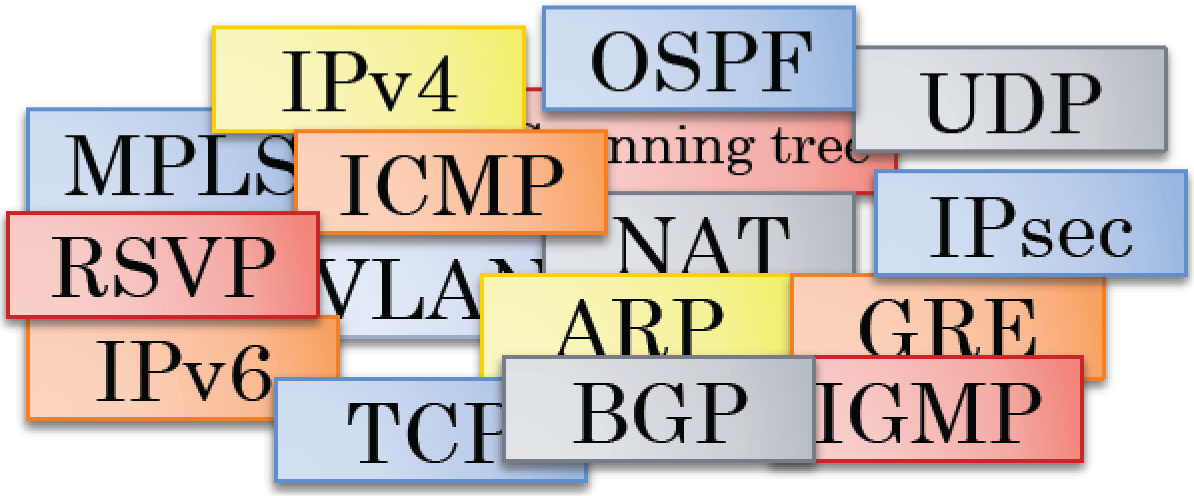
Network verification and synthesis

Protocols for Data Networks

(aka Advanced Computer Networks)

Networks are complex

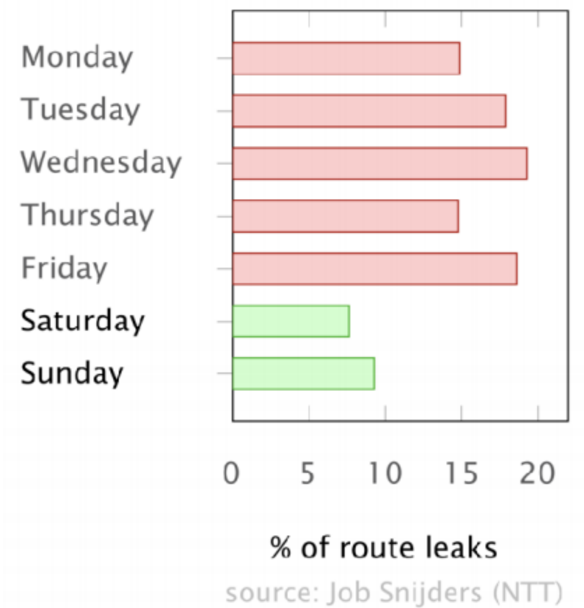
- A typical network is a complex mix of protocols



- Protocols interact in complex ways, causing unforeseen behavior
- Hard to manage, understand and predict the behavior of networks

Motivation

- **The Internet seems to be better off during week-ends…**



"Human factors are responsible for 50% to 80% of network outages"

Juniper Networks, What's Behind Network Downtime?, 2008

Motivation


The Register®
Biting the hand that feeds IT

DATA CENTRE SOFTWARE SECURITY DEVOPS BUSINESS PERSONAL TECH SCIENCE

Data Centre ▸ Networks

Google routing blunder sent Japan's Internet dark on Friday

Another big BGP blunder

By Richard Chirgwin 27 Aug 2017 at 22:35 40 SHARE ▼

Last Friday, someone in Google fat-thumbed a border gateway protocol (BGP) advertisement and sent Japanese Internet traffic into a black hole.

- Someone in Google fat-thumbed a BGP advertisement and sent Japanese Internet traffic into a black hole.
- The outage in Japan only lasted a couple of hours but was so severe that […] the country's Internal Affairs and Communications ministries want carriers to report on what went wrong.

Lecture plan

[NetComplete]

A system that synthesis network configurations, taking as input configurations with "holes" and "autocompleting" them

[p4v]

A tool for verifying data planes described using the P4 programming language

[Facon]

A tool that automatically generates SDN programs

Lecture plan

[NetComplete]

A system that synthesis network configurations, taking as input configurations with "holes" and "autocompleting" them

[p4v]

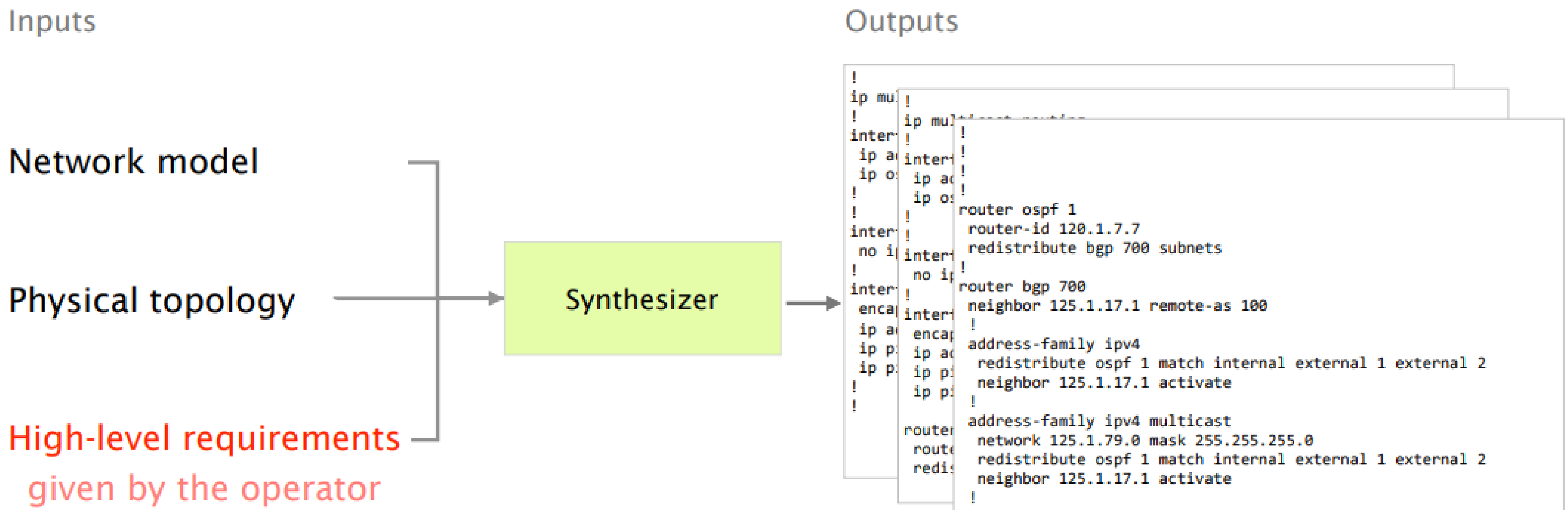
A tool for verifying data planes described using the P4 programming language

