Verified Compilation of Abstract Network Policies

by

Theodore Katz

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 17th, 2019

Certified by . . . .
Adam Chlipala
Associate Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Verified Compilation of Abstract Network Policies

by

Theodore Katz

Submitted to the Department of Electrical Engineering and Computer Science
on May 17th, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Configuring large networks can be very complex. A network administrator typically has a set of high-level policies in mind when creating a network configuration, but implementing the configuration onto existing hardware often requires specifying many low-level details. As a result, configuring a network is currently a very error-prone process, and misconfigurations resulting in network outages and security vulnerabilities occur frequently in practice. We present a formally verified compiler from high-level network policies to low-level executable routing rules, to simplify the process of correctly configuring networks and enforcing network policies.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Computer Science

Acknowledgements

I would like to thank my advisors, Adam Chlipala and Michael Greenberg, for their guidance and support on this project, without which the result would not have been possible. I would also like to thank my parents for their support throughout my education.

Contents

1 Introduction **9**

2 Compiler Architecture **13**

2.1 Overview of Coq, Gallina, and Ltac 13

2.2 Overview of OpenFlow 14

2.3 Components of the Compiler 14

3 Modeling the Network **17**

3.1 Topologies and Networking Basics 17

3.2 Network Policies 20

3.3 Assumptions about Network Structure 23

4 IR layers of the Compiler Backend **25**

4.1 The `next_node` Relation 25

4.2 Generating Valid `next_node` Relations 28

4.2.1 Avoiding Routing Cycles 29

4.2.2 Ensuring Enough Paths are Generated 31

4.3 Deterministic `next_node` Functions 33

4.4 Routing Tables 33

4.5 OpenFlow Actions 35

4.6 Correctness of Generated OpenFlow Entries 37

4.6.1 Modeling the Network 37

4.6.2 Correctness for Allowed Packets 41

4.6.3 Correctness for Disallowed Packets 41
4.7 Modeling Stateful Policies 42
4.7.1 Transition Systems for Dynamic Policies and Controllers . . . 42
4.7.2 Policy Validity 43
4.8 Limitations of the One-Packet-at-a-Time Network Model 47
4.8.1 Comparison to Real Networks 47
4.8.2 Efficiency 48

5 Components of the Compiler Frontend **51**
5.1 Overall Architecture and User Interface 51
5.2 Comparing the Compiler Frontend Architecture with Alternative Designs 52
5.3 Proof Obligations 53
5.3.1 Topology Validity 53
5.3.2 Policy and All-Pairs-Paths Validity 54
5.3.3 Host-IP Address Mapping 55
5.3.4 Enumeration of Nodes 55

6 I/O Translation Layer **57**
6.1 Evaluation of the I/O Layer 57
6.2 Impact of Unverified Components on Correctness 58

7 Evaluating the Compiler in Practice **61**

8 Conclusions and Future Work **65**

Bibliography **67**

Chapter 1

Introduction

Network administrators frequently need to create and enforce routing policies across a network. For instance, when configuring an Internet service with load balancing, it might be necessary to ensure that the load balancer can communicate over the network with each origin server, while also preventing origin servers from receiving any other packets from outside the network.

As the complexity of a network increases, implementing a high-level policy can be extremely error-prone, particularly if the implementation is carried out manually. Incorrectly blocking a network connection can result in a system outage, and incorrectly allowing a connection can expose private information or render a system vulnerable to attacks such as server-side request forgery. Indeed, bugs in network configurations routinely cause problems in practice (e.g. [15, 16]).

To avoid these problems, it is helpful to use a tool that automates some of the reasoning required for manual network configuration, allowing a network administrator to operate a network at a high level of abstraction without being concerned about low-level details. An ideal tool would be robust enough to provide strong correctness guarantees, while also being expressive enough to fulfill the complex requirements of real-world networks. While this ideal is difficult to achieve, several recent projects have proposed a variety of tools and abstractions that substantially reduce the effort required to manage a network.

VeriFlow [7] is a runtime network analysis tool that detects the violation of

network-wide invariants as they occur, ensuring that networking problems can be detected quickly. For example, one might use VeriFlow to ensure that a particular destination in a network remains reachable, and to raise an alert if this invariant is ever violated. VeriFlow operates on a "live" network at runtime, allowing it to test invariants without requiring access to the precise configuration details of network components.

