SNAP: Stateful Network-Wide Abstractions for Packet Processing

Mina Tahmasbi Arashloo¹, Yaron Koral¹, Michael Greenberg², Jennifer Rexford¹, and David Walker¹

¹ Princeton University, ² Pomona College
Software Defined Networks (SDN) - Centralized Control
Software Defined Networks (SDN) - Centralized Control
Software Defined Networks (SDN) - Centralized Control
Software Defined Networks (SDN) - Centralized Control

Program your network from a central logical point!
OpenFlow - Abstractions for SDN

<table>
<thead>
<tr>
<th>Prio</th>
<th>match</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing a network device with a brain symbol and OpenFlow table]

4
OpenFlow - Abstractions for SDN

<table>
<thead>
<tr>
<th>Prio</th>
<th>match</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dstip = 10.0.0.1</td>
<td>outport ← 1</td>
</tr>
<tr>
<td>2</td>
<td>dstip = 10.0.0.2</td>
<td>drop</td>
</tr>
</tbody>
</table>
OpenFlow - Abstractions for SDN

Each Rule can
• Match on header fields
• modify/forward/drop packets

<table>
<thead>
<tr>
<th>Prio</th>
<th>match</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dstip = 10.0.0.1</td>
<td>outport ← 1</td>
</tr>
<tr>
<td>2</td>
<td>dstip = 10.0.0.2</td>
<td>drop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Is OpenFlow Enough?

- OpenFlow rules are “stateless”
  - Rule tables process each packet independently from the rest
- Algorithms almost always need “stateful” processing
  - i.e., decide what to do with the packet based on packets seen so far!
Option #1 - All the state on the controller
Option #1 - All the state on the controller
Option #1 - All the state on the controller
Option #1 - All the state on the controller
Option #1 - All the state on the controller

Centralized control **but** not efficient!

- Switches process packets at **ns** scale
- Going through the controller, each update could take from **ms** to **a few seconds**
Option #2 - Middleboxes (MBs)

- Use dedicated **blackboxes** for **each functionality** alongside with switches
Option #2 - Middleboxes (MBs)

Efficient **but** we lose centralized control!

- MBs are ad-hoc blackboxes
- They make it hard to reason about network’s behavior
Our Goal

Stateful packet processing

with centralized control

without compromising on efficiency
Insight

- New switches offer more sophisticated **stateful** packet processing functionality
  - The switch has **local state**
  - Rules can match on/modify local state
Let’s push stateful processing to switches!
Let’s push stateful processing to switches!
• The stateful program is written on top of one big switch
• The stateful program is written on top of one big switch

• The actual network has collections of switches
• The stateful program is written on top of one big switch

• The actual network has collections of switches

• How should we realize the program collectively on the network of switches?
SNAP - Language and Compiler Overview

Intermediate Representation (FDD)

Distributed version of the program’s FDD

“Stateful” Rules
SNAP - Language
Packets!

Diagram:

```
  srcip | dstip | srcport | ...
```
Programming Model

- SNAP’s expressions are **functions**

  current state

  input packet

  updated state

  set of packets
Programming Model

• SNAP’s expressions are functions

current state
input packet
updated state
set of packets
Programming Model

• SNAP’s expressions are **functions**

current state

updated state

input packet

set of packets
Programming Model

- SNAP’s expressions are \textbf{functions}

current state

input packet

\textbf{updated state}

\textbf{set of packets}

Reads/Modifies state
Reads/Duplicate/Modifies packet
Programming Model

- SNAP’s expressions are **functions**

  - **current state**
  - **updated state**
  - **input packet**
  - **set of packets**

  Reads/Modifies state
  Reads/Duplicate/Modifies packet
Running Example - Detecting Malicious Domains

- Domains that change TTL frequently are suspected to be malicious
Running Example - Detecting Malicious Domains

- Domains that change TTL frequently are suspected to be malicious

IP address of www.google.com?
Running Example - Detecting Malicious Domains

• Domains that change TTL frequently are suspected to be malicious

  domain: www.google.com
  IP: 74.125.224.72
  TTL (valid for): 1 day
TTL Change Tracking in SNAP

```python
if dstip = CS_ip & srcport = DNS then
    if ~seen[dns.domain] then
        seen[dns.domain] ← True;
        last_ttl[dns.domain] ← dns.ttl;
        ttl_change[dns.domain] ← 0
    else
        if dns.ttl = last_ttl[dns.domain] then
            id
        else
            last_ttl[dns.domain] ← dns.ttl;
            ttl_change[dns.domain]++
    else id
```
TTL Change Tracking in SNAP

```python
if dstip = CS_ip & srcport = DNS then
  if ~seen[dns.domain] then
    seen[dns.domain] ← True;
    last_ttl[dns.domain] ← dns.ttl;
    ttl_change[dns.domain] ← 0
  else
    if dns.ttl = last_ttl[dns.domain] then
      id
    else
      last_ttl[dns.domain] ← dns.ttl;
      ttl_change[dns.domain]++
  else id
```
TTL Change Tracking in SNAP

```python
if dstip = CS_ip & srcport = DNS then
    if ~seen[dns.domain] then
        seen[dns.domain] ← True;
        last_ttl[dns.domain] ← dns.ttl;
        ttl_change[dns.domain] ← 0
    else
        if dns.ttl = last_ttl[dns.domain] then
            id
        else
            last_ttl[dns.domain] ← dns.ttl;
            ttl_change[dns.domain]++
        else id
```