[Facon]

A tool that automatically generates SDN programs

Configuration synthesis

- Configuration synthesis addresses the previous problems by deriving low-level configurations from high-level requirements



Challenges

- **Problem #1 interpretability**
  - Existing synthesizers can produce configurations that widely differ from humanly-generated ones
- **Problem #2 continuity**
  - Existing synthesizers can produce widely different configurations given slightly different requirements
- **Problem #3 scalability**
  - Existing synthesizers do not scale to large networks

A key issue is that synthesizers do not provide operators with a fine-grained control over the synthesized configurations

NetComplete

- NetComplete allows network operators to flexibly express their intents through configuration sketches
  - A configuration with "holes"
  - Holes can identify specific attributes such as: IP addresses, link costs, BGP local preferences
  - Or entire pieces of the configuration

```
interface TenGigabitEthernet1/1/1
  ip address ? ?
  ip ospf cost 10 < ? < 100

router ospf 100
 ?
 ...

router bgp 6500
  ...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit ?
ip community-list C2 permit ?
```

```
route-map imp-p1 permit 10
  ?

route-map exp-p1 ? 10
  match community C2
route-map exp-p2 ? 20
  match community C1
...
```

Netcomplete

- NetComplete "autocompletes" the holes such that the output configuration complies with the requirements
  - By reducing the autocompletion problem to a constraint satisfaction problem

```
interface TenGigabitEthernet1/1/1
  ip address 10.0.0.1 255.255.255.254
  ip ospf cost 15

router ospf 100
  network 10.0.0.1 0.0.0.1 area 0.0.0.0


router bgp 6500
  ...
  neighbor AS200 import route-map imp-p1
  neighbor AS200 export route-map exp-p1
  ...
ip community-list C1 permit 6500:1
ip community-list C2 permit 6500:2
```

```
route-map imp-p1 permit 10
  set community 6500:1
  set local-pref 50
route-map exp-p1 permit 10
  match community C2
route-map exp-p2 deny 20
  match community C1
...
```

Main steps

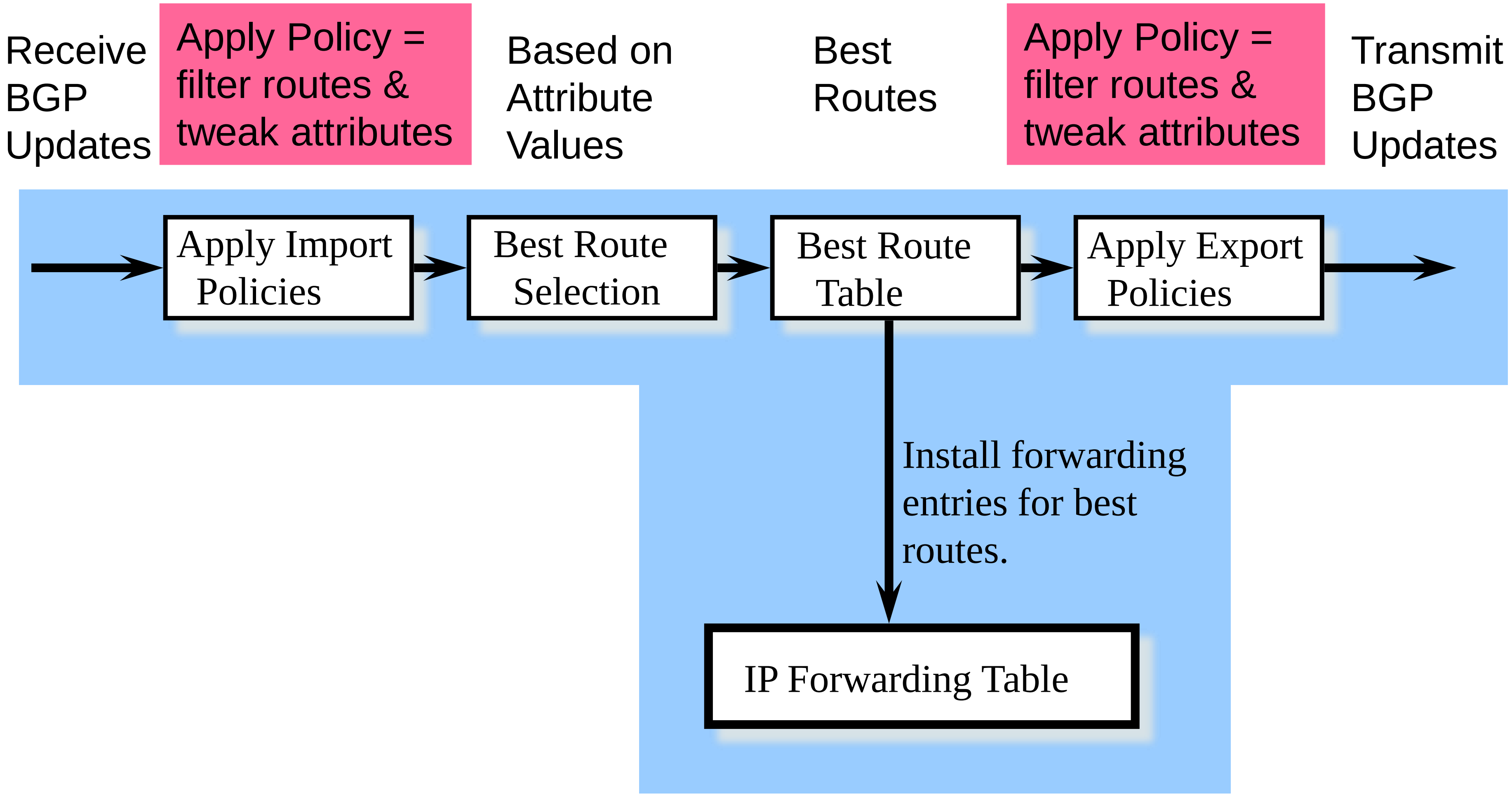
1. Encode the protocol semantics, high-level requirements, and partial configurations as a logical formula (in SMT)