While the ability to detect invariant violations at runtime is undoubtedly useful, it is often preferable to prevent invariant violations from occurring in the first place. VeriCon [3], VMN [11], and Kuai [9] apply model checking to network behavior, with the goal of proving that specified invariants cannot be violated under a formal model of possible network events. NetKAT [1] and "Decentralizing SDN Policies" [10] provide a formal basis for implementing network policies by optimally sending forwarding rules to distributed physical switches.

As an alternative to formally verifying very complex network configurations, some projects have focused on simplifying the programming model for large networks to reduce the risk of implementing bugs. Frenetic [5], SNAP [2], FatTire [12], and Merlin [14] introduce abstract languages describing the behavior of entire networks and compile the languages to executable rules running on distributed hardware. These languages vary in expressive power and purpose. Frenetic and SNAP's programming models aim to allow the creation of imperative networking programs, where Frenetic locates all program state at a controller, and SNAP provides a mechanism to distribute program state throughout the network. FatTire allows the user to specify paths through a network and automatically generates fault-tolerant implementations. Similarly, Merlin allows the user to declare constraints on network resources (e.g. a maximum bandwidth or reachability requirement between two switches) and automatically compiles these constraints to compliant implementations.

These projects significantly decrease the difficulty of configuring networks for many common use cases. However, some gaps in the existing work merit another look at the problem. Declaring the invariants of an entire network in a single programming model substantially reduces mental overhead in implementation, but existing

tools for declarative compilers (such as NetKAT, FatTire, and Merlin) provide fairly limited expressive power for network policies. Additionally, FatTire and Merlin are not formally verified, creating a risk of bugs in the respective compilers.

To address these gaps, we present a verified compiler for network policies. The compiler accepts as input a high-level network policy in a functional language, and it outputs executable routing rules along with a proof that the rules implement the policy.

Chapter 2

Compiler Architecture

Our main contribution is a proof-generating compiler, mostly written in Coq, from abstract network policies to executable routing rules.

To aid in understanding the compiler, we first provide a brief overview of Coq and OpenFlow.

2.1 Overview of Coq, Gallina, and Ltac

Coq is a "proof assistant", i.e. a programming language that acts as a tool for creating mathematical proofs. Coq itself is composed of two smaller languages, called Gallina and Ltac.[1]

Gallina is a dependently typed, strongly normalizing, pure functional language based on the Calculus of Inductive Constructions [6]. Using the Curry-Howard Correspondence, a user can create a proof of a mathematical statement in Gallina by constructing an inhabitant of a specified Gallina type.

To aid in the construction of proofs, a Coq user typically uses *Ltac* [4], a dynamically typed scripting language designed for manipulating Gallina terms. Notably, although Ltac provides an expressive mechanism for manipulating Gallina terms, it does not allow the "rules" of Gallina to be broken; if an Ltac script succeeds, it pro-

[1]We only provide a brief overview of Coq here. More information and resources about the language can be found at the project homepage `https://coq.inria.fr/`.

duces a valid Gallina term. As a result, Ltac allows the creation of *proof-generating* compilers which are not *proven correct*. In other words, a proof-generating compiler that uses an Ltac script may sometimes fail to output anything, but if it does produce an output, the output will be accompanied by a proof of correctness.

Throughout this text, we will include code snippets containing Coq code when discussing some of our formal models and correctness conditions. Understanding these code snippets is not a requirement for understanding the surrounding prose, but our intent is that they may aid in comprehending the finer details of the compiler. The full source code of the compiler, including associated proofs and tests, can be found at `https://github.com/mit-plv/network-configurations`.

2.2 Overview of OpenFlow

OpenFlow is an open protocol for communication between a network controller and a set of network switches. The OpenFlow specification [18] defines a binary format through which controllers can communicate with and install rules on switches. The simplest OpenFlow rules might specify that all packets that arrive at a switch should be forwarded to a certain location, or all packets that match a specific IP address mask should be forwarded to another location. The protocol also supports primitives such as timeouts and counters, which allow a controller to (e.g.) specify that a routing rule should only be in effect for a specific amount of time.

