Real Time Network Policy Checking using Header Space Analysis

Peyman Kazemian[†*], Michael Chang[†], Hongyi Zeng[†],
George Varghese[‡], Nick McKeown[†], Scott Whyte[§]
[†]*Stanford University,* [‡]*UC San Diego & Microsoft Research,* [§]*Google Inc.*
[†]{kazemian, mchang91, hyzeng, nickm}@stanford.edu, [‡]varghese@cs.ucsd.edu, [§]swhyte@google.com

Abstract

Network state may change rapidly in response to customer demands, load conditions or configuration changes. But the network must also ensure correctness conditions such as isolating tenants from each other and from critical services. Existing policy checkers cannot verify compliance in real time because of the need to collect "state" from the entire network and the time it takes to analyze this state. SDNs provide an opportunity in this respect as they provide a logically centralized view from which every proposed change can be checked for compliance with policy. But there remains the need for a fast compliance checker.

Our paper introduces a real time policy checking tool called *NetPlumber* based on Header Space Analysis (HSA) [8]. Unlike HSA, however, NetPlumber *incrementally* checks for compliance of state changes, using a novel set of conceptual tools that maintain a dependency graph between rules. While NetPlumber is a natural fit for SDNs, its abstract intermediate form is conceptually applicable to conventional networks as well. We have tested NetPlumber on Google's SDN, the Stanford backbone and Internet 2. With NetPlumber, checking the compliance of a typical rule update against a single policy on these networks takes 50-500μs on average.

1 Introduction

Managing a network today manually is both cumbersome and error-prone. For example, network administrators must manually login to a switch to add an access-control rule blocking access to a server. In a recent survey [15], network administrators reported that configuration errors are very common in their networks.

The problem is that *several* entities can modify the forwarding rules: in addition to manual configuration, distributed protocols (e.g. OSPF, spanning tree, BGP) write entries into forwarding tables. There is no single location where all of the state is observable or controllable, leaving network administrators to use ad-hoc tools like ping and traceroute to indirectly probe the current state of the forwarding rules.

Recently, there has been growing interest in automating network control using software-defined networks (SDNs). SDN separates the control plane from the forwarding plane; a well-defined interface such as OpenFlow [11] lets the control plane write <*match, action*> rules to switches. The controller *controls* the forwarding state because it decides which rules to write to the switches; and it *observes* the forwarding state because it was the sole creator. SDNs therefore present an opportunity to automate the verification of correct forwarding behavior. This is the premise of recent work on automatic analysis of forwarding state for SDNs [8, 10, 14]. The basic idea is that if we can analyze the forwarding state—either as it is written to switches, or after it has been written—then we can check against a set of invariants/policies and catch bugs *before* or *soon after* they take place.

Our paper describes a verification tool called NetPlumber for SDNs and conventional networks. In SDNs, NetPlumber sits in line with the control plane, and observes state changes (e.g. OpenFlow messages) between the control plane and the switches (Figure 1). NetPlumber checks every event, such as installation of a new rule, removal of a rule, port or switch up and down events, against a set of policies and invariants. Upon detecting a violation, it calls a function to alert the user or block the change. In conventional networks, NetPlumber can get state change notifications through SNMP traps or by frequently polling switches. Our evaluations use a large SDN (Google WAN) and two medium sized IP networks (Internet2 and the Stanford Network).

NetPlumber can detect simple invariant violations such as loops and reachability failures. It can also check more sophisticated policies that reflect the desires of human operators such as: "Web traffic from A to B should never pass through waypoints C or D between 9am and 5pm." Our NetPlumber prototype introduces a new formal language (similar to FML [6]) to express policy checks, and is fast enough to perform real-time checks each time a controller adds a new rule. In experiments with the Stanford backbone, Google's WAN, and Internet2's backbone, NetPlumber typically verifies a rule change in less than 1ms, and a link-up or link-down event in a few seconds.

[*]Peyman Kazemian was an intern at Google while doing this work.

NetPlumber's speed easily exceeds the requirements for an enterprise network where configuration state changes infrequently—say once or twice per day. But in modern multi-tenant data centers, fast programmatic interfaces to the forwarding plane allow control programs to rapidly change the network configuration - perhaps thousands of times per second. For example, we may move thousands of virtual machines (VMs) to balance load, with each change requiring a tenant's virtual network to be reconfigured.

NetPlumber builds on our earlier work on Header Space Analysis (HSA) [8]. HSA models networks using a geometric model that is much easier to reason about than the vendor-specific interfaces on switches and routers. NetPlumber improves upon HSA in two ways. First, by running HSA checks incrementally, NetPlumber enables real-time checking of updates; this in turn can prevent bugs from occurring. Second, NetPlumber provides a flexible way to express and check complex policy queries without writing new ad hoc code for each policy check, as was required by HSA.

The four contributions of this paper are:

1. *NetPlumber* (section 3): NetPlumber is our real-time policy checking tool with sub-millisecond average run time per rule update.
2. *Flexible Policy Query Mechanism* (section 4): NetPlumber introduces a flexible way to express complex policy queries in an extensible, regular-expression-based language called FlowExp.
3. *Distributed NetPlumber* (section 5): We show how to scale NetPlumber to large networks using a distributed implementation.
4. *Evaluation at Scale* (section 6): We evaluate NetPlumber on three production networks, including Google's global WAN carrying inter-datacenter traffic.

2 Header Space Analysis

NetPlumber uses HSA [8] as a foundation. HSA provides a uniform, vendor-independent and protocol-agnostic model of the network using a geometric model of packet processing. A header is a point (and a flow is a region) in a $\{0,1\}^L$ space, called the *header space*, where each bit corresponds to one dimension of this space and L is an upper bound on header length (in bits). Networking boxes are modeled using a *Switch Transfer Function* T, which transforms a header h received on input port p to a set of packet headers on one or more output ports: $T:(h,p)\rightarrow\{(h_1,p_1),(h_2,p_2),...\}$.

Each transfer function consists of an ordered set of *rules* R. A typical rule consists of a set of physical input ports, a *match* wildcard expression, and a set of actions to be performed on packets that match the wildcard expression. Examples of actions include: forward to a port, drop, rewrite, encapsulate, and decapsulate. Network topology is modeled using a *Topology Transfer Function*, Γ. If port p_{src} is connected to p_{dst} using a link, then Γ will have a rule that transfers (h,p_{src}) to (h,p_{dst}).

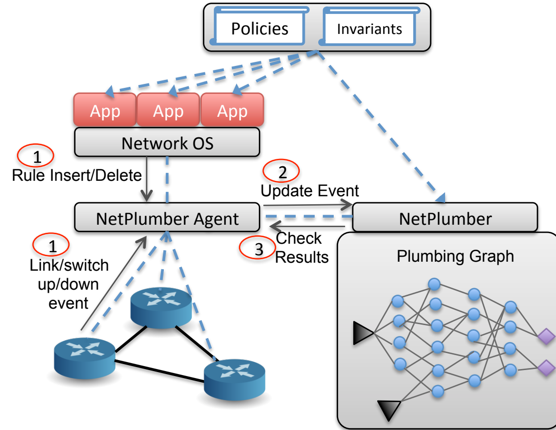

Figure 1: Deploying NetPlumber as a policy checker in SDNs.