2. Use a solver (Z3) to find an assignment for the undefined configuration variables such that the formula evaluates to True

- Main challenge: scalability

- Main techniques
  - BGP synthesis: optimised encoding
  - OSPF synthesis: counter-examples-based

BGP Policy


Receive BGP Updates
Apply Policy = filter routes & tweak attributes
Based on Attribute Values
Best Routes
Apply Policy = filter routes & tweak attributes
Transmit BGP Updates
Apply Import Policies
Best Route Selection
Best Route Table
Apply Export Policies
Install forwarding entries for best routes.
IP Forwarding Table

BGP decision process

Step	Attribute	Controlled by local or neighbor AS?
1.	Highest LocalPref	Local
2.	Lowest AS path length	Neighbor
3.	Lowest origin type	Neither
4.	Lowest MED	Neighbor
5.	eBGP-learned over iBGP-learned	Neither
6.	Lowest IGP cost to border router	Local
7.	Lowest router ID (to break ties)	Neither

BGP synthesis

- **NetComplete autocompletes router-level BGP policies by encoding the desired BGP behavior as a logical formula**

$$M \models \text{Reqs} \wedge \text{BGP}_{\text{protocol}} \wedge \text{Policies}$$

- **Reqs: how should the network forward traffic**
  - **concrete, part of the input**

$$\text{R1.BGP}_{\text{select}}(\text{A1},\text{A2}) \wedge \text{R1.BGP}_{\text{select}}(\text{A2},\text{A3})$$

- **$\text{BGP}_{\text{protocol}}$:how do BGP routers select routes**
  - **concrete, protocol semantic**

$$\text{BGPselect}(X,Y) \Leftrightarrow (X.\text{LocalPref} > Y.\text{LocalPref}) \vee \ldots$$

- **Policies: how routes should be modified**
  - **symbolic, to be found**

R1.SetLocalPref(A1) = VarX; R1.SetLocalPref(A2) = 200

Challenges

- **Naive encodings lead to complex constraints that cannot be solved in a reasonable time**
  - **The search space for policies can be huge**
  - **Solution: partial evaluation**

- **Another problem are the interactions between the inter-domain (BGP) and intra-domain (OSPF) protocols**
  - **Solution: iterative synthesis**

Solving logical formula

- **Solving the logical formula consists in assigning each symbolic variable with a concrete value**

BGPselect(X,Y) <=> (X.LocalPref > Y.LocalPref) ∨ …

$$M \vDash Reqs \wedge BGP_{protocol} \wedge Policies$$

$R1.BGP_{select}(A1,A2) \wedge R1.BGP_{select}(A2,A3)$

R1.SetLocalPref(A1) = **VarX**
R1.SetLocalPref(A2) = 200

- **M**
  - **VarX := 250**

Challenge #1

- **Naive encodings lead to complex constraints that cannot be solved in a reasonable time**
  - **The search space for policies can be huge**

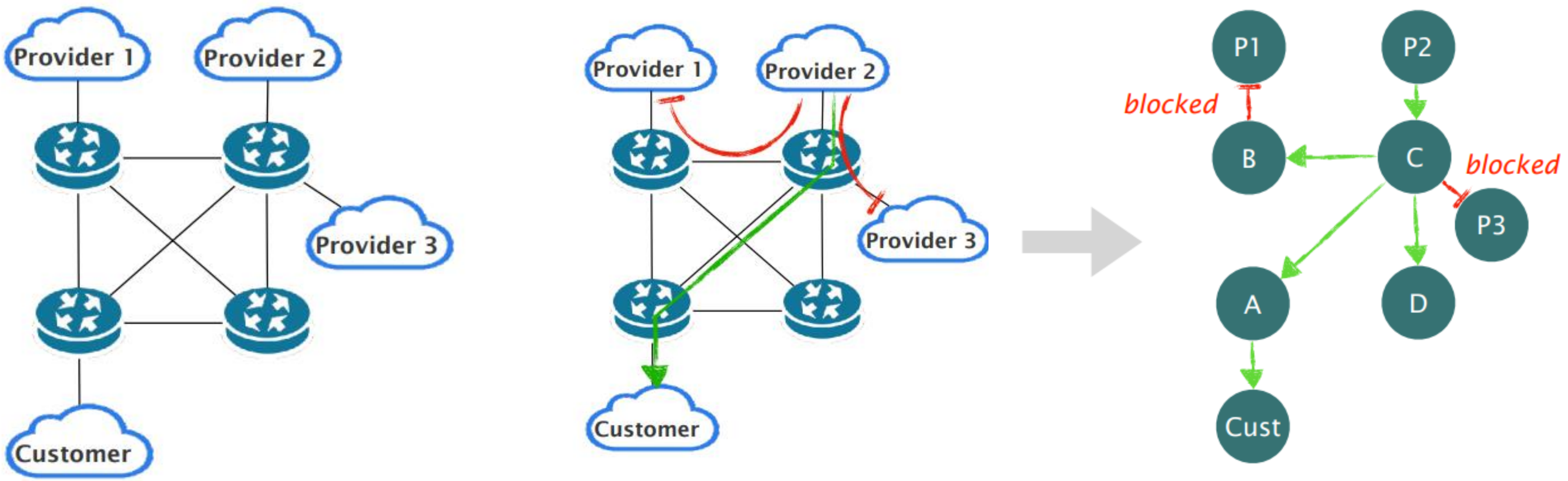
- **Solution: partial evaluation**
  - **NetComplete encodes reduced policies by relying on the requirements and the sketches**

