mathrel{?} \{id\} : \{drop\} \\
\textsc{to-xfdd}(\mathsf{atomic}(p)) & = & \textsc{to-xfdd}(p) \\
\textsc{to-xfdd}(p + q) & = & \textsc{to-xfdd}(p) \oplus \textsc{to-xfdd}(q) \\
\textsc{to-xfdd}(p; q) & = & \textsc{to-xfdd}(p) \odot \textsc{to-xfdd}(q) \\
\textsc{to-xfdd}(\text{if } x \text{ then } p \text{ else } q) & = & (\textsc{to-xfdd}(x) \odot \textsc{to-xfdd}(p)) \\
 & & \oplus\ (\ominus\textsc{to-xfdd}(x) \odot \textsc{to-xfdd}(q))
\end{array}
$$

Figure 6: xFDD syntax and translation.

NOTE: Policy compositions are translated to xFDD compositions

xFDD Composition

$$\{as_{11}, \cdots, as_{1n}\} \oplus \{as_{21}, \cdots, as_{2m}\} = \{as_{11}, \cdots, as_{1n}\} \cup \{as_{21}, \cdots, as_{2m}\}$$

$$(t\ ?\ d_1 : d_2) \oplus \{as_1, \cdots, as_n\} = (t\ ?\ d_1 \oplus \{as_1, \cdots, as_n\} : d_2 \oplus \{as_1, \cdots, as_n\})$$

$$(t_1\ ?\ d_{11} : d_{12}) \oplus (t_2\ ?\ d_{21} : d_{22}) = \begin{cases} (t_1\ ?\ d_{11} \oplus d_{21} : d_{12} \oplus d_{22}) & t_1 = t_2 \\ (t_1\ ?\ d_{11} \oplus (t_2\ ?\ d_{21} : d_{22}) : d_{12} \oplus (t_2\ ?\ d_{21} : d_{22})) & t_1 \sqsubset t_2 \\ (t_2\ ?\ d_{21} \oplus (t_1\ ?\ d_{11} : d_{12}) : d_{22} \oplus (t_1\ ?\ d_{11} : d_{12})) & t_2 \sqsubset t_1 \end{cases}$$

$$\ominus\{id\} = \{drop\}$$
$$\ominus\{drop\} = \{id\}$$
$$\ominus(t?d_1 : d_2) = (t? \ominus d_1 : \ominus d_2)$$

$$\begin{aligned} as \odot \{as_1, \cdots, as_n\} &= \{as \odot as_1, \cdots, as \odot as_n\} \\ as \odot (t\ ?\ d_1 : d_2) &= \text{(see explanations in §4.2)} \\ \{as_1, \cdots, as_n\} \odot d &= (as_1 \odot d) \oplus \cdots \oplus (as_n \odot d) \\ (t\ ?\ d_1 : d_2) \odot d &= (d_1 \odot d)|_t \oplus (d_2 \odot d)|_{\sim t} \end{aligned}$$

$$\{as_1, \cdots, as_n\}|_t = (t\ ?\ \{as_1, \cdots, as_n\} : \{drop\})$$

$$(t_1\ ?\ d_1 : d_2)|_{t_2} = \begin{cases} (t_1\ ?\ d_1 : \{drop\}) & t_1 = t_2 \\ (t_2\ ?\ (t_1\ ?\ d_1 : d_2) : \{drop\}) & t_2 \sqsubset t_1 \\ (t_1\ ?\ d_1|_{t_2} : d_2|_{t_2}) & t_1 \sqsubset t_2 \end{cases}$$

Figure 7: Definitions of xFDD composition operators.

State Placement and Routing Problem

SNAP places the program for one-big-switch in the network

DNS-tunnel-detect

```
if dstip = 10.0.6.0/24 & srcport = 53 then
  orphan[dstip][dns.rdata] <- True;
  susp-client[dstip]++;
  if susp-client[dstip] = threshold then
    blacklist[dstip] <- True
  else id
else
  if srcip = 10.0.6.0/24 & orphan[srcip][dstip]
  then orphan[srcip][dstip] <- False;
       susp-client[srcip]--
  else id
```

Figure 1: SNAP implementation of DNS-tunnel-detect.

