CS434/534: Topics in Network Systems

Network Verification:
Anteater, HSA, Veriflow, APV

Yang (Richard) Yang
Computer Science Department
Yale University
208A Watson
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Outline

- Admin and recap
- Network modeling and verification
  - Overview
  - Data plane (DP) based modeling and verification
    - Anteater (network verification as Boolean logic SAT)
    - Header space analysis (HSA; symbolic, set theoretical model and verification of networks)
    - Real-time DP analysis
      - Real-time HSA
      - VeriFlow
      - Atomic predicate verifier
Admin

- Instructor office hours
  - Tuesday: 1:30-2:30
  - Thursday: 3:00-4:00
  - Fridays: 1:30-2:30 pm

- Projects
  - Milestones (exactly 4 weeks left)
    - 4/17 (T+1 week): Finish reading major related work; a google doc listing related papers (at least 4 papers)
    - 4/24 (T+2 weeks): Finish architecture design (slides/write up of architecture, including all key components)
    - 5/1 (T+3 weeks): Initial, preliminary evaluations (slides/write up, about experiment/analysis setup)
    - 5/8 (T+4 weeks) 5:30 pm, final report due
Recap: NF/Middlebox Programming Model: Click

- A large number of small elements, each performing a simple packet function
  - queues as explicit elements as sources or sinks
- Elements connected together in a directed graph
  - elements interact through push or pull
Recap: NF/Middlebox Programming Model: ClickNP

- Basic idea: programmers write Click programs, system compiles to joint CPU/FPGA packet processing.
Recap: NF/Middlebox Programming Model: Bro

- Speed match/avoid blocking
  - Three-level hierarchy
    - Event Engine and Policy Script Interpreter connect through event queue, to allow asynchronously policy interpretation of expensive policies

- Stream based processing
  - Predefined TCP_Connection as a base for others to extend
Recap: NF/Middlebox Programming: NetBricks

- High-level NF programming abstractions

<table>
<thead>
<tr>
<th>Packet Processing</th>
<th>Control Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse/Deparse</td>
<td>Group By</td>
</tr>
<tr>
<td>Transform</td>
<td>Shuffle</td>
</tr>
<tr>
<td>Filter</td>
<td>Merge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte Stream</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td>Bounded</td>
</tr>
<tr>
<td>Packetize</td>
<td>Consistency</td>
</tr>
</tbody>
</table>

- Language (type system) based isolation
Recap: NF/Middlebox Orchestration

- **Placement** (which NF runs where)
  - Elastic scaling (adapting the number of NF instances and balancing load across them)

- **Service composition and isolation**
  - Composition ambiguity, composition under dynamicity

- **Resource scheduling and isolation**
  - Scheduling considering both network and NF load balancing/constraints

- **Fault-tolerance**

- **Energy management**

- **Monitoring**
Question to Think About

- Current network:

- What does the network look like in next-generation (5G) w/ SDN+NFV support?
Outline

- Admin and recap
- Network modeling and verification
Setting

- Modern networks are increasingly more complex
- Outages increasingly more costly

Microsoft: misconfigured network device led to Azure outage

Massive route leak causes Internet slowdown

Google Compute Engine Incident #16015
Networking issue with Google Compute Engine services
Incident began at 2016-08-05 00:54 and ended at 2016-08-05 02:40 (all times are US/Pacific).

Router Crashes Trigger Major Southwest IT System Failure
By: Chris Preimesberger | July 21, 2016
Simple Example

- Previously, an intrusion detection and prevention (IDP) device inspected all traffic to/from dorms
- IDP couldn’t handle load; added bypass
- IDP only inspected traffic between dorm and campus

Q: How do you know if it is acceptable?
Discussion

- How may you verify whether the behavior of a network (system) is acceptable?
Basic Correctness Properties

- **Reachability**: two entities reach each other if and only allowed by policy
  - No want reach but does not reach
    - No loop
    - No blackhole
  - No want no reach but reach
    - No leakage among tenants

- **Constraints**
  - Reachability only if pass a specified sequence of middleboxes

- **Consistency**
  - Two nodes (e.g., mirrors) have exactly the same behavior
Basic Types of Approaches

Control Plane

- Batfish
- Minesweeper
- ...