1. **Capture how announcements should propagate using the requirements: BGP propagation graph**

2. **Combine the graph with constraints imposed by sketches via symbolic execution: partially evaluated formulas**

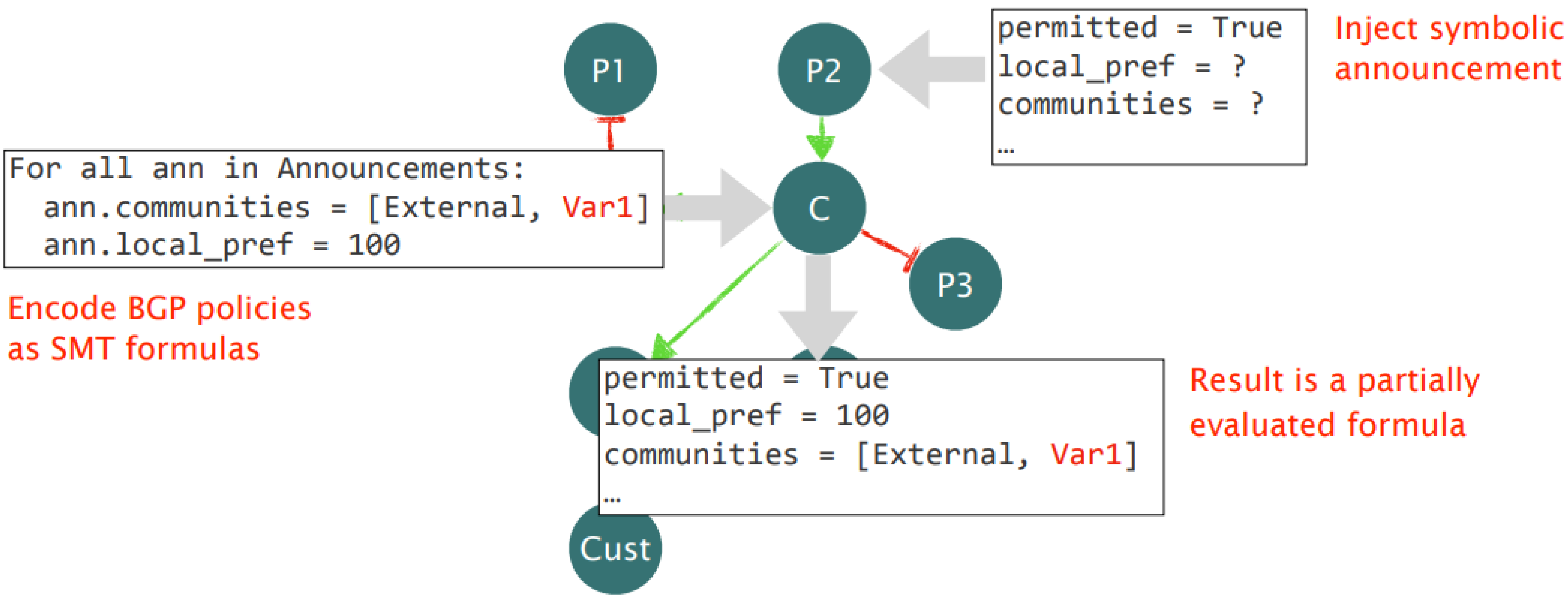
BGP propagation graph

- NetComplete relies on the requirements to figure out where BGP announcements should (not) propagate
- Example: only customers should be able to send traffic to Provider #2



Partially evaluated formulas

- **NetComplete concretizes symbolic announcements by propagating them through the graph and sketches**



Iterative synthesis

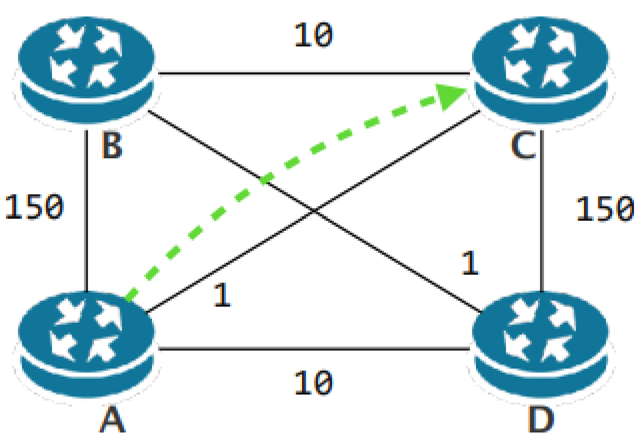
- BGP may select routes based on path costs (computed by OSPF)
- When this is necessary the BGP synthesizer outputs additional requirements to be enforced by the OSPF synthesizer

OSPF synthesis

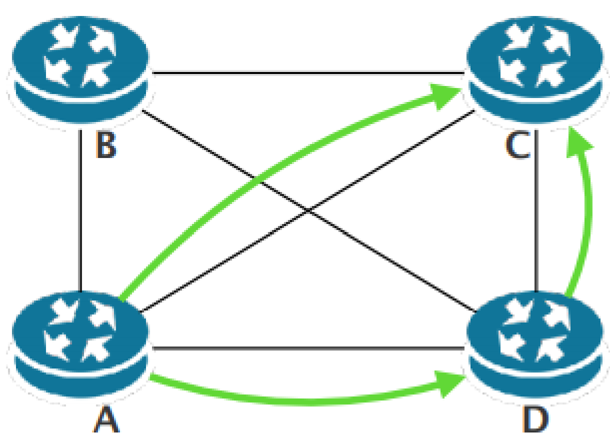
- As for BGP, Netcomplete phrases the problem of finding weights as a constraint satisfaction problem

- Example: for performance reasons, the operator wants to enable load-balancing

Initial configuration



Desired configuration



- What should be the weights for this to happen?