2.3 Components of the Compiler

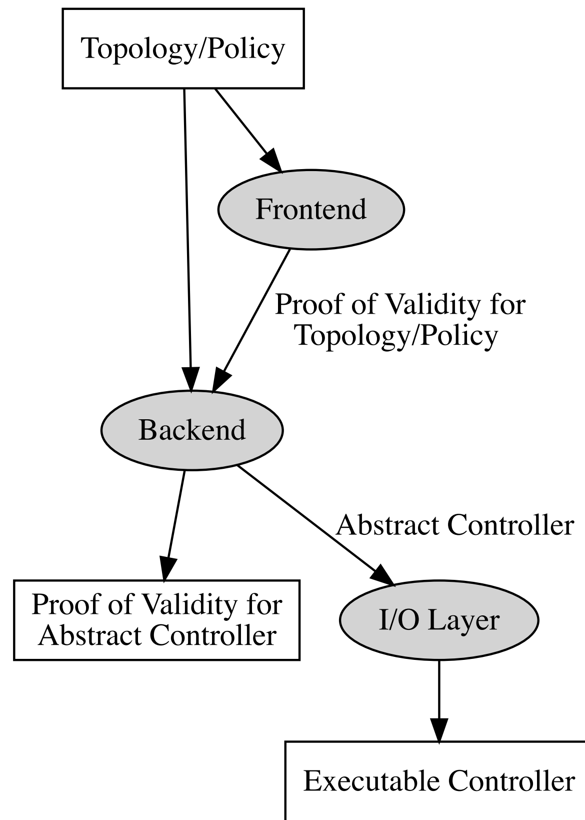
The internal structure of the compiler consists of three main components:

1. The largest part of the compiler is the backend: a total dependently typed function, expressed in Gallina, that accepts as arguments a topology, a policy, and a set of proof obligations ensuring that the inputs to the compiler are valid.

2. In order to help satisfy these proof obligations, the compiler contains a small

frontend consisting of Ltac scripts. The user of the compiler is expected to provide certain inputs as Gallina terms, and these Ltac scripts typically automate most of the process of proving the correctness of the inputs.

3. The output of the compiler backend is an abstract representation of an OpenFlow controller, as well as a proof that the controller produces correct network behavior according to the abstract network model. The abstract controller representation is then compiled with the final piece of the compiler, a small unverified conversion layer written in OCaml which performs I/O and serialization. The result is an executable binary that can run on a real network.



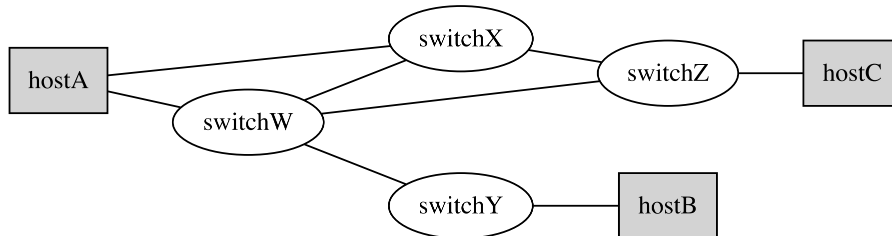
High-Level Structure of the Compiler

Chapter 3

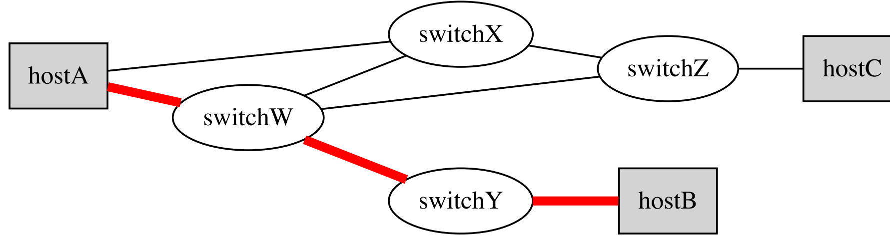
Modeling the Network

3.1 Topologies and Networking Basics

A typical network contains a set of *hosts*, or devices that may need to communicate with each other. To facilitate communication between hosts, networks also contain *switches* which connect to hosts and to each other.



For example, in the figure above, suppose `hostA` is a server that needs to send a message to `hostB`, a database. `hostA` would accomplish this by sending one or more *packets* containing information to an adjacent switch, such as `switchW`. `switchW` can then forward the packet to another adjacent switch, such as `switchY`. Since `switchY` is adjacent to the intended destination of `hostB`, `switchY` could then forward the packet directly to `hostB`. At this point, `hostB` has received the message and can take action as appropriate.



The collection of hosts, switches, and links between them defines a *network topology*. More formally, we model a network topology as a finite directed graph, where each node in the graph is tagged as either a host or a switch. For simplicity, we require that every host in the network has an outgoing edge to at least one switch.