State variable is a key-value dictionary
TTL Change Tracking in SNAP

```python
if dstip = CS_ip & srcport = DNS then
    if ~seen[dns.domain] then
        seen[dns.domain] ← True;
        last_ttl[dns.domain] ← dns.ttl;
        ttl_change[dns.domain] ← 0
    else
        if dns.ttl = last_ttl[dns.domain] then id
        else
            last_ttl[dns.domain] ← dns.ttl;
            ttl_change[dns.domain]++
        else id
```
TTL Change Tracking in SNAP

```python
if dstip = CS_ip & srcport = DNS then
    if ~seen[dns.domain] then
        seen[dns.domain] ← True;
        last_ttl[dns.domain] ← dns.ttl;
        ttl_change[dns.domain] ← 0
    else
        if dns.ttl = last_ttl[dns.domain] then
            id
        else
            last_ttl[dns.domain] ← dns.ttl;
            ttl_change[dns.domain]++
        else id
```
Adding Forwarding

- Operator wants to specify where packets should be forwarded to

```plaintext
forwarding = if dstip = CS_ip then outport ← CS
            else if dstip = EE_ip then outport ← EE
            else if dstip = ISP1_ip then outport ← ISP1
            else if dstip = ISP2_ip then outport ← ISP2
            else drop
```

- Forwarding is composed with TTL change tracking

```plaintext
ttl_change ; forwarding
```
SNAP Compiler

- Identify State Dependencies
- Translate to Intermediate Representation (FDD)
- Identify mapping from packets to state variables
- Optimally distribute the FDD
- Generate rules per switch
SNAP Compiler

- Identify State Dependencies
- Translate to Intermediate Representation (FDD)
- Identify mapping from packets to state variables
- Optimally distribute the FDD
- Generate rules per switch

ttl_change → last_ttl → seen
SNAP Compiler

- Identify State Dependencies
  - ttl_change → last_ttl → seen

- Translate to Intermediate Representation (FDD)
  - ?

- Identify mapping from packets to state variables
  - ?

- Optimally distribute the FDD
  - ?

- Generate rules per switch
  - ?
Why Forwarding Decision Diagrams (FDDs)?

- Efficient
  - in terms of number of generated rules
  - for extraction of mapping from packets to state variables (next phase)
Forwarding Decision Diagrams (FDDs)

- Generalization of binary decision diagrams [1]
- Intermediate node: test on header fields and state
- Leaf: set of action sequences

Forwarding Decision Diagrams (FDDs)

- Three types of tests
  - field = value
  - field₁ = field₂
  - state_var[e₁] = e₂
Forwarding Decision Diagrams (FDDs)

- Three types of tests
  - field = value
  - field₁ = field₂
  - state_var[e₁] = e₂

\[
\begin{align*}
\text{dstip} &= 10.0.0.1 \\
\text{srcip} &= \text{dstip} \\
\text{s[srcip]} &= 2 \\
\{\text{s[dstip]} \leftarrow 2\} &\quad \{\text{drop}\}
\end{align*}
\]
Forwarding Decision Diagrams (FDDs)

- Three types of tests
  - \( field = value \)
  - \( field_1 = field_2 \)
  - \( state\_var[e_1] = e_2 \)
Forwarding Decision Diagrams (FDDs)

- Three types of tests
  - field = value
  - field₁ = field₂
  - state_var[e₁] = e₂
Forwarding Decision Diagrams (FDDs)

- Three types of tests
  - \( \text{field} = \text{value} \)
  - \( \text{field}_1 = \text{field}_2 \)
  - \( \text{state}_\text{var}[\text{e}_1] = \text{e}_2 \)
SNAP Expression to FDD

dstip = CS_ip
srcport = DNS
seen[dns.domain] = True

{outport ← EE}
{outport ← CS}

dstip = EE_ip

{drop}

seen[dns.domain] = True
last_ttl[dns.domain] = dns.ttl

{last_ttl[dns.domain] ← dns.ttl;
  ttl_change[dns.domain]++;
  outport ← CS}
{outport ← CS}

{seen[dns.domain] ← True;
  last_ttl[dns.domain] ← dns.ttl;
  ttl_change[dns.domain] ← 0;
  outport ← CS}
SNAP Expression to FDD

dstip = CS_ip

seen[dns.domain] = True

{outport ← EE}

dstip = EE_ip

srcport = DNS

seen[dns.domain] = True

{drop}

last_ttl[dns.domain] = dns.ttl

{outport ← CS}

{last_ttl[dns.domain] ← dns.ttl; ttl_change[dns.domain] ++; outport ← CS}

{seen[dns.domain] ← True; last_ttl[dns.domain] ← dns.ttl; ttl_change[dns.domain] ← 0; outport ← CS}
SNAP Compiler