Synthesis procedure

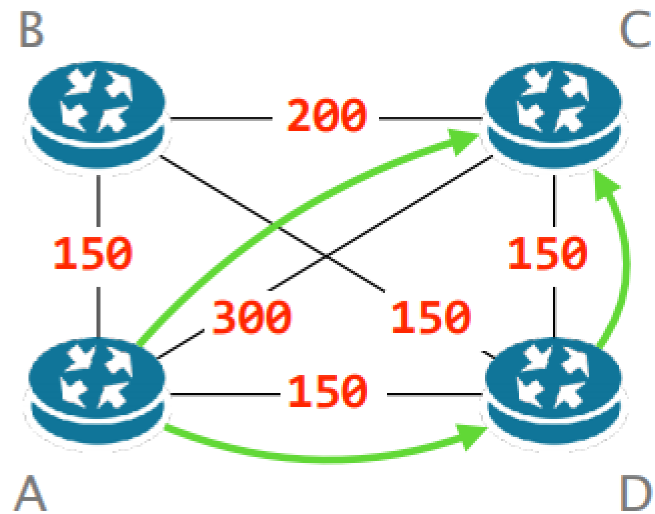
$\forall X \in \text{Paths}(A,C) \backslash \text{Reqs}$

↓

$\text{Cost}(A \to C) = \text{Cost}(A \to D \to C) < \text{Cost}(X)$

↓

Solve



Synthesized weights

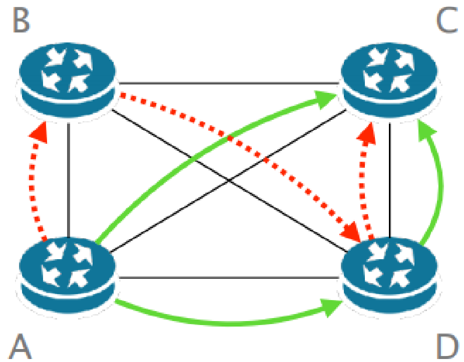
- **Problem: it does not scale**
  - $\forall X \in$ Paths(A,C)
  - There can be an exponential number of paths between A and C…

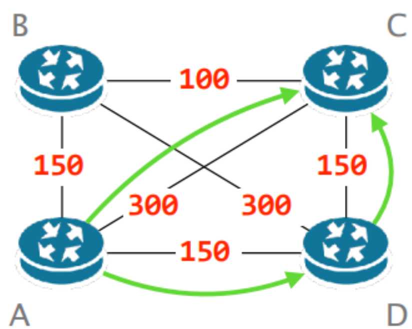
Solution: CEGIS

- **Counter-Example Guided Inductive Synthesis**
  - A contemporary approach to synthesis where a solution is iteratively learned from counter-examples

- **While enumerating all paths is hard, computing shortest paths given weights is easy!**

Synthesis procedure





Synthesized weights

$\forall X \in$ SamplePaths(A.C)\Reqs

Sample: { [A,B,D,C] }

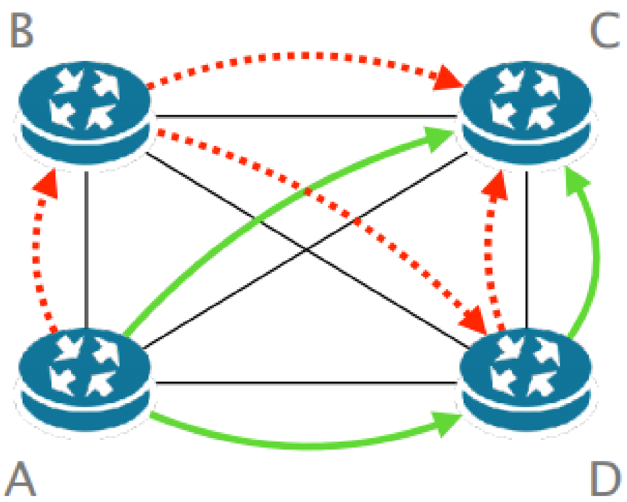
Cost(A→C) = Cost(A→D→C) < Cost(X)

Solve

- **Wait: The synthesized weights are incorrect: cost(A → B → C]) = 250 < cost(A → C) = 300**

Synthesis procedure

- Now, add the counter example to SamplePaths and repeat the procedure



$\forall X \in \text{SamplePaths(A,C)} \setminus \text{Reqs}$

Sample: { [A,B,D,C] } U { [A,B,C] }

- The entire procedure usually converges in few iterations making it very fast in practice

Main results

- **NetComplete synthesizes configurations for large networks in few minutes**
  - **Previous work could take hours or days**

- **With CEGIS, OSPF synthesis is >100x faster**

Lecture plan

[NetComplete]

A system that synthesis network configurations, taking as input configurations with "holes" and "autocompleting" them

[p4v]

A tool for verifying data planes described using the P4 programming language

[Facon]

A tool that automatically generates SDN programs

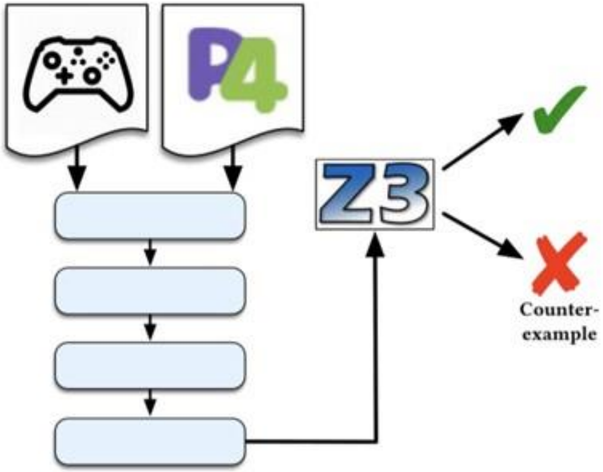
Motivation

- **Fixed-function data planes**
  - How do we know that they work?
  - Testing
    - Expensive – lots of packet formats & protocols

- **Programmable data planes**
  - Give programmers language-based verification tools

p4v

- Automated tool for verifying P4 programs
- Considers all paths
  - But also practical for large programs
- Includes basic safety properties for any program
- Extensible framework
  - Verify custom, program-specific properties
  - Or assert-style debugging



Example P4 program

- IPv6 router w/ access control list (ACL)

```
control ingress { apply(acl); }

table acl {
  reads { ipv6.dstAddr: lpm; }
  actions { allow; deny; }
}

action allow() {
  modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }
```

- What could go wrong?

What could go wrong?