Data Plane

executed

- Anteater
- HSA
- Veriflow
- Atomic Predicates
- ...

Control plane verification

Data plane verification
Data Plane Verification: Setting

- **Input:** Each device has already computed its data plane (flow table)

- **Output:** Verify that the network’s behavior satisfy given correctness properties

<table>
<thead>
<tr>
<th>Destination</th>
<th>Iface</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1.1.0/24 &amp; port = 80</td>
<td>Drop</td>
</tr>
<tr>
<td>10.1.1.128/25</td>
<td>v'</td>
</tr>
<tr>
<td>10.1.1.0/24</td>
<td>v</td>
</tr>
<tr>
<td>10.1.1.2</td>
<td>v</td>
</tr>
</tbody>
</table>
Network Verification can be Hard

Correctness (reachability): s cannot reach t

Network mechanism: assume flow table allows one-bit packet filters, e.g.,

“if $p[43] = 0$ then drop”

Then: verify reachability is NP-Hard

$p[4] = 1$

$p[7] = 1$

$p[1] = 0$

$\left( x_4 \_ x_7 \_ x_1 \right)^\wedge (\ldots)^\wedge (\ldots)^\wedge (\ldots)$
Look Ahead

- Despite theoretical hardness, much success in practical systems
Outline

- Admin and recap
- Network functions (carrier cloud)
- Network modeling and verification
  - Overview
  - Data plane (DP) based modeling and verification
    - Anteater
Among the first to apply formal method (Boolean satisfiability; SAT) to network verification

- Model datapath correctness requirements as Boolean expressions
- Verify network properties by computing Boolean Satisfiability (e.g., use SAT tool such as Boolector)
Input

- Given a Boolean expression \( P(x_1, x_2, \ldots, x_n) \), where each \( x_i \) is a Boolean variable (true, false)

Goal: find an assignment to \( x_i \) to make \( P(x_1, x_2, \ldots, x_n) \) true, e.g.,

- \( P(x, y) = x \lor y \)
- \( P(x, y) = x \land \neg x \)

Ex: what if you want to verify that \( P(x_1, x_2, \ldots, x_n) \) is always true

Substantial progress in SAT solvers, typically based on an approach called “systematic search.”
"Systematic" SAT Search

- The search space is a tree
  - each vertex representing a Boolean variable
  - out edges representing the two choices (true and false) for this variable.

- The DPLL/Davis-Putnam-Logemann-Loveland algorithm
  - based on 3 operations: decide, propagate, and backtrack.
  - benefits from a restricted representation of formulas in conjunctive normal form, or CNF, e.g., \( \neg p \land (p \lor q) \)
Define \( P(u, v) \) as the policy function for packets traveling from \( u \) to \( v \)

- A packet can flow over \( (u, v) \) if and only if it satisfies \( P(u, v) \)

\[
P(u, v) = \text{dst_ip} \in 10.1.1.0/24
\]
DP as Boolean Expressions: Simpler Example

\[
P(u, v) = \text{true}
\]

**Default routing**
DP as Boolean Expressions: Examples

### Packet filtering

\[ P(u, v) = \text{dst}_\text{ip} \in 10.1.1.0/24 \wedge \text{dst}_\text{port} \neq 80 \]

### Longest prefix matching

\[ P(u, v) = (\text{dst}_\text{ip} \in 10.1.1.0/24 \wedge \text{dst}_\text{ip} \notin 10.1.1.128/25) \lor \text{dst}_\text{ip} \in 10.1.2.0/24 \]
Reachability as Solving SAT

Express reachability from $u$ to $w$ as a Boolean condition:

$u$ can reach $w$

$\iff \exists A$ packet that can flow over $(u,v)$ and $(v,w)$

$\iff \exists A$ packet that makes $P(u,v) \land P(v,w)$ true

$\iff C = (P(u,v) \land P(v,w))$ is satisfiable

General: one path $p$: for each link $l$ in $p$: $P(p) = \land_l P(l)$

General: multiple paths $p_1, p_2, \ldots$: $P(p_1) \lor P(p_2) \lor \ldots$
From 1 Path to $M$ Paths