HSA computes reachability from source A, via switches $X,Y,...$ to destination B as follows. First, create a header space region at A representing the set of all possible packets A could send: the all-wildcard flow with L wildcard bits and covering the entire L-dimensional space. Next, apply switch X's transfer function to the all-wildcard flow to generate a set of regions at its output ports, which in turn are fed to Y's switch transfer function. The process continues until a subset of the flows that left A reach B. While the headers may have been transformed in the journey, the original headers sent by A can be recovered by applying the *inverse transfer function*. Despite considerable optimization, the Python-based implementation called Hassel described in [8] requires tens of seconds to compute reachability.

3 NetPlumber

NetPlumber is much faster than Hassel at update time because instead of recomputing all the transformations each time the network changes, it incrementally updates only the portions of those transfer function results affected by the change. Underneath, NetPlumber still uses HSA. Thus, it inherits from HSA the ability to verify a wide range of policies—including reachability between ports, loop-freedom, and isolation between groups—while remaining protocol agnostic.

Figure 1 shows NetPlumber checking policies in an SDN. An agent sits between the control plane and switches and sends every state update (installation or removal of rules, link up or down events) to NetPlumber which in turn updates its internal model of the network; if a violation occurs, NetPlumber performs a user-defined action such as removing the violating rule or notifying

the administrator.

The heart of NetPlumber is the *plumbing graph* which captures all possible paths of flows[1] through the network. Nodes in the graph correspond to the *rules* in the network and directed edges represent the *next hop dependency* of these rules:

- A rule is an OpenFlow-like `<match, action>` tuple where the action can be `forward`,[2] `rewrite`, `encapsulate`, `decapsulate`, etc.
- Rule A has a next hop dependency to rule B if 1) there is a physical link from rule A's box to rule B's box; *and* 2) the domain of rule B has an intersection with range of rule A. The domain of a rule is the set of headers that match on the rule and the range is the region created by the `action` transformation on the rule's domain.

Initialization: NetPlumber is initialized by examining the forwarding tables to build the plumbing graph. Then it computes reachability by computing the set of packets from source port s, that can reach destination port d by injecting an"all-wildcard flow" at s and propagating it along the edges of the plumbing graph. At each rule node, the flow is filtered by the `match` part of the rule and then transformed by the `action` part of the rule. The resulting flow is then propagated along the outgoing edges to the next node. The portion of the flow, if any, that reaches d is the set of all packets from s that can reach d. To speed up future calculations, whenever a rule node transforms a flow, it remembers the flow. This caching lets NetPlumber quickly update reachability results every time a rule changes.

Operation: In response to insertion or deletion of rules in switches, NetPlumber adds or removes nodes and updates the routing of flows in the plumbing graph. It also re-runs those policy checks that need to be updated.

3.1 The Plumbing Graph

The nodes of the plumbing graph are the forwarding rules, and directed edges represent the next-hop dependency of these rules. We call these directed edges *pipes* because they represent possible paths for flows. A pipe from rule a to b has a *pipe filter* which is the intersection of the range of a and the domain of b. When a flow passes through a pipe, it is filtered by the pipe filter. Conceptually the pipe filter represents all packet headers at the output of rule a that can be processed by b.

A rule node corresponds to a rule in a forwarding table in some switch. Forwarding rules have priorities; when a packet arrives to the switch it is processed by the highest priority matching rule. Similarly, the plumbing graph needs to consider rule priorities when deciding which rule node will process a flow. For computational efficiency, each rule node keeps track of higher priority rules in the same table. It calculates the domain of each higher priority rule, subtracting it from its own domain. We refer to this as *intra-table dependency* of rules.

Figure 2 shows an example network and its corresponding plumbing graph. It consists of 4 switches, each with one forwarding table. For simplicity, all packet headers are 8 bits. We will use this example though the rest of this section.

Let's briefly review how the plumbing graph of Figure 2 is created: There is a pipe from rule 1 in table 1 (rule 1.1) to rule 2 in table 2 (rule 2.2) because (a) ports 2 and 4 are connected and (b) the range of rule 1.1 (1010xxxx) and the domain of rule 2.2 (10xxxxxx) has a non-empty intersection (pipe filter: 1010xxxx). Similarly there is a pipe from rule 2.2 to rule 4.1 because (a) ports 5 and 8 are connected and (b) the range of rule 2.2 (111xxxxx) and the domain of rule 4.1 (xxxxx010) has a non-empty intersection (pipe filter: 111xx010). Also rule 1.1 has an intra-table influence on rule 1.3 because their domains and input port sets have a non-empty intersection (intersecting domain: 1010xxxx, port: 1). The rest of this plumbing graph is created in similar fashion.

3.2 Source and Sink Nodes

NetPlumber converts policy and invariants to equivalent reachability assertions. To compute reachability, it inserts flow from the source port into the plumbing graph and propagates it towards the destination. This is done using a "flow generator" or *source node*. Just like rule nodes, a source node is connected to the plumbing graph using directed edges (pipes), but instead of processing and forwarding flows, it generates flow.

Continuing our example, we compute reachability between port 1 and 10 in Figure 3 by connecting a source node, generating the all-wildcard flow, to port 1. We have also connected a special node called a *probe* node to port 10. Probe nodes will be discussed in the next section. The flow generated by the source node first reaches rules 1.1, 1.2 and 1.3. Rule 1.1 and 1.2 are not affected by any higher priority rules and don't rewrite flows. Therefore the input flow is simply forwarded to the pipes connecting them to rule 2.2 (i.e. 1010xxxx and 10001xxx flows reach rule 2.2). However rule 1.3 has an intra-table dependency to rule 1.1 and 1.2. This means that from the incoming 10xxxxxx flow, only 10xxxxxx − (1010xxxx ∪ 10001xxx) should be processed by rule 1.3. The remainder has already been processed by higher priority rules. Rule 1.3 is a simple forward rule and will forward the flow, unchanged, to rule 3.1. However, when this flow passes through the pipe filter between rule 1.3 and

[1]In what follows, a flow corresponds to any region of header space.
[2]A drop rule is a special case of forward rule with empty set of output ports.