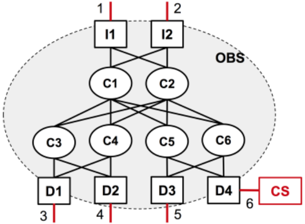


Figure 2: Topology for the running example.

This is achieved by constructing an MILP problem

Problem Formulation

Variables and Constants

Decision variables

- $R_{uvij} \in \{0, 1\}$: flow from u to v that traverses link (i, j)
- $P_{sn} \in \{0, 1\}$: state variable s is placed on node n
- $P_{suvij} \in \{0, 1\}$: flow from u to v has traversed state variable s before traversing link (i, j)

Constants

- d_{uv}: traffic from u to v
- c_{ij}: link capacity for link (i, j)
- S_{uv}: state variables that flow from u to v depends on
- state dependencies: $(s, t) \in \mathit{tied}$ - state variables s and t must be put on the same device
- state dependencies: $(s, t) \in \mathit{dep}$ - state variables s must be visited before t

Problem Formulation

Routing Constraints

$$\sum_j R_{uvuj} = 1 \qquad \forall u, v \quad \text{(source)}$$

$$\sum_i R_{uviv} = 1 \qquad \forall u, v \quad \text{(sink)}$$

$$\sum_{u,v} R_{uvij} d_{uv} \leq c_{ij} \qquad \forall i, j \quad \text{(link capacity)}$$

$$\sum_i R_{uvin} = \sum_j R_{uvnj} \qquad \forall u, v, n \quad \text{(flow conservation)}$$

$$\sum_i R_{uvin} \leq 1 \qquad \forall u, v, n \quad \text{(single path)}$$

Problem Formulation

State Constraints

State constraints

$$\sum_n P_{sn} = 1 \qquad \forall s \quad \text{(visited once)}$$

$$\sum_i R_{uvin} \geq P_{sn} \qquad \forall u, v, \forall s \in S_{uv} \quad \text{(only placed on path)}$$

$$P_{sn} = P_{tn} \qquad \forall (s, t) \in \mathit{tied} \quad \text{(state placement constraint)}$$

$$P_{suvij} \leq R_{uvij} \qquad \forall u, v, s \in S_{uv} \quad \text{(only traversed on path)}$$

$$P_{sn} + \sum_i P_{suvin} = \sum_j P_{suvij} \qquad \forall u, v, s, \in S_{uv} \quad \text{(state traversal)}$$

$$P_{sv} + \sum_i P_{suviv} = 1 \qquad \forall u, v, s \in S_{uv} \quad \text{(state sink)}$$

$$P_{sn} + \sum_i P_{suvin} \geq P_{tn} \qquad \forall u, v, (s, t) \in \mathit{dep} \quad \text{(state dependency)}$$

Summary

- SNAP enables stateful network policies
- SNAP translates one-big-switch policies into xFDD to analyze the state dependencies
- The routing and state placement are modeled as an MILP problem

The End

Summary

In the coming lectures, we cover the following topics on network control programming

- SDN-based policy routing
- distributed enforcement of centralized policy
- programming stateful networks

In this lecture, you should

- learn network programming languages Merlin, Propane and SNAP
- learn path constraints expressed in regular expressions
- learn how to use product graph to find policy compliant paths
- learn how to use ILP to solve state placement problem

Thanks!

kaigao@scu.edu.cn

References I

[1] Mina Tahmasbi Arashloo et al. "SNAP: Stateful Network-Wide Abstractions for Packet Processing". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 29–43. URL: https://doi.org/10.1145/2934872.2934892.

[2] Ryan Beckett et al. "Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 328–341. URL: http://doi.acm.org/10.1145/2934872.2934909.

[3] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76.

[4] Robert Soulé et al. "Merlin: A Language for Provisioning Network Resources". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 213–226. URL: http://doi.acm.org/10.1145/2674005.2674989.