Ex: What is the Boolean expression to check reachability from $s$ to $t$ in the example above?
From 1 Path to M Paths

- Problem: naïve enumeration of each path may need exponentially many paths
  - Solution: dynamic programming (a.k.a. loop unrolling)
    - Intermediate variables: “Can reach x in k hops?”

```python
function reach(s, t, k, G)
    r[t][0] ← true
    r[v][0] ← false for all v ∈ V(G) \ t
    for i = 1 to k do
        for all v ∈ V(G) \ t do
            r[v][i] ← ∨ (P(v, u) ∧ r[u][i-1])
        end for
    end for
    return ∨ r[s][i] for 1 ≤ i ≤ k
```
Ex: Generate Boolean expression to check whether there are packets originated from $v$ but dropped without reaching any nodes of the destinations $D$ in the network (assume no loop).

```python
function packet_loss(v, D, G)
    n ← the number of network devices in $G$
    d ← a new vertex in $V(G)$
    for all $u ∈ D$ do
        $(u, d) ←$ a new edge in $E(G)$
        $P(u, d) ←$ true
    end for
    $c ←$ reach($v, d, n, G$)
    Test satisfiability of $¬c$
```
Offline Exercise

- Boolean expressions to detect

Loop detection

Consistency
Ex: Packet Transformation

- Essential to model NAT, MPLS, QoS, etc.

- Solution
  - Model the history of packets: vector over time
  - Packet transformation $\Rightarrow$ boolean constraints over adjacent packet versions

\[(p_i \cdot dst\_ip \in 0.1.1.0/24) \land (p_{i+1} \cdot label = 5)\]

In general $p_{i+1} = f(p_i)$
Anteater Success

- Evaluated with UIUC campus network
  - ~178 routers supporting >70,000 machines
  - Predominantly OSPF, also uses BGP and static routing 1,627 FIB entries per router (mean)

- Result: Revealed 23 bugs with 3 invariants in 2 hours

<table>
<thead>
<tr>
<th></th>
<th>Loop</th>
<th>Packet loss</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Being fixed</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Stale config.</td>
<td>0</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>False pos.</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Total alerts</td>
<td>9</td>
<td>17</td>
<td>2</td>
</tr>
</tbody>
</table>
Benefits of Anteater

- First systematic application of Boolean logic to verify some common network properties
- Many follow-up applying SAT in particular and SMT (Satisfiability modulo theories) in general
  - Read SMT tutorial
  - Read Minesweeper on applying SMT to network verification
Discussion

- Limitations of modeling using only SAT?
Outline

- Admin and recap
- Network functions (carrier cloud)
- Network modeling and verification
  - Overview
  - Data plane (DP) based modeling and verification
    - Anteater (network verification as SAT)
    - Header space analysis (HSA: symbolic, set theoretical model of networks)
Goal: an “algebra” to compute complete network behavior (e.g., not only if there are packets which can reach t from s, but the complete set)
HSA: High-Level Approach

Consider each network element as a transfer function
Transfer Function Model

- **Input:**
  - localized packet, indicated by \(<header h, physical port p>\) (why localized?)

- **Output:**
  - localized packets
    - Multiple localized packets (why multiple?)

\[
T : (h, p) \rightarrow \{(h_1, p_1), \ldots, (h_n, p_n)\}
\]
Transfer Function Model

Match

11xx..0x
P=1

Action
Send to port 2
Rewrite with 1x01xx..x1;
Send to port 3
Rewrite with 1xx011..x1

Port ID
Packets in HSA

- Each packet, based on its header bits, is modeled as a point in \( \{0,1\}^L \) space, where \( L \) is the number of header bits.