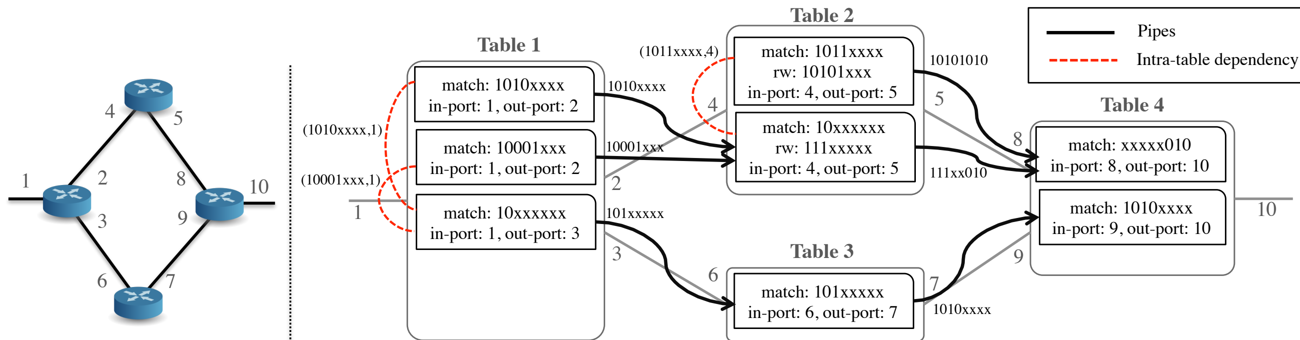

Figure 2: Plumbing graph of a simple network consisting of 4 switches each with one table. Arrows represent pipes. Pipe filters are shown on the arrows. Dashed lines indicate intra-table dependency of rules. The intersecting domain and input port is shown along the dashed lines.

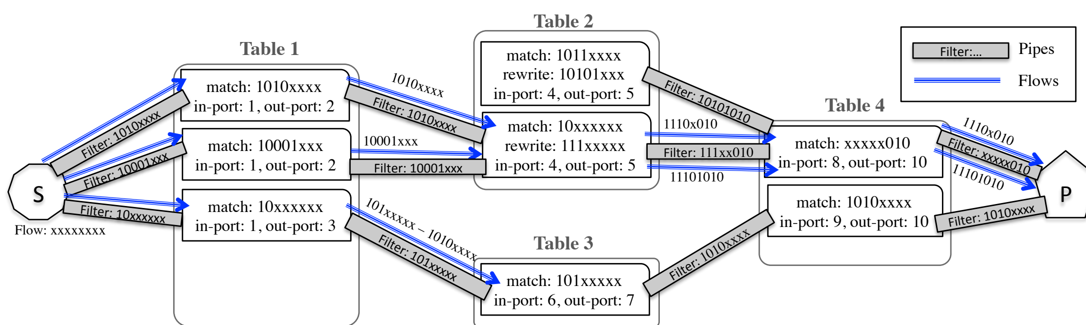

Figure 3: Finding reachability between S and P. Source node S is generating all-wildcard flow and inserting it into the plumbing graph. The solid lines show the path of flow from the source to the destination. Flow expressions are shown along the flows.

3.1 (101xxxxx), it shrinks to 101xxxxxx − 1010xxxx.[3]

The flows which reach rule 2.2 continue propagating through the plumbing graph until they reach the probe node (P), as depicted in Figure 3. However the other flow that has reached rule 3.1 does not propagate any further as it cannot pass through the pipe connecting rule 3.1 to rule 4.2. This is because the intersection of the flow (101xxxxxx − 1010xxxx = 1011xxxx) and pipe filter (1010xxxx) is empty.

Sink Nodes: Sink nodes are the dual of source nodes. A sink node absorbs flows from the network. Equivalently, a sink node generates "sink flow" which traverses the plumbing graph in the reverse direction. When reaching a rule node, a sink flow is processed by the inverse of the rule.[4] Reachability can be computed using sink nodes: if a sink node is placed at the destination port D, then the sink flow at source port S gives us the set of packet headers from S that will reach D. Sink nodes do not increase the expressive power of NetPlumber; they only simplify or optimize some policy checks (see section 4).

3.3 Probe Nodes

A fourth type of node called a *probe node* is used to check policy or invariants. Probe nodes can be attached to appropriate locations of the plumbing graph, and can be used to check the path and header of the received flows for violations of expected behavior. In section 4, we discuss how to check a policy using a source (sink) and probe node. As a simple example, if in our toy example of Figure 2 the policy is "port 1 and 10 can only talk using packets matching xxxxx010", then we place a source node at port 1 (S), a probe node at port 10 (P) and configure P to check whether all flows received from S match xxxxx010 (Figure 3).

Probe nodes can be of two types: *source probe nodes* and *sink probe nodes*. The former check constraints on flows generated by source nodes, and the latter check flows generated by sink nodes. We refer to both as probe nodes.

3.4 Updating NetPlumber State

As events occur in the network, NetPlumber needs to update its plumbing graph and re-route the flows. There are 6 events that NetPlumber needs to handle:

Adding New Rules: When a new rule is added, NetPlumber first creates pipes from the new rule to all po-

[3] [10xxxxxx − (1010xxxx ∪ 10001xxx)] ∩ 101xxxxx = 101xxxxx − 1010xxxx.

[4] The inverse of a rule gives us all input flows that can generate a given flow at the output of that rule [8].

tential next hop rules, and from all potential previous hop rules to the new rule. It also needs to find all intra-table dependencies between the new rule and other rules within the same table. In our toy example in Figure 4, a new rule is added at the 2nd position of table 1. This creates three new pipes to rules 2.1, 2.2 and the source node, and one intra-table dependency for rule 1.4.

Next, NetPlumber updates the routing of flows. To do so, it asks all the previous hop nodes to pass their flows on the *newly created* pipes. The propagation of these flows then continues normally through the plumbing graph. If the new rule has caused any intra-table dependency for lower priority rules, we need to update the flows passing through those lower priority rules by subtracting their domain intersection from the flow. Back to the example in Figure 4, after adding the new rule, the new flows highlighted in bold propagate through the network. Also, the intra-table dependency of the new rule on rule 1.4 is subtracted from the flow received by rule 1.4. This shrinks the flow to the extent that it cannot pass through the pipe connecting it to rule 3.1 (empty flow on the bottom path).

Deleting Rules: Deleting a rule causes all flows which pass through that rule to be removed from the plumbing graph. Further, if any lower priority rule has any intra-table dependency on the deleted rule, the effect should be added back to those rules. Figure 5 shows the deletion of rule 1.1 in our toy example. Note that deleting this rule causes the flow passing through rule 1.3 to propagate all the way to the probe node, because the influence of the deleted rule is now added back.

Link Up: Adding a new link to the network may cause additional pipes to be created in the plumbing graph, because more rules will now have physical connections between them (first condition for creating a pipe). The nodes on the input side of these new pipes must propagate their flows on the new pipes, and then through the plumbing graph as needed. Usually adding a new link creates a number of new pipes, making a Link Up event slower to process than a rule update.

Link Down: When a link goes down, all the pipes created on that link are deleted from the plumbing graph, which in turn removes all the flows that pass through those pipes.

Adding New Tables: When a new table (or switch) is discovered, the plumbing graph remains unchanged. Changes occur only when new rules are added to the new table.

Deleting Tables: A table is deleted from NetPlumber by deleting all the rules contained in that table.

3.5 Complexity Analysis

The complexity of NetPlumber for the addition of a single rule is $O(r + spd)$, where r is the number of entries in each table and s is the number of source (sink) nodes attached to the plumbing graph (which is roughly proportional to the number of policies we want to check), p is the number of pipes to and from the rule and d is the diameter of the network.