Each outgoing edge from a node is labelled by a locally unique sixteen-bit integer representing a *port*. Port numbers are unique on a per-node basis. (For example, any particular node can have at most one outgoing edge on port 5, but the network as a whole can contain multiple edges on port 5 in different locations.)

```
Context {Switch : Set}.
Context {Host : Set}.

Inductive Node :=
| SwitchNode : Switch → Node
| HostNode : Host → Node
.

Definition Port : Set := word 16.

Definition network_topology := Node → Node → option Port.

Record valid_topology (topology : network_topology) := {
  no_duplicate_ports : ∀ node outgoing₁ outgoing₂ ,
    match topology node outgoing₁ , topology node outgoing₂ with
    | Some port₁ , Some port₂ ⇒ port₁ = port₂ → outgoing₁ = outgoing₂
    | _, _ ⇒ ⊤
    end;
  valid_port_numbers : ∀ switch₁ node₂ ,
    match topology (SwitchNode switch₁ ) node₂ with
```

```
    | Some port ⇒ port ≠ natToWord 16 0
    | None ⇒ ⊤
    end;
  no_isolated_hosts : ∀ host,
    ∃ switch,
      topology (HostNode host) (SwitchNode switch) ≠ None
}.
```

Using this formalization, we can represent the example topology used above (with port numbers chosen arbitrarily):

```
Inductive ExampleHost :=
| hostA
| hostB
| hostC
.

Inductive ExampleSwitch :=
| switchW
| switchX
| switchY
| switchZ
.

Definition example_topology node₁ node₂ :=
  match node₁ , node₂ with

  (* hostA <--> switchW edge *)
  | HostNode hostA, SwitchNode switchW ⇒ Some (portNo 1)
  | SwitchNode switchW, HostNode hostA ⇒ Some (portNo 2)

  (* switchW <--> switchY edge *)
  | SwitchNode switchW, SwitchNode switchY ⇒ Some (portNo 3)
  | SwitchNode switchY, SwitchNode switchW ⇒ Some (portNo 4)

  (* ... other edges omitted for brevity ... *)
  | _, _ ⇒ None
  end.
```

From this simple example topology, we make two basic observations:

1. The decisions made by switches in a network are very important to ensure the

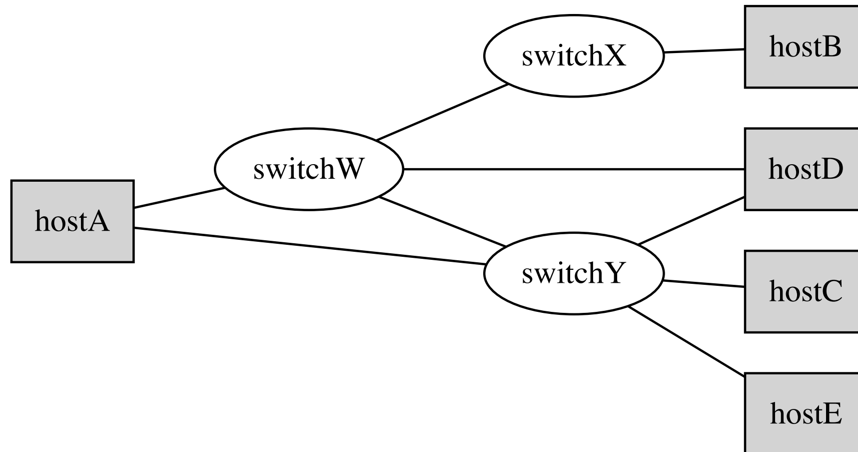
correct function of the network as a whole. For example, if `switchW` erroneously forwarded `hostA`'s packet to `switchZ` instead of `switchY`, then the packet would end up in a "black hole" where it could no longer reach its intended destination of `hostB` from `switchZ`. In effect, `hostA` would be rendered entirely unable to send messages to `hostB`, because none of its packets would arrive.

2. If desirable, a certain set of switches in a network can collaborate to ensure that a specific pair of hosts is never able to communicate, by simply dropping (refusing to forward) any packets sent between the two hosts.

We are far from the first to make these observations; using network switches to block specific connections is the main purpose of network firewalls, which have seen widespread usage for decades. However, these observations motivate the idea of a network policy, i.e. a mechanism for regulating the connections on a network by controlling the switches.