- What if we didn’t receive an IPv6 packet? ipv6 header will be invalid
  - Implementations are free to return an arbitrary result
- Table reads arbitrary values
  - Intended ACL policy may be violated
- Can read values from a previous packet
  - Side channel vulnerability!
- Property #1: header validity
  - Real programs are complicated: hard to keep validity in your head

Types of properties

- General safety
  - Header validity
  - Arithmetic-overflow checking
  - Index bounds checking (header stacks, registers, meters, ...)

What could go wrong?

- What if acl table misses (no rule matches)?
  - Forwarding decision is unspecified
- What goes wrong
  - Forwarding behaviour depends on hardware
  - May not do what you expect!
  - Code not portable
- Property #2: unambiguous forwarding

Types of properties

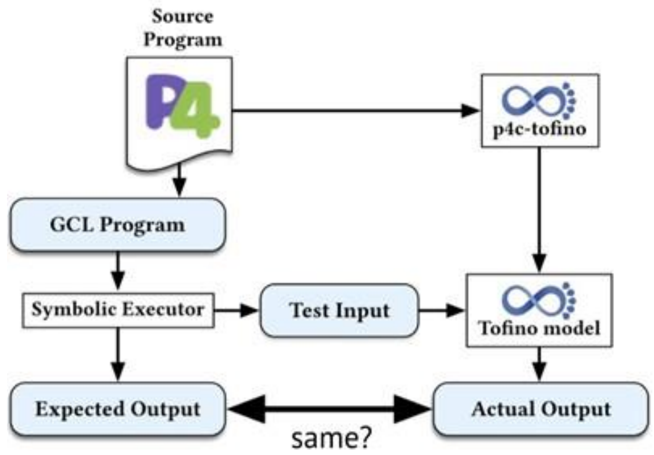
- General safety
  - Header validity
  - Arithmetic-overflow checking
  - Index bounds checking (header stacks, registers, meters, ...)

- Architectural
  - Unambiguous forwarding
  - Reparseability
  - Mutual exclusion of headers
  - Correct metadata usage (e.g., read-only metadata)

- Program-specific
  - Custom assertions in P4 program – e.g., IPv4 ttl correctly decremented

Challenge #1: imprecise semantics

- P4 language spec doesn't give precise semantics
  - Precise meaning of many constructs is not entirely clear
- Solution: defined semantics by translation to GCL (a simple imperative language)
- How do they ensure the p4v frontend captures the intended P4 semantics?
  - By testing: symbolically executed GCL to generate input-output tests for several programs



Challenge #2: modelling control plane

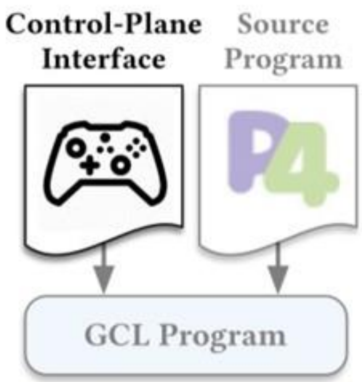
- **A P4 program is just half the program**
  - Table rules are not statically known
  - Populated by the control plane at run time

- **We could delay verification until the forwarding rules are known, and then verify the entire program**
  - Problem: it changes verification from compile-time to run-time, requiring repeatedly verifying the program when rules change

- **Solution: constrain the behavior of the control plane using symbolic constraints in a control-plane interface**
  - Symbolic constraints means we do not need to specify the exact values

Control plane interface

- **Given as second input to p4v**
  - **Constrains choices made by tables**
  - **Written in domain-specific syntax**
    - **Instrument the program with "zombie" state**




```
table acl {
  reads {
    ipv6.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```

```
assume
  reads(acl, ipv6.dstAddr) == 2001:db8::/32
implies
  action(acl) == deny
```