Identify State Dependencies

Translate to Intermediate Representation (FDD)

Identify mapping from packets to state variables

Optimally distribute the FDD

Generate rules per switch

ttl_change → last_ttl → seen
SNAP Compiler

- Identify State Dependencies
- Translate to Intermediate Representation (FDD)
- Identify mapping from packets to state variables
- Optimally distribute the FDD
- Generate rules per switch

**ttl_change → last_ttl → seen**

flows to CS need all three state variables
SNAP Compiler

- Identify State Dependencies
  - $\text{ttl\_change} \rightarrow \text{last\_ttl} \rightarrow \text{seen}$

- Translate to Intermediate Representation (FDD)
  - ✔

- Identify mapping from packets to state variables
  - flows to CS need all three state variables

- Optimally distribute the FDD
  - ?

- Generate rules per switch
  - ?
Optimal Distribution of the FDD

optimizing network congestion

CS
Optimal Distribution of the FDD

optimizing network congestion
Optimal Distribution of the FDD

optimizing network congestion

CS
SNAP Compiler

- Identify State Dependencies
- Translate to Intermediate Representation (FDD)
- Identify mapping from packets to state variables
- Optimally distribute the FDD
- Generate rules per switch

- \(\text{ttl\_change} \rightarrow \text{last\_ttl} \rightarrow \text{seen}\)
- Flows to CS need all three state variables

?
SNAP Compiler

Identify State Dependencies

Translate to Intermediate Representation (FDD)

Identify mapping from packets to state variables

Optimally distribute the FDD

Generate rules per switch

ttl_change \rightarrow \text{last}_ttl \rightarrow \text{seen}

flows to CS need all three state variables

?
SNAP Compiler

- Identify State Dependencies
  - ttl_change → last_ttl → seen

- Translate to Intermediate Representation (FDD)
  - ✔

- Identify mapping from packets to state variables
  - ✔

- Optimally distribute the FDD
  - ✔

- Generate rules per switch
  - ✔
Putting It All Together

ISP1

ISP2

CS

EE

dstip = CS_ip

srcport = DNS

seen[dns.domain] = True

dstip = EE_ip

{outport ← EE}

{drop

{drop

seen[dns.domain] = True
Putting It All Together

- \(dstip = CS_{ip}\)
- \(srcport = DNS\)
- \(seen[dns.domain] = True\)
- \(dstip = EE_{ip}\)
- \(srcport = DNS\)
- \(seen[dns.domain] = True\)
Putting It All Together

ISP1 -> CS

dstip = CS_ip

srcport = DNS

seen[dns.domain] = True

{outport ← EE}

dstip = EE_ip

{drop}

seen[dns.domain] = True

{drop}
Putting It All Together

ISP1

<table>
<thead>
<tr>
<th>6</th>
</tr>
</thead>
</table>

ISP2

CS

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
</table>

EE

dstip = CS_ip

srcport = DNS

seen[dns.domain] = True

1

dstip = EE_ip

{outport ← EE}

4

6

{drop}
dstip = CS_ip

seen[dns.domain] = True

srcport = DNS

dstip = EE_ip

{drop}

{outport ← EE}
Putting It All Together

ISP1

ISP2

CS

EE

seen[dns.domain] = True

{seen[dns.domain] ← True; last_ttl[dns.domain] ← dns.ttl; ttl_change[dns.domain] ← 0; outport ← CS}

{outport ← CS}
Putting It All Together

ISP1

ISP2

CS

EE

\[
\text{seen[dns.domain]} = \text{True}
\]

{outport ← CS}

\[
\text{seen[dns.domain]} \leftarrow \text{True; last_ttl[dns.domain] ← dns.ttl; ttl_change[dns.domain] ← 0; outport ← CS}
\]
Putting It All Together

ISP1

ISP2

seen[dns.domain] = True

{seen[dns.domain] ← True; last_ttl[dns.domain] ← dns.ttl; ttl_change[dns.domain] ← 0; outport ← CS}

{outport ← CS}
Evaluation

• Evaluated on three campus networks and four ASs
  • 25-160 switches
  • 100-650 links

• Cold-start compilation takes 35-600 seconds
  • most of the time goes for optimally distributing the FDD

• Re-compilation time can be reduced to under one minute by fixing state placement
Related Work

• **NetKAT**
  - inspired basic language constructs

• **Fast NetKAT Compiler**
  - stateless FDDs

• **Stateful NetKAT** (largely concurrent with SNAP)
  - simple registers (vs general dictionaries)
  - formal definition and proof of correctness for updates
  - Different optimization goal (rule space)
Thanks!
Questions?
Atomic Blocks

• We assume that state reads/writes in a single switch happen *atomically*.

• If the programmer puts a part of the program in the **atomic** block, all the state variables in the block end up on the same switch.
Distributing a State Variable

- We can partition a state variable into disjoint parts and place the partitions on different switches.
- State variable $s$ from IP addresses to $X$ can be partitioned to state variables $s_i$ from a subset of IP addresses $IP_i$
  - $IP_i$'s are disjoint
  - Each $s_i$ can be placed on a separate switch