The run time complexity arises as follows: when a new rule is added, we need to first find intra-table dependencies. These require intersecting the `match` portion of the new rule with the `match` of all the other rules in the same table. We also need to create new pipes by doing $O(r)$ intersections of the range of the new rule with the domain of rules in the neighboring tables ($O(r)$ such rules).

Next, we need to route flows. Let us use the term *previous nodes* to denote the set of rules which have a pipe to the new rule. First, we need to route the flows at previous nodes to the new rule. There are $O(s)$ flows on each of these previous nodes because each source (sink) node that is connected to NetPlumber can add a flow. We need to pass these flows through $O(p)$ pipes to route them to the new rule. This is $O(sp)$ work. With a *linear fragmentation*[5] argument similar to [8], there will be $O(s)$ flows that will survive this transformation through the pipes [6] (and not $O(sp)$). The surviving flows will be routed in the same manner through the plumbing graph, requiring the same $O(sp)$ work at each node in the routing path. Since the maximum path length is the diameter d, the overall run time of this phase is $O(spd)$.

We also need to take care of intra-table dependencies between this rule and lower priority rules, and subtract the domain intersection from the flows received by lower priority rules. This subtraction is done lazily and is therefore much faster than flow routing; hence we ignore its contribution to overall run time.

4 Checking Policies and Invariants

A probe node monitors flows received on a set of ports. In the plumbing graph, it is attached to the output of all the rules sending out flows on those ports. Each probe node is configured with a *filter* flow expression and a *test* flow expression. A flow expression or *flowexp* for short, is a regular expression specifying a set of conditions on the path and the header of the flows. The *filter* flowexp constrains the set of flows that should be examined by the probe node, and the *test* flowexp is the constraint that

[5]This assumption states that if we have R flows at the output of a transfer function, and we apply these flow to the next hop transfer functions with R rules per transfer function, we will get cR flows at the output where $c << R$ is a constant. This assumption is based on the observation that flows are routed end-to-end in networks. They are usually aggregated, and not randomly fragmented in the core of the network.

[6]An alternate way to reach the same conclusion is as follows: the new rule, after insertion will look like any other rule in the network, and should on average have $O(s)$ flows.

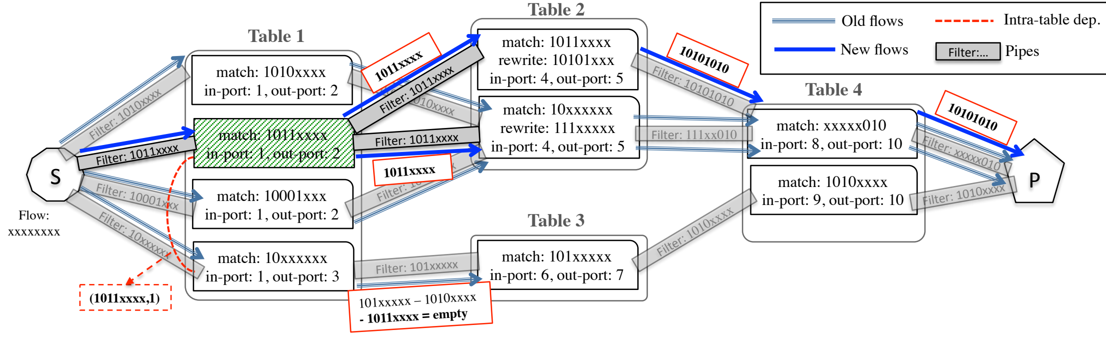

Figure 4: Adding rule 1.2 (shaded in green) to table 1. As a result a) 3 pipes are created connecting rule 1.2 to rule 2.1 and 2.2 and to the source node. b) rule 1.4 will have an intra-table dependency to the new rule (1011xxxx,1). c) The flows highlighted in bold will be added to the plumbing graph. Also the flow going out of rule 1.4 is updated to empty.

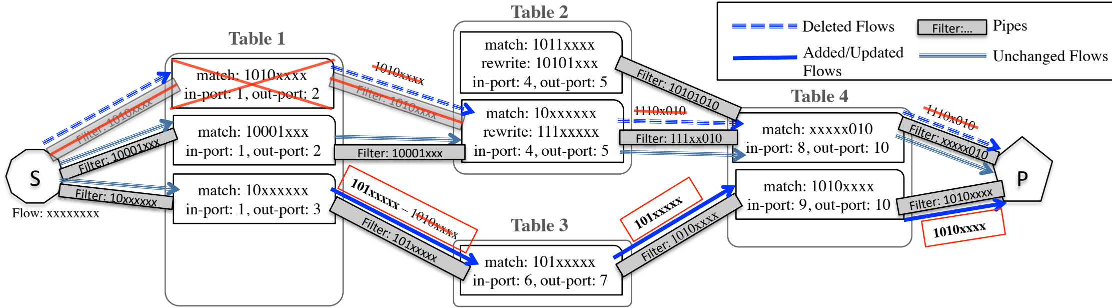

Figure 5: Deleting rule 1.1 in table 1 causes the flow which passes through it to be removed from the plumbing graph. Also since the intra-table dependency of rule 1.3 to this rule is removed, the flow passing through 1.3 through the bottom path is updated.

is checked on the matching flows. Probe nodes can be configured in two modes: *existential* and *universal*. A probe fires when its corresponding predicate is violated. An existential probe fires if none of the flows examined by the probe satisfy the test flow expression. By contrast, a universal probe fires when a single flow is received that does *not* satisfy the test constraint. More formally:

(Universal) $\forall\{f \mid f \sim filter\} : f \sim test$. All flows f which satisfy the filter expression, satisfy the test expression as well.

(Existential) $\exists\{f \mid f \sim filter\} : f \sim test$. There exist a flow f that satisfies both the filter and test expressions.

Using flow expressions described via the flowexp language, probe nodes are capable of expressing a wide range of policies and invariants. Section 4.1 will introduce the flowexp language. Sections 4.2 and 4.3 discuss techniques for checking for loops, black holes and other reachability-related policies.

4.1 Flowexp Language

Each flow at any point in the plumbing graph, carries its complete history: it has a pointer to the corresponding flow at the previous hop (node). By traversing these pointers backward, we can examine the entire history of the flow and all the rules that have processed this flow along the path. The flow history always begins at the generating source (or sink) node and ends at the probe node checking the condition.

$Constraint$	$\rightarrow$	`True` | `False` | `!`$Constraint$
	|	($Constraint$ | $Constraint$)
	|	($Constraint$ & $Constraint$)
	|	$PathConstraint$
	|	$HeaderConstraint$;
$PathConstraint$	$\rightarrow$	`list(`$Pathlet$`)`;
$Pathlet$	$\rightarrow$	`Port Specifier` [$p \in \{P_i\}$]
	|	`Table Specifier` [$t \in \{T_i\}$]
	|	`Skip Next Hop [.]`
	|	`Skip Zero or More Hops [.*]`
	|	`Beginning of Path [^]` `(Source/Sink node)`
	|	`End of Path [$]` `(Probe node)`;
$HeaderConstraint$	$\rightarrow$	$H_{received} \cap H_{constraint} \neq \phi$
	|	$H_{received} \subset H_{constraint}$
	|	$H_{received} == H_{constraint}$;

Table 1: Flowexp language grammar

Flowexp is a regular expression language designed to check constraints on the history of flows received by probe nodes. Table 1 shows the grammar of flowexp in a standard BNF syntax. Flowexp consists of logical operations (i.e. *and*, *or* and *not*) on constraints enforced on the *Path* or *Header* of flows received on a probe node.