Packet with header 110

Flow with header 1xx
A complete model of a packet is a localized packet: a packet localized on a physical port is modeled as a point in $\{0,1\}^L \times \{1,\ldots,P\}$ space (assign each one of the $P$ ports a unique port ID).
Transfer Function: Example

- **IPv4 Router - Forwarding**
  
  - 172.24.74.0  255.255.255.0  Port1
  - 172.24.128.0  255.255.255.0  Port2
  - 171.67.0.0  255.255.0.0  Port3

\[ T(h, p) = \begin{cases} 
  (h,1) & \text{if } \text{dst}_\text{ip}(h) = 172.24.74.x \\
  (h,2) & \text{if } \text{dst}_\text{ip}(h) = 172.24.128.x \\
  (h,3) & \text{if } \text{dst}_\text{ip}(h) = 171.67.x.x 
\end{cases} \]
Transfer Function: Example

IPv4 Router - Forwarding + TTL

- 172.24.74.0 255.255.255.0 Port1
- 172.24.128.0 255.255.255.0 Port2
- 171.67.0.0 255.255.0.0 Port3

\[
T(h, p) = \begin{cases} 
(d_{\text{dec}_\text{ttl}}(h),1) & \text{if } \text{dst\_ip}(h) = 172.24.74.x \\
(d_{\text{dec}_\text{ttl}}(h),2) & \text{if } \text{dst\_ip}(h) = 172.24.128.x \\
(d_{\text{dec}_\text{ttl}}(h),3) & \text{if } \text{dst\_ip}(h) = 171.67.x.x 
\end{cases}
\]
Transfer Function: Example

- **IPv4 Router - Forwarding+TTL+MAC rewrite**

  - 172.24.74.0  255.255.255.0  Port1
  - 172.24.128.0 255.255.255.0  Port2
  - 171.67.0.0   255.255.0.0   Port3

  \[
  T(h, p) = \begin{cases} 
  (\text{rw\_mac}(\text{dec\_ttl}(h),\text{next\_mac}), 1) & \text{if dst\_ip}(h) = 172.24.74.x \\
  (\text{rw\_mac}(\text{dec\_ttl}(h),\text{next\_mac}), 2) & \text{if dst\_ip}(h) = 172.24.128.x \\
  (\text{rw\_mac}(\text{dec\_ttl}(h),\text{next\_mac}), 3) & \text{if dst\_ip}(h) = 171.67.x.x 
  \end{cases}
  \]
Transfer Function Actions

- **Rewrite**: rewrite bits 0--2 with value 101
  - $(h & 000111...) | 101000…$

- **Encapsulation**: encapsulate packet in a 1010 header
  - $(h >> 4) | 1010…$

- **Decapsulation**: decap 1010 header
  - $(h << 4)$

- **Load balance using hashing on 5 tuple**
  - $LB(h, P) = \{(h, P1), ..., (h, Pn)\}$
Composing Transfer Functions

\[ T_3(T_2(T_1(h, p))) \]
Inverse Transfer Function

- Tell us all possible input packets that can generate an output packet.
Q: What are the packets that go from A to B using the path A->Box1->Box2->Box3->B?
Q: What are the packets that go from A to B using
   - either the path A->Box1->Box2->Box3->B
   - or the path A->Box1->Box4->Box3->B?

Q: how to compute packets at A to reach B?

Q: what are the packets: seen at B or A?

\[ T_3(T_2(T_1(X,A))) \lor T_3(T_4(T_1(X,A))) \]
Ex: Reachability Example
Figure 2: Example for computing reachability function from $a$ to $b$. For simplicity, we assume a header length of 8 and show the first 4 bits on the x-axis and the last 4 bits on the y-axis. We show the range (output) of each transfer function composition along the paths that connect $a$ to $b$. At the end, the packet headers that $b$ will see from $a$ are $01011x10 \cup 10010x10$. 
Reachability Example

Figure 2: Example for computing reachability function from \(a\) to \(b\). For simplicity, we assume a header length of 8 and show the first 4 bits on the \(x\)-axis and the last 4 bits on the \(y\)-axis. We show the range (output) of each transfer function composition along the paths that connect \(a\) to \(b\). At the end, the packet headers that \(b\) will see from \(a\) are 01011x10 ∪ 10010x10.
Reachability Example