3.2 Network Policies

In a typical network, it is important that messages between certain pairs of hosts are allowed, and that messages between other pairs of hosts are disallowed. For example, consider the following network:



Suppose `hostB` is a public-facing server, `hostA` is an untrusted client for that server, and `hostC` is a database connected to that server. It is likely desirable that `hostA` can send messages to `hostB`, and `hostB` can send messages to `hostC`. However, for security reasons, it might also be desirable that `hostA` cannot send messages directly to `hostC`. To ensure that these requirements are followed, it might be necessary to install nontrivial routing logic on the switches between `hostA`, `hostB`, and `hostC`.

This set of requirements is an example of a *static network policy*, i.e. a static declaration of which pairs of hosts are allowed to connect to each other, and which pairs are not.

(hostA, hostB): allowed
(hostB, hostC): allowed
(hostA, hostC): disallowed
...

An example of a static policy

More formally, we define a static policy as a computable predicate over network flows, where a *network flow* is an ordered pair of hosts in the topology that might

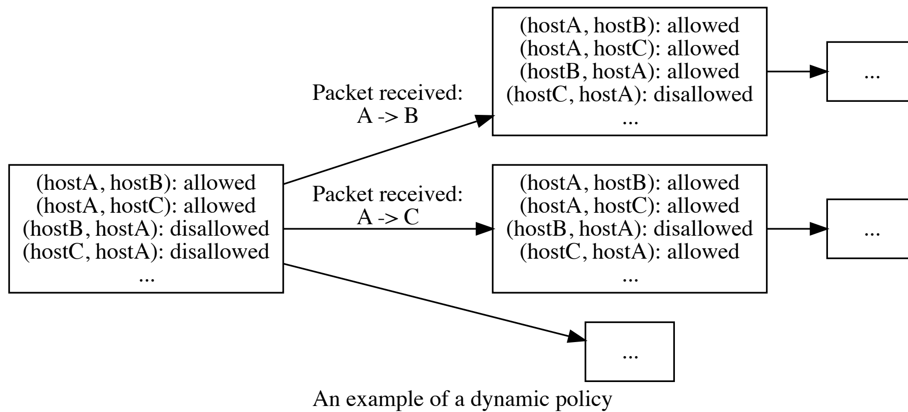
attempt to connect to each other.

```
Record flow := {
  Src : Host;
  Dest : Host
}.

Definition static_network_policy := flow → 𝔹.
```

While static network policies are relatively simple to reason about, many real-world networks have requirements which cannot be expressed through static policies. In the previous diagram, suppose `hostD` is an external server, and `hostE` is a potential client for that server on an internal network. Initially, it might be undesirable for `hostD` to be able to send messages to `hostE`, since this would require untrusted packets from `hostD` to be able to enter the internal network. However, it might be acceptable for `hostE` to initiate a connection with `hostD`, at which point `hostD` would need to send a message back to `hostE` in order to provide a response. In other words, it might be permissible for `hostD` to send a message to `hostE` only if `hostE` has sent a message to `hostD` at some point in the recent past.

More generally, this example suggests that the set of permissible network flows should be able to change over time based on the packets that have previously been sent over the network. To account for this use case, we define a *dynamic policy* as a labelled transition system where each state produces a static policy, and state transitions are triggered by successful packet transmissions in the network.



An example of a dynamic policy

Note that as the network topologies and dynamic policies increase in complexity, it becomes substantially more difficult to create an ad-hoc description of how each switch should operate while still satisfying all of the necessary constraints. For example, in the running example it is not immediately clear what the behavior of `switchY` should be over time. (Should it blindly forward all packets and leave another switch with the responsibility of policy enforcement? Should it take responsibility for policy enforcement between certain pairs of nodes, but not others?) Under this view, it is not surprising that bugs occur so frequently when manually configuring highly complex networks.

3.3 Assumptions about Network Structure

In our model of a network, we make several simplifying assumptions.

1. All of the switches in a network topology are under the control of a single network administrator.

   Our compiler produces a set of routing rules for every switch in the topology, and the correctness of the system depends on all of the switches behaving according to those routing rules. (In some real-life networks, some switches might be operated by a third party, making it difficult to install custom logic on those switches. These scenarios are considered out-of-scope for our model.)

2. Network topologies are assumed to be completely static; no switches or hosts can be added or removed at runtime.

   In large real-life networks, this assumption typically does not hold; links to switches can occasionally go down, and a network is typically expected to be resilient to a single switch failing. However, there is substantial existing work (e.g. VMN [11], FatTire [12]) on the issue of reliable fault tolerance for link failures. To simplify our model, we assume no link failures can occur. A user of our compiler would likely choose to combine it with something like FatTire [12] before deploying it in a network where link failures are a concern.