A *PathConstraint* is used to specify constraints on the

path taken by a flow. It consists of an ordered list of *pathlets* that are checked sequentially on the path of the flow. For example a flow that originates from source S, with the path $S \rightarrow A \rightarrow B \rightarrow C \rightarrow P$ to probe P, will match on flowexp " $\hat{}(p = A)$", because port A comes immediately after the source node. It also matches on "$(p = A).(p = C)$" because the flow passes through exactly one intermediate port from A to C.

A *HeaderConstraint* can check if 1) The received header has any intersection with a specified header; this is useful when we want to ensure that some packets of a specified type can reach the probe. 2) The received header is a subset of a specific header; this is useful when we wish to limit the set of headers that can reach the probe. 3) The received header is exactly equal to a specified header; this is useful to check whether the packets received at the probe are exactly what we expect.

Since flowexp is very similar to (but much simpler than) standard regular expression language, any standard regexp checking technique can be used at probe nodes.

4.2 Checking Loops and Black Holes

As flows are routed through the plumbing graph, each rule by default (i.e., without adding probe nodes for this purpose) checks received flows for loops and black holes. To check for a loop, each rule node examines the flow history to determine if the flow has passed through the current table before. If it has, a loop-detected callback function is invoked[7].

Similarly, a black hole is automatically detected when a flow is received by a non-drop-rule R that cannot pass through any pipes emanating from R. In this case, a black-hole-detected callback function is invoked.

4.3 Checking Reachability Policies

In this section, we describe how to express reachability-related policies and invariants such as the isolation of two ports, reachability between two ports, reachability via a middle box and a constraint on the maximum number of hops in a path. We express and check for such reachability constraints by attaching one or more source (or sink) nodes and one or more probe nodes in appropriate locations in the plumbing graph. The probe nodes are configured to check the appropriate filter and test flowexp constraints as shown below.

Basic Reachability Policy: Suppose we wish to ensure that a server port S should not be reachable from guest machine ports $\{G_1, ...G_k\}$.

Solution using a source probe: Place a source node that generates a wildcarded flow at each of the guest ports. Next, place a source probe node on port S and configure it to check for the flow expression: $\forall f : f.path \sim\ ![\hat{}\,(p \in \{G_1, ...G_k\})]$ - i.e., a *universal* probe with no filter constraint and a test constraint that checks that the source node in the path is not a guest port.

If, instead, the policy requires S to be reachable from $\{G_1, ...G_k\}$, we could configure the probe node as follows: $\exists f : f.path \sim [\hat{}\,(p \in \{G_1, ...G_k\})]$. Intuitively, this states that there exists some flow that can travel from guest ports to the server S. Note that the server S is not specified in the flow expression because the flow expression is placed at S.

Dual Solution using a sink probe: Alternately, we can put a sink node at port S and a sink probe node in each of the G_i ports. We also configure the probes with Flowexp $\forall f : f.path \sim [\hat{}\,(p \in \{S\})]$.

Reachability via a Waypoint: Next, suppose we wish to ensure that all traffic from port C to port S must pass through a "waypoint" node M.

Solution: Put a source node at C that generates a wildcarded flow and a probe node at S. Configure the probe node with the flow expression: $\forall\{f \mid f.path \sim [\hat{}\,(p \in \{C\})]\} \ :\ f.path \sim [\hat{}\,.^*(t = M)]$. This is a universal probe which filters flows that originate from C and verifies that they pass through the waypoint M.

Path length constraint: Suppose we wish to ensure that no flow from port C to port S should go through more than 3 switches. This is a policy that was desired for the Stanford network for which we found violations. The following specification does the job assuming that each switch has one table.

Solution: Place a probe at S and a source node at C as in the previous example. Configure the probe node with the following constraint: $\forall\{f \mid f.path \sim [\hat{}\,(p \in \{C\})]\} \ :\ f.path \sim [\ \hat{}.\$ \mid \hat{}..\$ \mid \hat{}...\$\ ]$. The filter expression ensures that the check is done only for flows from C, and the test expression only accepts a flow if it is one, two or three hops away from the source.

Source probes versus Sink probes: Roughly speaking, if a policy is checking something at the destination regardless of where the traffic comes from, then using sink probes is more efficient. For example, suppose a manager wishes to specify that all flows arriving at a server S pass through waypoint M. Using source probes would require placing one source probe at every potential source. This can be computationally expensive as the run time of NetPlumber grows linearly with number of source or sink nodes. On the other hand, if the policy is about checking a condition for a particular source – such as computer C should be able to communicate with all other nodes – then using a source probe will be more efficient. Intuitively, we want to minimize the amount of flow in the plumbing graph required to check a given policy, as generating flow is computationally expensive.

[7]The callback function can optionally check to see if the loop is infinite or not; an algorithm to check for infinite loops is described in [8].

4.4 Policy translator

So far we have described a logical language called flowexp which is convenient for analysis and specifying precisely how flows are routed within the network. Flowexp is, however, less appropriate as a language for network managers to express higher level policy. Thus, for higher level policy specification, we decided to reuse the policy constructs proposed in the Flow-based Management Language (FML) [6], a high-level declarative language for expressing network-wide policies about a variety of different management tasks. FML essentially allows a manager to specify predicates about groups of users (e.g., faculty, students), and specifies which groups can communicate. FML also allows additional predicates on the *types* of communication allowed such as the need to pass through waypoints.

Unfortunately, the current FML implementation is tightly integrated with an OpenFlow controller, and so cannot be easily reused in NetPlumber. We worked around this by encoding a set of constructs inspired by FML in Prolog. Thus, network administrators can use Prolog as the frontend language to declare various bindings inspired by FML, such as hosts, usernames, groups and addresses. Network administrators can also use Prolog to specify different policies. For example, the following policy describes *1)* the `guest` and `server` groups, and *2)* a policy: "Traffic should go through firewall *if* it flows from a guest to a server".

```
guest(sam).
guest(michael).
server(webserver).
waypoint(HostSrc, HostDst, firewall):-
    guest(HostSrc),
    server(HostDst).
```

We have written a translator that converts such high level policy specifications written in Prolog to 1) the placement of source nodes, 2) the placement of probe nodes, and 3) the filter and test expressions for each probe node. In the example above, the translator generates two source nodes at Sam and Michael's ports and one probe node at the web server's port. The `waypoint` keyword is implemented by flowexp: *.*(t=firewall)*.

The output of the translator is, in fact, a C++ struct that lists all source, sink, and probe nodes. The source probes and sink probes are encoded in flowexp syntax using ASCII text. Finally, NetPlumber translates flowexp into C++ code that it executes.