- **Range:** Ra→b is $10010x10 \cup 01011x10$ as expected

- To find the set of headers that a can send to b, compute

\[
T_A^{-1}(\Gamma(T_B^{-1}(\Gamma(T_C^{-1}(\Gamma(T_E^{-1}(10010x10, E_2)))))))) \cup \\
T_A^{-1}(\Gamma(T_B^{-1}(\Gamma(T_C^{-1}(\Gamma(T_D^{-1}(\Gamma(T_E^{-1}(01011x10, E_2))))))))),
\]
Reachability Complexity

- Assume Linear Fragmentation assumption.
- Under this assumption, the running time is $O(dR^2)$ where
  - $d$ is the network diameter - the maximum number of hubs that a packet will go through before reaching the destination,
  - $R$ the maximum number of forwarding rules in a router.

- By contrast, a brute-force algorithm that simulates the sending of every possible packet has $O(2^L)$ complexity.
Reachability -> Other Path Properties

- Blackhole freedom (A -> B and notice unexpected drop)
- Communication via middle box. (A->B packets must pass through C)
- Maximum hop count (length of path from A -> B never exceeds L)
- Isolation of paths (http and https traffic from A->B don’t share the same path)
- Loop freedom (No packet should be reachable from any port to itself)
Finding Loops

- Is the loop infinite?
Benefits

- Analyzed Stanford networks and found 26 loops
## Computation Time

<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to generate Network and Topology Transfer Function</td>
<td>151 s</td>
</tr>
<tr>
<td>Runtime of loop detection test (30 ports)</td>
<td>560 s</td>
</tr>
<tr>
<td>Average per port runtime</td>
<td>18.6 s</td>
</tr>
<tr>
<td>Max per port runtime</td>
<td>135 s</td>
</tr>
<tr>
<td>Min per port runtime</td>
<td>8 s</td>
</tr>
<tr>
<td>Average runtime of reachability test</td>
<td>13 s</td>
</tr>
</tbody>
</table>

*Table 2: Runtime of loop detection and reachability tests on Stanford backbone network*
Outline

- Admin and recap
- Network functions (carrier cloud)
- Network modeling and verification
  - Overview
  - Data plane (DP) based modeling and verification
    - Anteater (network verification as SAT)
    - Header space analysis (HSA: symbolic, set theoretical model of networks)
    - Real-time DP analysis
Discussion

- Benefits of real-time verification?

- Challenges of real-time verification?
Why Real-time Verification

- Changes
- New DP
- RT-Verify

Good

Bad
- Block,
- Diagnosis, or
- Adapt
Outline

- Admin and recap
- Network functions (carrier cloud)
- Network modeling and verification
  - Overview
  - Data plane (DP) based modeling and verification
    - Anteater (network verification as SAT)
    - Header space analysis (HSA: symbolic, set theoretical model of networks)
    - Real-time DP analysis
      - Real-time HSA
Discussion

- How to speed up HSA?
HSA w/ Pre-Computation & Dependency Marking
Reachability
Updating Update Graph: Add a Rule
Updating Update Graph: Delete a Rule
Performance: Computation Time

![Google inter-datacenter WAN network.](image)

<table>
<thead>
<tr>
<th>Network:</th>
<th>Google</th>
<th>Stanford</th>
<th>Internet 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Rule (ms)</td>
<td>0.28</td>
<td>0.2</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>0.23</td>
<td>0.065</td>
<td>0.52</td>
</tr>
<tr>
<td>Add Link (ms)</td>
<td>1510</td>
<td>3020</td>
<td>4760</td>
</tr>
<tr>
<td></td>
<td>1370</td>
<td>2120</td>
<td>2320</td>
</tr>
</tbody>
</table>

Table 3: Average and median run time of NetPlumber, for a single rule and link update, when only one source node is connected to NetPlumber.
Outline

- Admin and recap
- Network functions (carrier cloud)
- Network modeling and verification
  - Overview
  - Data plane (DP) based modeling and verification
    - Anteater (network verification as SAT)
    - Header space analysis (HSA: symbolic, set theoretical model of networks)
    - Real-time DP analysis
      - Real-time HSA
      - VeriFlow
An alternative approach to computing incremental verification
A “competitor” of HSA
Veriflow Basic Idea