```
table tunnel_decap {
  ...
  actions { decap_6in4; }
}

table tunnel_term {
  ...
  actions { term_6in4; }
}
```

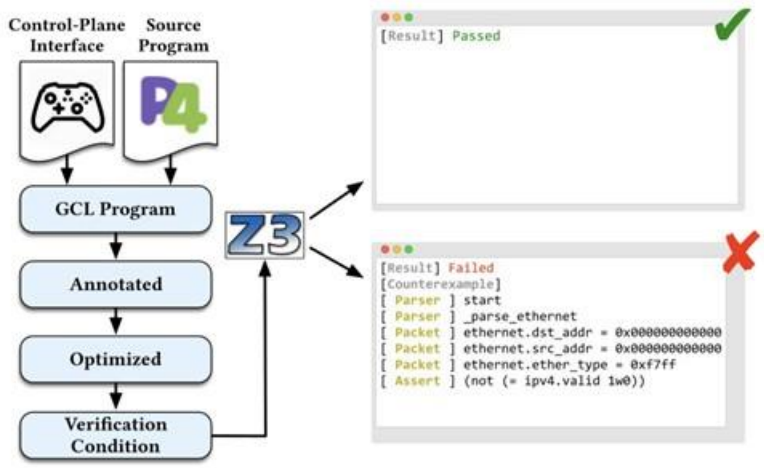
```
assume
  action(tunnel_decap) == decap_6in4
iff
  action(tunnel_term) == term_6in4
```

Challenge #3: scaling to large programs

- Not using compositional verification
  - High burden: needs annotations at component boundaries
- Not using symbolic execution
  - Exponential path explosion → explicitly exploring paths is not tractable
  - P4 programs are very branchy
- Instead, generate single logical formula (a verification condition)
  - Formula valid ⇔ program satisfies assertions on all execution paths
  - Hand formula to solver → verification success or counterexample
- Also some standard optimizations
  - Constant folding / propagation
  - Dead-code elimination

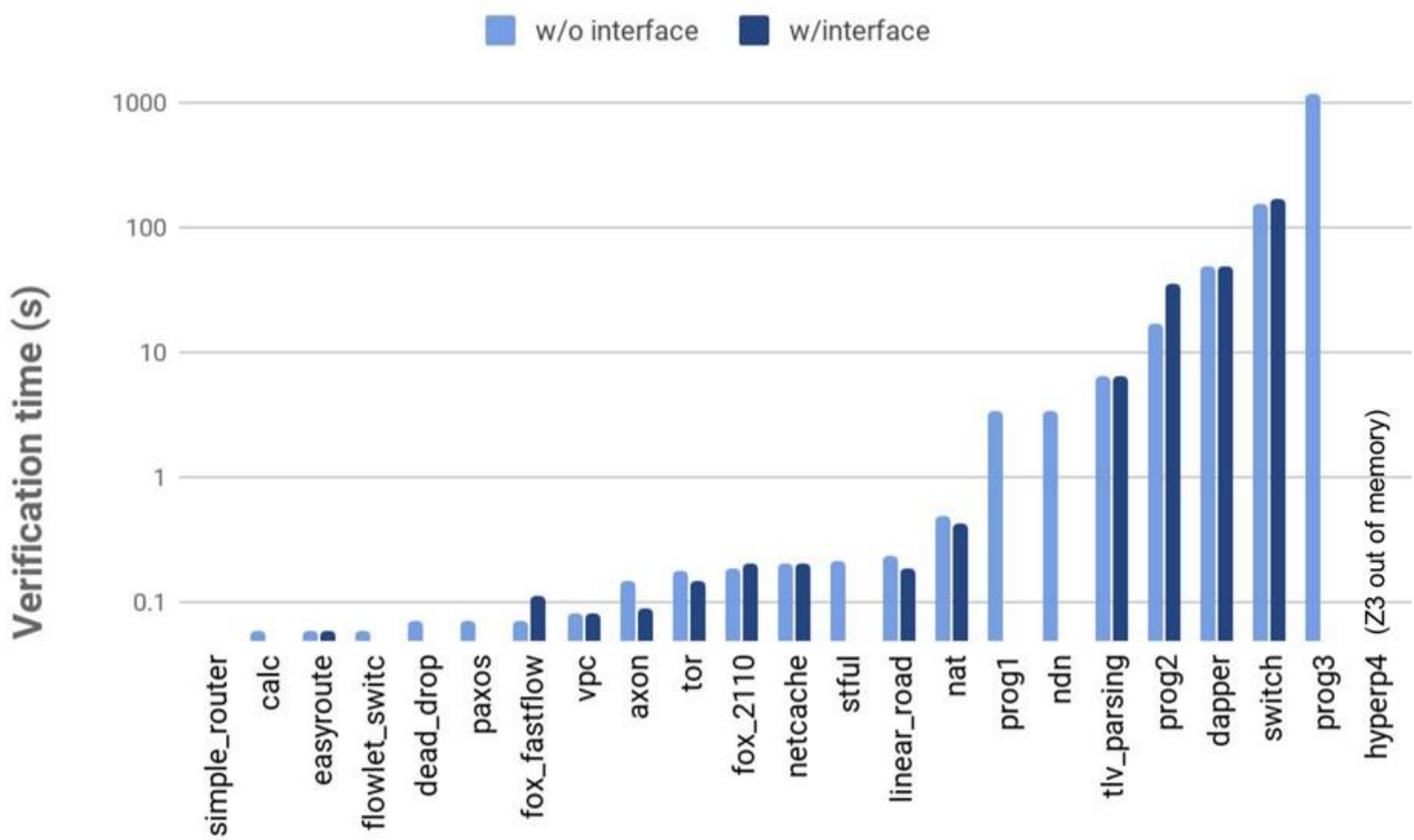
p4v architecture

1. Start w/ CPI & P4 program
2. Translate to GCL
3. Auto-annotate w/ assertions
4. Standard optimizations
5. Generate formula
6. Send to Z3
7. Success or counter example
   - Input packet
   - Program trace
   - Violated assertion



Evaluation

- Found bugs in many P4 programs
- Performance:



Lecture plan

[NetComplete]

A system that synthesis network configurations, taking as input configurations with "holes" and "autocompleting" them

[p4v]

A tool for verifying data planes described using the P4 programming language

[Facon]

A tool that automatically generates SDN programs

Problem

- SDN mitigates distributed complexity by centralized view
  - but controller programs are still complicated to implement

- High-level Domain-Specific Languages (DSL) reduce lines of codes, but have steep learning curve ([Frenetic], [Pyretic], etc.)

Facon

- Networking by input-output examples

- Network operator provides some input-output (I/O) pairs, and the computer automatically synthesizes a program

- Advantages of synthesizing program (vs configuration)
  - More understandable to human
  - The root cause of a configuration error can be a bug in the program
  - Compose with other programs to form complex features
  - Reuse in other settings

Approach

- Synthesize NDLog program
  - Leverage the compactness of NDLog programs
    - Smaller search space for program synthesis
  - Enables "divide-and-conquer" approach to the problem
- NDLog (Network Datalog)
  - Logic-programming family
  - Inputs and Outputs are organized as structured tables
  - Program consists of a set of rules
  - Rules transform input to output

Input: packetIn

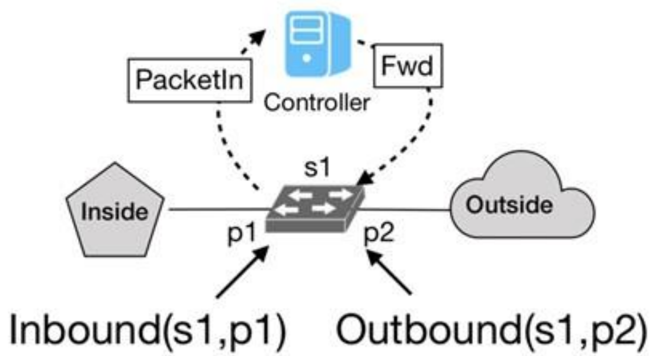
SrcIP	DstIP	InPort
10.0.0.1	10.0.0.2	1
10.0.0.3	10.0.0.2	2
10.0.0.4	10.0.0.5	1

fwd(IP, Port) :-
packetIn(SrcIP, DstIP, InPort),
IP=DstIP, InPort=Port.

Output: fwd

IP	Port
10.0.0.2	1
10.0.0.2	2
10.0.0.5	1

Example-guided synthesis



background knowledge

Input-output	
PacketIn 10.0.0.1 -> 172.217.11.46	Fwd 10.0.0.1, port 2
PacketIn 10.0.0.1 -> 172.217.11.46	Fwd 172.217.11.46, port 1

examples

Facon the synthesizer