Note that because FML is not designed to declare path constraints that can be expressed in flowexp, we found it convenient to make the translator extensible. For example, two new policy constructs we have built-in beyond the FML-inspired constructs are "at most N hops" and "immediately followed by"—but it is easy to add further constructs.

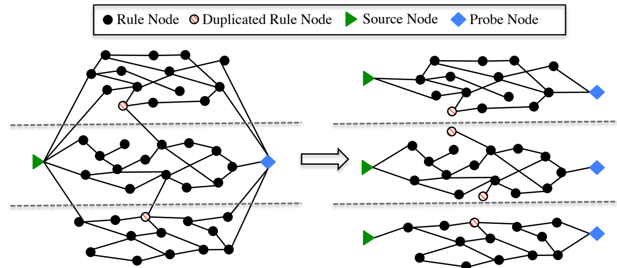

Figure 6: A typical plumbing graph consists of clusters of highly dependent rules corresponding to FECs in network. There may be rules whose dependency edges cross clusters. By replicating those rules, we can create clusters without dependencies and run each cluster as an isolated NetPlumber instance running on a different machine.

5 Distributed NetPlumber

NetPlumber is memory-intensive because it maintains considerable data about every rule and every flow in the plumbing graph. For very large networks, with millions of rules and a large number of policy constraints, NetPlumber's memory requirements can exceed that of a single machine. Further, as shown in section 3.5, the run time of NetPlumber grows linearly with the size of the tables. This can be potentially unacceptable for very large networks.

Thus, a natural approach is to run parallel instances of NetPlumber, each verifying a subset of the network and each small enough to fit into the memory of a single machine. Finally, a collector can be used to gather the check results from every NetPlumber instance and produce the final result.

One might expect to parallelize based on switches: i.e., each NetPlumber instance creates a plumbing graph for a subset of switches in the network (*vertical distribution*). This can address the memory bottleneck, but need not improve performance, as the NetPlumber instances can depend on each other. In the worst case, an instance may not be able to start its job unless the previous instance is done. This technique can also require considerable communication between different instances.

A key observation is that in every practical network we have seen, the plumbing graph looks like Figure 6: there are clusters of highly dependent rules with very few dependencies between rules in different clusters. This is caused by forwarding equivalence classes (FECs) that are routed end-to-end in the network with possible aggregation. The rules belonging to a forwarding equivalence class have a high degree of dependency among each other. For example, 10.1.0.0/16 subnet traffic might be a FEC in a network. There might be rules that further divide this FEC into smaller subnets (such as 10.1.1.0/24,

10.1.2.0/24), but there are very few rules outside this range that has any interaction with rules in this FEC (an exception is the default 0.0.0.0/0 rule).

Our distributed implementation of NetPlumber is based on this observation. Each instance of NetPlumber is responsible for checking a subset of rules that belong to one cluster (i.e. a FEC). Rules that belong to more than one cluster will be replicated on all the instances they interact with (see Figure 6). Probe nodes are replicated on all instances to ensure global verification. The final probe result is the aggregate of results generated by all the probes—i.e., all probe nodes should meet their constraints in order for the constraint to be verified. The instances do not depend on each other and can run in parallel. The final result will be ready after the last instance is done with its job.

The run time of distributed NetPlumber, running on n instances for a single rule update, is $O(m_{avg}(r/n + spd/m))$ where m is the number of times that rule get replicated and m_{avg} is the average replication factor for all rules. This is because on each replica, the size of tables are $O(m_{avg}r/n)$ and the number of pipes to a rule that is replicated m times is $O(m_{avg}p/m)$. Note that if we increase n too much, most rules will be replicated across many instances ($m, m_{avg} \rightarrow n$,) and the additional parallelism will not add any benefit.

How should we cluster rules? Graph clustering is hard in general; however for IP networks we generated natural clusters heuristically as follows. We start by creating two clusters based on the IP address of the network we are working with; if the IP address of hosts in the network belong to subnet 10.1.0.0/16, create two clusters: one for rules that match this subnet, and one for the rest (i.e. 10.1.0.0/16 and 0.0.0.0/0 - 10.1.0.0/16 subnets). Next, divide the first cluster into two clusters based on bit 17 of the destination IP address. If one of the resulting clusters is much larger than the other, we divide the larger cluster based on the next bit in IP destination address. If two clusters are roughly the same size, we divide both clusters further. This process continues until division does not reduce cluster size further (because of replication) or the specified number of clusters is reached.

Note that while we introduced the plumbing graph originally to facilitate incremental computation, the plumbing graph also allows us to decompose the computation much more effectively than the naive decomposition by physical nodes.

6 Evaluation

In this section we evaluate the performance and functionality of our C++ based implementation[8] of NetPlumber on 3 real world networks: the Google inter-

[8]source code available at [5].

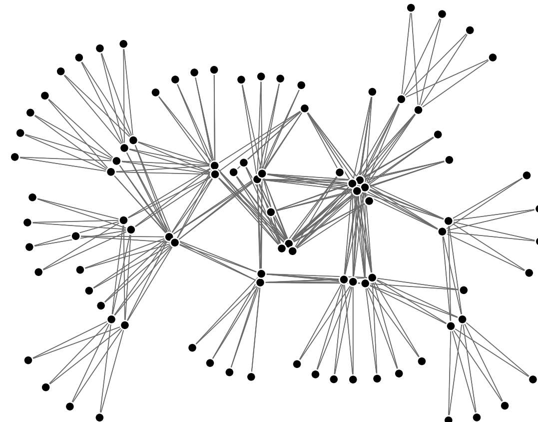

Figure 7: Google inter-datacenter WAN network.

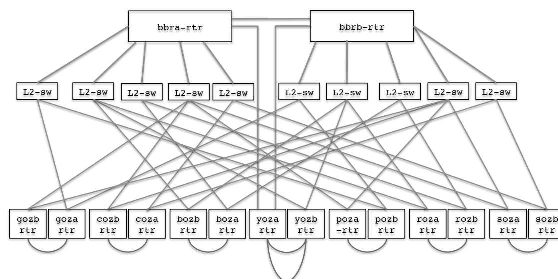

Figure 8: Stanford backbone network.

datacenter WAN, Stanford's backbone network and the Internet 2 nationwide network. All the experiments are run on Ubuntu machines, with 6 cores, hyper-threaded Intel Xeon processors, a 12MB L2-cache and 12GB of DRAM.

To feed the snapshot data from these networks into NetPlumber, we wrote 3 parsers capable of parsing Cisco IOS, Juniper Junos and OpenFlow dumps in protobuf [12] format. We used a json-rpc based client to feed this data into NetPlumber. NetPlumber has the json-rpc server capability and can receive and process updates from a remote source.

6.1 Our data set

Google WAN: This is a software-defined network, consisting of OpenFlow switches distributed across the globe. It connects Google data centers world-wide. Figure 7 shows the topology of this network. Overall there are more than 143,000 OpenFlow rules installed in these switches. Google WAN is one of the largest SDNs deployed today; therefore we stress-test NetPlumber on this network to evaluate its scalability.