- A different incremental decomposition
  - Consider the partition of match space as defining a set of equivalence classes (EC)
  - Assume that each update will change only a small number of ECs.
Ex: IP Prefix as an Interval

- Consider each IP prefix as a half closed interval \([a, b)\)

- Examples
  - 0.0.0.10/31
    - \([10, 11)\)
  - 0.0.0.0/0
    - \([0, 2^{32}-1)\)
  - 0.0.0.0/1
    - \([0, 2^{31}-1)\)
IP Prefixes and Equivalent Classes
IP Prefixes and Equivalent Classes
Equivalent Class => EC
Forwarding Graph
Equivalent Classes under Changes: Add a New Rule

**Equivalence Class 1**

**Equivalence Class 2**

**Equivalence Class 3**
IP Prefixes and Equivalent Classes: EC as adding a new rule

Equivalence Class 1'  
Equivalence Class 3'
**ViriFlow Architecture**

- **Network Controller**
  - New rules

- **ViriFlow**
  - Generate equivalence classes
  - Generate forwarding graphs
  - Run queries

- **Diagnosis report**
  - Type of invariant violation
  - Affected set of packets

- **Good rules**

- **Rules violating network invariant(s)**
Generic Approach to Computing Equivalence Classes

(device, rule) pairs

(don’t care/wildcard)

Equivalence classes

Header value ranges
97.8% of the updates were verified within 1 millisecond.
Effect of Equivalence Class Count

Number of ECs strongly influences verification time

Average verification time (ms)

Number of ECs affected by new rule
VeriFlow: Summary

- An interesting design to identify incremental/reusable computation

- Multiple remaining issues
  - How does VeriFlow scale in a large network?
  - How to handle packet transformation?
  - How to handle multiple controllers?
Outline

- Admin and recap
- Network modeling and verification
  - Overview
  - Data plane (DP) based modeling and verification
    - Anteater (network verification as SAT)
    - Header space analysis (HSA; symbolic, set theoretical model of networks)
    - Real-time DP analysis
      - Real-time HSA
      - VeriFlow
      - Atomic predicate verifier (APV)
APV: Motivation

- Potential Veriflow problem: naïve EC may generate a huge number of ECs (ECs and EC forwarding graph).

<table>
<thead>
<tr>
<th></th>
<th>Stanford</th>
<th>Internet2</th>
<th>Purdue</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of boxes</td>
<td>16</td>
<td>9</td>
<td>1,646</td>
</tr>
<tr>
<td>No. of ports used</td>
<td>58</td>
<td>56</td>
<td>2,736</td>
</tr>
<tr>
<td>No. of rules</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forwarding</td>
<td>757,170</td>
<td>1,584</td>
<td>126,017</td>
</tr>
<tr>
<td>ACL</td>
<td>1,584</td>
<td>126,017</td>
<td>3,605</td>
</tr>
</tbody>
</table>
**APV Basic Idea**

- Consider each EC as defined by a predicate.
- Instead of enumerating (raw) predicates, consider atomic predicates generating the predicates
  - think of a base as in linear algebra
Atomic Predicates - Definition

Given a set $\mathcal{P}$ of predicates, its set $\{p_1, \ldots, p_k\}$ of atomic predicates satisfies five properties:

1. $p_i \neq \text{false}, \forall i \in \{1, \ldots, k\}$
2. $\forall p_i = \text{true}$
3. $p_i \land p_j = \text{false}$, for $i \neq j$
4. Each predicate $P \in \mathcal{P}, P \neq \text{false}$, is equal to the disjunction of a subset of atomic predicates:
   $$P = \bigvee_{i \in S(P)} p_i,$$
   where $S(P) \subseteq \{1, \ldots, k\}$
5. $k$ is the minimum number such that the set $\{p_1, \ldots, p_k\}$ satisfies the above four properties
Simple Alg to Compute Atomic Predicates

Compute the set of atomic predicates for predicate $P$:

$$\mathcal{A}([P]) = \begin{cases} \{\text{true}\} & \text{if } P = \text{false} \text{ or true} \\ \{P, \neg P\} & \text{otherwise} \end{cases}$$