Chapter 4

IR layers of the Compiler Backend

We now begin our discussion of the architecture of the compiler itself, starting with the compiler backend. The compiler contains several intermediate representations of network routing rules, starting with highly abstract specifications of routing behavior and gradually moving down to concrete implementations.

4.1 The `next_node` Relation

At a high level of abstraction, the task of our compiler is to decide, given any switch and packet, the set of locations where the switch should be permitted to forward the packet. We encode this decision in a general representation of a set of routing rules, called a `next_node` relation. For any given switch, and any given network flow for a packet arriving at that switch, a `next_node` relation defines a predicate on nodes that determines whether the given node is to be considered a valid hop from the switch.

```
Definition next_node := Switch → flow → Node → ℙ.
```

Intuitively, a `next_node` relation can be considered to represent the routing behavior of an entire network at a fixed point in time.

It should be noted that the routing decisions of a `next_node` relation are inhabitants of Coq's `Prop` type (represented by $\mathbb{P}$ in the snippet above), which can contain

arbitrary logical propositions. As a result, routing decisions are not directly computable from `next_node` relations in the general case. (For example, a `next_node` relation could specify that a particular packet should be dropped if and only if P equals NP; such a relation would be valid but very difficult to work with.) Later, we will consider a variant of `next_node` which requires relations to be decidable.

The `next_node` relation allows us to define a correctness condition for routing behavior given a particular topology and static policy, before considering more complex questions about network policy implementation (e.g. decentralization of routing decisions). We define a `next_node` relation to be correct when all of the following conditions hold:

1. The `next_node` relation respects the topology, i.e. it only specifies that a switch should forward a packet to a node if there exists a link between the switch and the node in the topology.

2. For all network flows allowed by the current static policy, and all switches connected to the source of the flow, there exists a path from the first switch to the destination host such that each switch in the path can forward packets for that flow to the next node in the path, when making routing decisions based on the `next_node` relation.

   Note that this claim is stronger than requiring such a path to exist between the source and destination host, because it requires the path to exist regardless of where the host decides to send the packet on the first hop. The compiler produces executable rules that run on network switches, but has no control over the behavior of network hosts. Therefore, when evaluating the correctness of the compiler, we require the system to behave appropriately regardless of the decisions made by hosts.

3. For all network flows disallowed by the current static policy, there is no path from the source of the flow to the destination of the flow when making decisions based on the `next_node` relation.

4. The `next_node` relation prohibits "black holes". In other words, if a switch can forward an allowed packet to a particular location, the packet will eventually reach its destination from that location after a finite number of hops.

5. The `next_node` relation prohibits cycles. In other words, for any switch `s` there exists no nonempty path from `s` to `s` for which all hops in the path are allowed by the `next_node` relation.

6. The `next_node` relation never forwards a packet from a switch to a host unless that host is the intended destination of the packet.

```
Record next_node_valid
    (topology : network_topology)
    (policy : static_network_policy)
    (next : next_node)
:= {
  all_hops_in_topology : ∀ here current_flow hop_target,
    next here current_flow hop_target →
      topology (SwitchNode here) hop_target ≠ None;

  path_exists_only_for_valid_flows : ∀ current_flow first_switch port,
    topology (HostNode current_flow.(Src)) (SwitchNode first_switch)
      = Some port
      → (
        (policy current_flow = true) ↔
        ∃ path, is_next_node_path next path first_switch current_flow
      );

  no_black_holes : ∀ here current_flow hop_target,
    policy current_flow = true
      → next here current_flow hop_target
      →
        match hop_target with
        | HostNode dest ⇒
          dest = current_flow.(Dest)
        | SwitchNode next_switch ⇒
          ∃ path,
            is_next_node_path next path next_switch current_flow
        end;
```

```
  all_paths_acyclic : ∀ path here current_flow,
    is_next_node_path next path here current_flow
      → NoDup (here :: path);

  forwards_to_correct_host : ∀ here current_flow end_host,
    next here current_flow (HostNode end_host)
      → end_host = current_flow.(Dest)
}.
```

4.2 Generating Valid next_node Relations

Having defined the conditions under which a given next_node relation is considered valid, we now discuss how a valid relation could be constructed. We consider a function that attempts to find a path in the topology between a given switch and a given destination host.

```
Definition all_pairs_paths := Switch → Host → option (list Switch).
```

Note: Our Gallina encoding of a path here only includes intermediate hops and omits the endpoints. This avoids the need to store a heterogeneous list, since the intermediate hops are always switches and the destination endpoint is always a host. However, when discussing paths in prose here, we will include both the hops and the endpoints for clarity.