Stanford University Backbone Network. With a population of over 15,000 students, 2,000 faculty, and five /16 IPv4 subnets, Stanford represents a mid-size enterprise network. There are 14 operational zone (OZ) Cisco routers connected via 10 Ethernet switches to 2

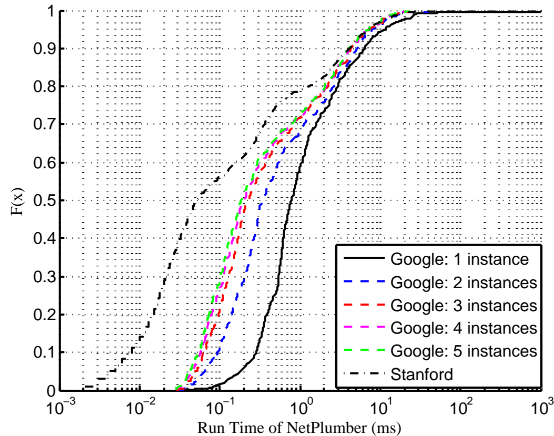

Figure 9: CDF of the run time of NetPlumber per update, when checking the all-pair reachability constraint in Google WAN with 1-5 instances and in Stanford backbone with a single instance.

#instances:	1	2	3	4	5	8
median (ms)	0.77	0.35	0.23	0.2	0.185	0.180
mean (ms)	5.74	1.81	1.52	1.44	1.39	1.32

Table 2: Average and median run time of distributed NetPlumber, checking all-pair connectivity policy on Google WAN.

backbone Cisco routers that in turn connect Stanford to the outside world (Figure 8). Overall, the network has more than 757,000 forwarding entries, 100+ VLANs and 1,500 ACL rules. Data plane configurations are collected through CLI. Stanford has made the entire configuration rule set public and it can be found in [5].

Internet2 is a nationwide backbone network with 9 Juniper T1600 routers and 100 Gb/s interfaces, supporting over 66,000 institutions in United States. There are about 100,000 IPv4 forwarding rules. All Internet2 configurations and FIBs of the core routers are publicly available [7], with the exception of ACL rules, which are removed for security reasons. We only use the IPv4 network of Internet 2 in this paper.

6.2 All-pair connectivity of Google WAN

As an internal, inter-datacenter WAN for Google, the main goal of Google WAN is to ensure connectivity between different data centers at all times. Therefore in our first experiment, we checked for the *all-pair connectivity* policy between all 52 leaf nodes (i.e. data center switches). We began by loading a snapshot of all the OpenFlow rules of Google WAN — taken at the end of July 2012 — into NetPlumber. NetPlumber created the initial plumbing graph in 33.39 seconds (an average per-rule runtime of $230\mu s$). We then attach one probe and one source node at each leaf of the network and set up the probes to look for one flow from each of the sources. If no probes fire, it means that all data centers are reachable from each other. The initial all-pair connectivity test took around 60 seconds. Note that the above run times, are for the *one-time initialization* of NetPlumber. Once NetPlumber is initialized, it can incrementally update check results much faster when changes occur. Note that the all-pair reachability check in Google WAN corresponds to 52^2 or more than 2600 pair-wise reachability checks.

Next, we used a second snapshot taken 6 weeks later. We found the diff of the two snapshots and applied them to simulate incremental updates. The diff includes both insertion and deletion of rules. Since we did not have timing information for the individual updates, we knew the *set* of updates in the difference but not the *sequence* of updates. So we simulated two different orders. In the first ordering, we applied all the rule insertions before the rule deletions. In the second ordering, we applied all deletions before all insertions.

As expected, the all-pair connectivity policy was maintained during the first ordering of update events, because new reachable paths are created before old reachable paths are removed. However the second ordering resulted in violations of the all-pair connectivity constraint *during* the rule deletion phase. Of course, this does not mean that the actual Google WAN had reachability problems because the order we simulated is unlikely to have been the actual order of updates. At the end of both orderings, the all-pair connectivity constraint was met.

NetPlumber was able to check the compliance of each insertion or deletion rule in an average time of 5.74ms with a median time of 0.77ms. The average run time is much higher than the median because there are a few rules whose insertion and deletion takes a long time (about 1 second). These are the default forwarding rules that have a large number of pipes and dependencies from/to other rules. Inserting and deleting default rules require significant changes to the plumbing graph and routing of flows. The solid line in Figure 9 shows the run time CDF for these updates.

To test the performance of distributed NetPlumber we repeated the same experiment in distributed mode. We simulated[9] the running of NetPlumber on $2-8$ machines and measured the update times (dashed lines in Figure 9). Table 2 summarizes the mean and median run times. This suggests that most of the benefits of distribution is achieved when the number of instances is 5. This is because in the plumbing graph of the Google WAN, there are about 5 groups of FECs whose rules do not influence

[9]To simulate, we run the the instances in serial on the same machine and collected the results from each run. For each rule insertion/deletion, we reported the run time as the maximum run time across all instances, because the overall job will be done only when the last instance is done.

each other. Trying to put these rules in more than 5 clusters will result in duplication of rules; the added benefit will be minimal.

6.3 Checking policy in Stanford network

Unlike the Google WAN, there are a number of reachability restrictions enforced in the Stanford network by different ACLs. Examples of such policies include isolation of machines belonging to a particular research group from the rest of the network, or limitation on the type of traffic that can be sent to a server IP address. For example, all TCP traffic to the computer science department is blocked except for those destined to particular IP addresses or TCP port numbers. In addition, there is a global reachability goal that every edge router be able to communicate to the outside world via the uplink of a specified router called `bbra_rtr`. Finally, due to the topology of the network, the network administrators desired that all paths between any two edge ports be no longer than 3 hops long to minimize network latency.

In this experiment we test all these policies. To do so, we connect 16 source nodes, one to each router in the plumbing graph. To test the maximum-3-hop constraint, we connected 14 probe nodes, one to each OZ router. We also placed a probe node at a router called `yoza_rtr` to check reachability policies at the computer science department. NetPlumber took 0.5 second to create the initial plumbing graph and 36 seconds to generate the initial check results. We found no violation of the reachability policies of the computer science department. However NetPlumber did detect a dozen un-optimized routes, whose paths take 4 hops instead of 3. We also found 10 loops, similar to the ones reported in [8][10].

We then tested the per-update run time of NetPlumber by randomly selecting 7% of rules in the Stanford network, deleting them and then adding them back. Figure 9 shows the distribution of the per-update run time. Here, the median runtime is $50\mu s$ and the mean is $2.34ms$. The huge difference between the mean and the median is due to a few outlier default rules which take a long time to get inserted and deleted into NetPlumber.

6.4 Performance benchmarking

The previous two experiments demonstrated the scalability and functionality of NetPlumber when checking actual policies and invariants of two production networks. However, the performance of NetPlumber depends on s, the number of sources in the network which is a direct consequences of the quantity and type of policies specified by each network. Thus it seems useful to have a metric that is per source node and even per policy, so we can extrapolate how run time will change as we add more independent policies, each of which require adding a new source node.[11] We provide such a unit run time benchmark for NetPlumber running on all three data sets: Google WAN, Stanford and Internet 2.