$P_1, P_2$ two sets of predicates. $P_1$'s set of atomic predicates is $\{b_1, \ldots, b_l\}$ and $P_2$'s set of atomic predicates is $\{d_1, \ldots, d_m\}$. Compute a set of predicates, $\{a_1, \ldots, a_k\}$:

$$\{a_i = b_{i_1} \land d_{i_2} | a_i \neq \text{false}, i_1 \in \{1, \ldots, l\}, i_2 \in \{1, \ldots, m\}\}$$
## Atomic Predicates in Real Networks

<table>
<thead>
<tr>
<th></th>
<th>Stanford</th>
<th>Internet2</th>
<th>Purdue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No. of rules</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forwarding</td>
<td>757,170</td>
<td>126,017</td>
<td></td>
</tr>
<tr>
<td>ACL</td>
<td>1,584</td>
<td>3,605</td>
<td></td>
</tr>
<tr>
<td><strong>No. of atomic predicates</strong></td>
<td>494</td>
<td>21</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>3,917</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Time to Compute Atomic Predicates

<table>
<thead>
<tr>
<th>atomic predicates for ACLs</th>
<th>random selection (ms)</th>
<th>smallest ACL first (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanford</td>
<td>1.56</td>
<td>0.84</td>
</tr>
<tr>
<td>Purdue</td>
<td>886.21</td>
<td>450.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>atomic predicates for forwarding</th>
<th>random selection (ms)</th>
<th>selection by box (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanford</td>
<td>210.26</td>
<td>201.40</td>
</tr>
<tr>
<td>Internet2</td>
<td>154.91</td>
<td>148.28</td>
</tr>
</tbody>
</table>
Result Depend on Order

![Graph showing the dependence of results on order]

- **Random selection**
- **Smallest ACL first**

The graph illustrates the number of atomic predicates versus the number of ACLs. The results depend on the order in which the ACLs are processed.
Network Verification using Atomic Predicates: Packet Set Specification

- Assume only forwarding and ACL
- The set $P$ of packets that can pass through an output port is specified by the conjunction of its predicates for forwarding and ACLs, where

\[ P = \left( \forall i \in S_F \ f_i \right) \land \left( \forall j \in S_A \ a_j \right) \]

- $S_F$ is the set of integer identifiers of atomic predicates for forwarding,
- $S_A$ the set of integer identifiers of atomic predicates for ACLs.
APV: Example

A small network example.

The reachability tree of *port*$_1$.
Storage Cost

<table>
<thead>
<tr>
<th></th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hassel in C</td>
<td>323.06</td>
</tr>
<tr>
<td>AP Verifier</td>
<td>8.7/0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hassel in C</td>
<td>187.60</td>
</tr>
<tr>
<td>AP Verifier</td>
<td>6.7/2</td>
</tr>
</tbody>
</table>

Stanford network (58 ports)  Internet2 (56 ports)

- Storing reachability trees for all ports
  - Hassel in C (HSA) required 37 times more memory for the Stanford network and 28 times more memory for Internet2
Loop Detection by Computing the Reachability Tree for One Port

Reachability tree computation from one port (loop detection) in Stanford network.

<table>
<thead>
<tr>
<th></th>
<th>Average (ms)</th>
<th>Median (ms)</th>
<th>Maximum (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hassel in C</td>
<td>218.22</td>
<td>53.45</td>
<td>1881.41</td>
</tr>
<tr>
<td>AP Verifier</td>
<td>0.95</td>
<td>1.03</td>
<td>1.38</td>
</tr>
<tr>
<td>AP Verifier (BDD)</td>
<td>4.23</td>
<td>4.21</td>
<td>10.67</td>
</tr>
</tbody>
</table>

Reachability tree computation from one port (loop detection) in Internet2.

Twelve infinite loop paths in the Stanford network
Two infinite loop paths in Internet2
Benefits of Atomic Predicates

- Atomic predicates for a given set of predicates
  - They specify the (base to specify) equivalence classes of packets
- Each predicate stored and represented as a set of integers
  - Space efficient
- Conjunction (disjunction) of two predicates computed as intersection (union) of two sets of integers
  - Time efficient