Given such an all-pairs paths generator, we can generate a next_node relation as follows: at each switch s, check whether the current flow is allowed by the policy. If it is, allow the switch to forward only to the first node on the path from s to the destination host.

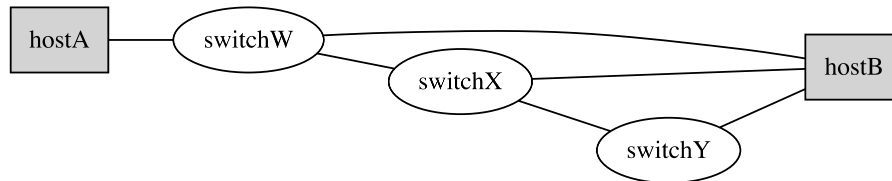
```
Definition all_pairs_paths_next_node_generator
  (paths : all_pairs_paths)
  (topology : network_topology)
  (policy : static_network_policy)
  here
  current_flow
  hop_target
```

```
:=
  policy current_flow = true ∧
  match paths here current_flow.(Dest) with
  | Some (hop_target' :: _) ⇒ hop_target = SwitchNode hop_target'
  | Some [] ⇒ hop_target = HostNode current_flow.(Dest)
  | None ⇒ ⊥
  end.
```

Intuitively, this implementation of a `next_node` relation simply works by trying to compute a path in the topology from every switch to every host, and making routing decisions locally at each switch based on the computed path from that switch to the destination.

4.2.1 Avoiding Routing Cycles

Perhaps surprisingly, this routing strategy is not always valid. Consider the following example:



Suppose a given `all_pairs_paths` implementation decides that the path from `switchW` to `hostB` should proceed through `switchX` (i.e. the path `[switchW; switchX; hostB]`), and the path from `switchX` to `hostB` should proceed through `switchW` (i.e. the path `[switchX; switchW; switchB]`). Both of these paths are valid on their own, but generating a `next_node` relation from these paths will cause a packet from `hostA` to `hostB` to bounce between `switchW` and `switchX` indefinitely. (At `switchW`, the next step on the path to `hostB` is `switchX`, and at `switchX` the next step on the path to `hostB` is `switchW`.)

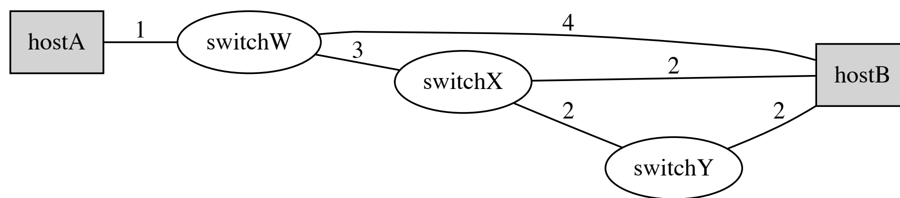
This anomaly arises because an `all_pairs_paths` implementation is not necessarily *consistent*. At each switch, we consider the entire path to the destination host,

and forward a packet to the next switch along that path. But once the packet reaches the next switch, it might be forwarded according to an entirely different path to the destination.

While sending a packet in a loop forever is certainly undesirable, we consider the absence of a consistency requirement in `all_pairs_paths` to be a useful allowance. In real networks, switches do not typically need to be aware of every hop that a packet will take to its destination; instead, they can just forward the packet based on some heuristic and let the other switches in the network decide on the subsequent hops.

Therefore, we solve this problem without introducing a global path consistency requirement, by instead requiring that the `all_pairs_paths` implementation generates *decreasing costs*. An implementation is said to generate decreasing costs when there exists a global cost function mapping each edge of the topology to a positive integer, such that the path cost to the destination from the first hop of a path is always less than the path cost from the start of the path.

For example, consider the following topology with a cost assignment to each edge:



As before, suppose the given `all_pairs_paths` implementation specifies that the appropriate path from `switchW` to `hostB` is `[switchW; switchX; hostB]`. The total cost of this path is 5 (it has one hop with cost 3, and one hop with cost 2).