Network:	Google		Stanford		Internet 2	
Run Time	mean	median	mean	median	mean	median
Add Rule (ms)	0.28	0.23	0.2	0.065	0.53	0.52
Add Link (ms)	1510	1370	3020	2120	4760	2320

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.

To obtain this benchmark, we connect a single source node at one of the edge ports in the plumbing graph of each of our 3 networks. Then we load NetPlumber with 90% of the rules selected uniformly at random. Finally, we add the last 10% and measure the update time. We then repeated the same experiment by choosing links in the network that are in the path of injected flows, deleting them and then adding them back and measuring the time to incorporate the added link. The results are summarized in Table 3. As the table suggests, link up events take much longer (seconds) to incorporate. This is in fact expected and acceptable, because when a link is added, a potentially large number of pipes will be created which changes routing of flows significantly. Fortunately, since the link up/down event should be rare, this run time appears acceptable.

7 Discussion

Conventional Networks: Conceptually, NetPlumber can be used with conventional networks as long as we implement a notification mechanism for getting updated state information. One way to do this is through SNMP traps; every time a forwarding entry or link state changes, NetPlumber gets a notification. The drawback of such a mechanism is resource consumption at the switch.

Handling Transient Violations: Sometimes, during a sequence of state updates, transient policy violations may be acceptable (e.g. a black hole is acceptable while installing a path in a network). NetPlumber probes can be turned off during the transition and turned on when the update sequence is complete.

Handling Dynamic Policies: In multi-tenant data centers, the set of policies might change dynamically upon VM migration. NetPlumber can handle dynamic policy changes easily. In the plumbing graph, if we attach a source node to every edge port (as we did in the case of Google WAN), we can update policies by changing the locations and test conditions of probe nodes. This update is fast as long as the structure of the plumbing graph and routing of flows doesn't change.

[10] We used the same snapshots.

[11] By contrast, dependent policies can be checked using a single source node.

Limitations of NetPlumber: NetPlumber, like HSA relies on reading the state of network devices and therefore cannot model middleboxes with dynamic state. To handle such dynamic boxes, the notion of "flow" should be extended to include other kind of state beyond header and port. Another limitation of NetPlumber is its greater processing time for verifying link updates. As a result, it is not suitable for networks with a high rate of link up/down events such as energy-proportional networks.

8 Related Work

Recent work on network verification, especially on troubleshooting SDNs, focuses on the following directions.

Programming foundations: Frenetic [3] provides high-level abstractions to achieve per-packet and per-flow consistency during network updates [13]. NetPlumber, on the other hand, verifies forwarding policies.

Offline checking: `rcc` [2] verifies BGP configurations. NICE [1] applies model checking techniques to find bugs in OpenFlow control programs. HSA [8] checks data plane correctness against invariants. Anteater [10] uses boolean expressions and SAT solvers for network modeling and checking. However, offline checking cannot prevent bugs from damaging the network until the periodic check runs.

Online monitoring: Several tools help troubleshoot network programs at run-time. OFRewind [14] captures and reproduces the sequence of problematic OpenFlow command sequence. ATPG [16] systematically generates test packets against router configurations, and monitors network health by perioidically sending these tests packets. NDB [4] is a network debugger. These tools complement but not replace the need for real-time policy verification.

VeriFlow [9] is the work most closely related to NetPlumber. VeriFlow also verifies the compliance of network updates with specified policies in real time. It uses a *trie structure* to search rules based on equivalence classes (ECs), and upon an update, determines the affected ECs and updates the forwarding graph for that class. This in turn triggers a rechecking of affected policies. NetPlumber and VeriFlow offer similar run-time performance. While both systems support verification of forwarding actions, NetPlumber additionally can verify arbitrary header modifications, including rewriting and encapsulation. NetPlumber is also protocol-independent.

9 Conclusions

This paper introduces NetPlumber as a real-time policy checker for networks. Unlike earlier work that checks periodic snapshots of the network, NetPlumber is fast enough to validate every update in real time. Users can express a wide range of policies to be checked using an extensible regular-expression like language, called Flowexp. Since Flowexp might be too low-level for administrators to use, we implemented a higher level policy language (inspired by FML) implemented in Prolog.

The fundamental idea of the dependency graph formalized as a plumbing graph benefits us in three ways. First, it allows incremental computation by allowing only the (smaller) dependency subgraph to be traversed when a new rule is added. Second, it naturally leads us to generalize to probe nodes that can be configured to check for new policies—without the ad hoc programming effort required by Hassel. Third, clustering the graph to minimize inter-cluster edges provides a powerful way to parallelize computation.

NetPlumber is useful as a foundation that goes beyond static policy checking. For example, it can be used in ATPG [16] to allow the suite of ATPG tests packets to be updated swiftly when the configuration changes. Also NDB [4] may benefit from NetPlumber. Like GDB, NDB allows setting break points in the system when a specified condition is met. To achieve this goal, NDB adds a "postcard generating action" that captures and sends samples of matching packets to a central database. NetPlumber can be used to notify NDB when a rule that requires postcard action is about to be added to the network. While these are only two examples, we believe that the ability to incrementally and quickly do header space analysis will be a fundamental building block for network verification tools going forward.

10 Acknowledgements

We would like to thank our shepherd, Brad Karp, and the anonymous reviewers for their valuable comments. We thank Faro Rabe (Google) for providing the snapshots of Google WAN and Urs Hölzle and Stephen Stuart (Google) for internal review of this work. This research was sponsored by Google Summer Internship program, Stanford CS Undergraduate Research Internship (CURIS) program and NSF grant CNS-0855268.

References

[1] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proceedings of NSDI'12*, 2012.

[2] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of NSDI'05*, pages 43–56, 2005.

[3] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. *SIGPLAN Not.*, 46(9):279–291, Sept. 2011.

[4] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-

defined network? In *Proceedings of HotSDN '12*, pages 55–60, 2012.

[5] Header Space Library and NetPlumber. https://bitbucket.org/peymank/hassel-public/.

[6] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *Proceedings of WREN '09*, pages 1–10, 2009.

[7] The Internet2 Observatory Data Collections. http://www.internet2.edu/observatory/archive/data-collections.html.

[8] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proceedings of NSDI'12*, 2012.

[9] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proceedings of NSDI'13*, 2013.

[10] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of SIGCOMM '11*, pages 290–301, 2011.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR.*, 38:69–74, March 2008.

[12] Protobuf. http://code.google.com/p/protobuf/.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of SIGCOMM '12*, pages 323–334, 2012.

[14] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: enabling record and replay troubleshooting for networks. In *Proceedings of USENIX-ATC'11*, 2011.

[15] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Technical Report Stanford/TR12-HPNG-061012, Stanford University, June 2012.

[16] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *Proceedings of CoNEXT 2012*, Nice, France, December 2012.