```
Fwd(swi, dstIP, srcIP, prt) :- PacketIn(swi, srcIP, dstIP, prt),
   InBound(swi, prt).
Fwd(swi, srcIP, dstIP, prt) :- PacketIn(swi, srcIP, dstIP, prt2),
   InBound(swi, prt2), Outbound(swi, prt).
```

Symbolic Rules

Synthesis algorithm

1. Construct all valid NDlog rules
   - This helps prune the search space
   - Only search within the syntax-correct rule space

2. Construct candidate program

3. Verify the candidate against all examples

   If it satisfies, done

   Otherwise, go back to step 2

YouTube blames Pakistan network for 2-hour outage

Company appears to confirm reports that Pakistan Telecom was responsible for routing traffic according to erroneous Internet Protocols.

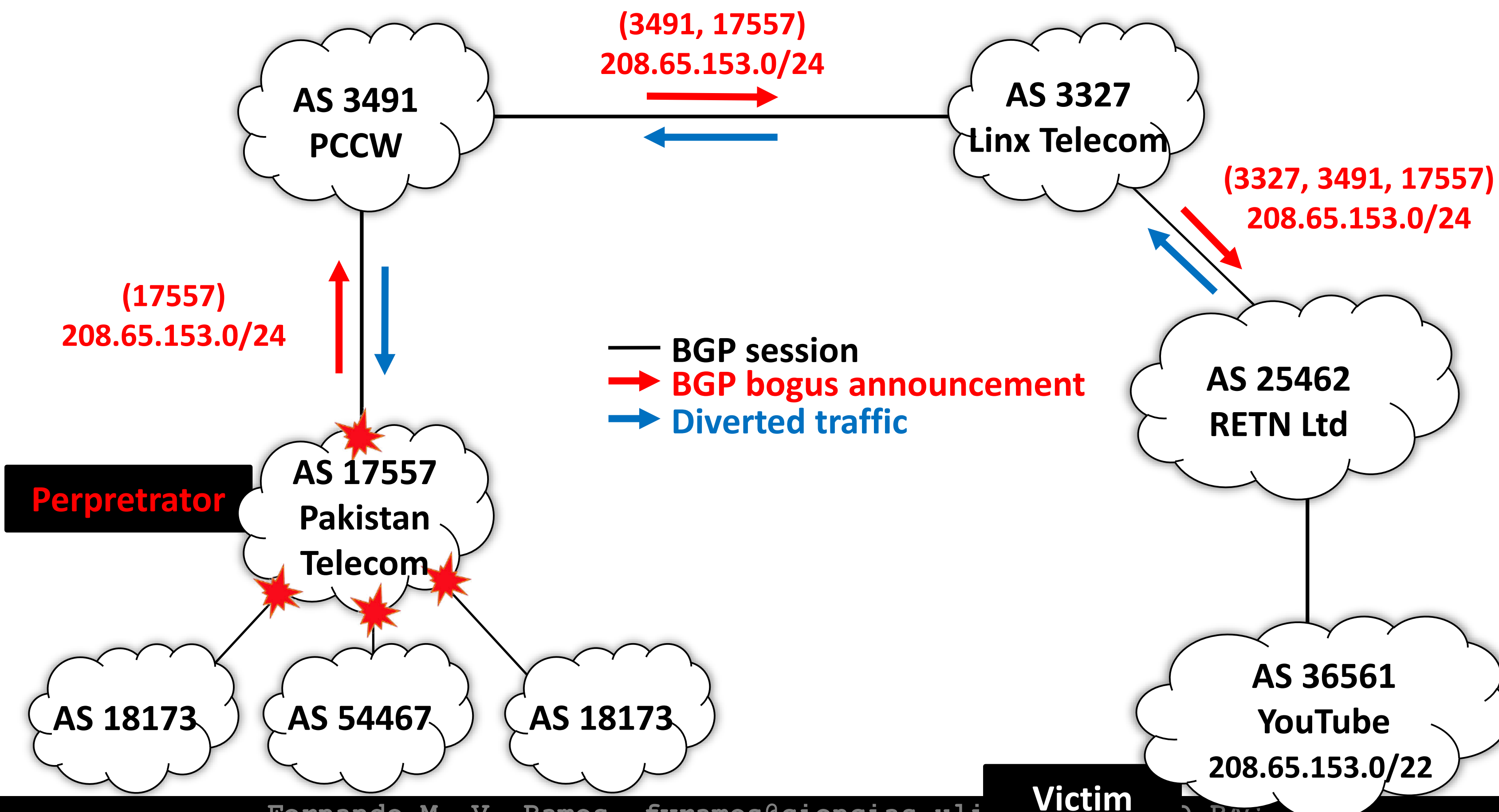
Connect with us

cnet Search CNET Reviews News Video How To Deals US Edition

CNET › Tech Culture ›
How Pakistan knocked YouTube offline (and how to make sure it never happens again)

How Pakistan knocked YouTube offline (and how to make sure it never happens again)

YouTube becoming unreachable isn't the first time that Internet addresses were hijacked. But if it spurs interest in better security, it may be the last.



Lecture 27/5: network security and censorship

- **Mandatory (one of these two)**
  - S. Goldberg et al., How Secure are Secure Interdomain Routing Protocols?, SIGCOMM, 2011

  The authors ask if secure network protocols are really secure -- guess the answer.
  - V. Giotsas et al., Inferring BGP Blackholing Activity in the Internet, IMC 2017

  The authors develop and evaluate a methodology to automatically detect BGP blackholing activity in the wild.
  - S. Burnett and N. Feamster, Encore: Lightweight Measurement of Web Censorship with Cross-Origin Requests, SIGCOMM 2015

  Encore is a (very controversial) system that enables detection of Internet censorship at large scale

- [Optional]
  - P. Gill et al., A survey of interdomain routing policies, ACM CCR, 2014

  A short survey on interdomain network policies
  - S. Goldberg, Why is it taking so long to secure Internet routing?, Communications of the ACM, 2014

  Sharon Goldberg, an expert on BGP security, asks why is it taking so long to secure routing
  - D. Kreutz et al., Towards Secure and Dependable Software-Defined Networks, HotSDN 2013

  The authors investigate the security and dependability of SDN