Since the first hop on the path leads to `switchX`, the decreasing-costs requirement ensures that the path from `switchX` to `hostB` must have a total cost less than 5. This rules out the anomalous path `[switchX; switchW; hostB]` discussed before, since that path has cost 7 with our current choice of cost assignments. However, the `all_pairs_paths` implementation could still generate the path `[switchX; switchY; hostB]` or simply `[switchX; hostB]`, since they have costs of 4 and 2 respectively.

Note that at this point, we are unconcerned about whether an `all_pairs_paths` implementation selects an *optimal* set of paths in order to minimize costs; in fact, an `all_pairs_paths` implementation can be created independently of a cost function. We only require that there is some cost assignment for which path costs monotonically decrease on all switch-to-host paths.

Given such a cost assignment, it is relatively simple to show that the resulting routing rules prohibit cycles, by induction on the path cost for each switch/host pair.

```
Definition generates_decreasing_costs
  topology
  (paths : all_pairs_paths)
  costs
:=
  only_positive_costs topology costs ∧
  ∀ src dest,
    match paths src dest with
    | Some (hop_target :: cdr) ⇒
      match paths hop_target dest with
      | Some path ⇒
        match
          path_cost topology costs hop_target dest path,
          path_cost topology costs src dest (hop_target :: cdr)
        with
        | Some remaining_cost, Some original_cost ⇒
          remaining_cost < original_cost
        | _, _ ⇒ ⊥
        end
      | None ⇒ ⊥
      end
    | _ ⇒ ⊤
    end.
```

4.2.2 Ensuring Enough Paths are Generated

Not all topologies are strongly connected; in some cases, there is no possible path from a particular switch to a particular host. We account for this possibility by defining `all_pairs_paths` implementations to return `option (list Switch)` rather than `list Switch`; an implementation can explicitly fail to generate a path for a

particular switch-host pair by producing `None` for that pair. If there is no path between a particular switch and a destination host, the switch will simply never forward any packets for flows with that destination.

Since our overall goal is to ensure that packets allowed by the current policy will always reach their destinations, we therefore need to ensure that this scenario will never occur for packets allowed by the current static policy. To accomplish this, we impose the following restrictions on `all_pairs_paths` implementations:

1. If the flow `(src, dest)` is allowed by the current static policy, then for every switch `sw` adjacent to `src`, the `all_pairs_paths` implementation generates a path for the switch-host pair `(sw, dest)`.

2. For any switch `sw`, if the path from `sw` to `dest` is `sw` :: `hop` :: `cdr`, then the `all_pairs_paths` implementation generates a path for the switch-host pair `(hop, dest)`.

From these constraints, the proof of path existence follows by induction on the initial path from `sw` to `dest`.

```
Record all_pairs_paths_valid topology policy paths := {
  paths_in_topology : ∀ src dest,
    match paths src dest with
    | Some path ⇒ is_path_in_topology topology src dest path
    | _ ⇒ ⊤
    end;

  paths_exist_for_valid_flows : ∀ current_flow first_switch port,
    policy current_flow = true
      →
        topology
          (HostNode current_flow.(Src))
          (SwitchNode first_switch)
        = Some port
      →
        match paths first_switch current_flow.(Dest) with
        | Some _ ⇒ ⊤
        | None ⇒ ⊥
```

```
      end;

  paths_move_closer_to_destination : ∃ (costs : edge_costs),
    generates_decreasing_costs topology paths costs
}.
```

4.3 Deterministic next_node Functions

Using the `all_pairs_paths` abstraction, we have shown that we can create a `next_node` relation that satisfies our validity requirements. The `next_node` relation specifies that packet transitions are nondeterministic and are not computable in the general case, which makes `next_node` somewhat unwieldy to work with. We specialize the general `next_node` relation to `dec_next_node`, a computable, deterministic function that decides where a switch forwards any particular network flow.

```
Definition dec_next_node := Switch → flow → option Node.

Definition dec_next_node_valid topology policy dec_next :=
  next_node_valid topology policy (λ here current_flow hop_target ⇒
    dec_next here current_flow = Some hop_target
  ).
```

We can generate valid `dec_next_node` functions in the same manner as valid `next_node` relations using an `all_pairs_paths` implementation.

4.4 Routing Tables

Given a valid `dec_next_node` function, we define a set of *routing tables* as a mapping from each switch to the enumeration of all allowed flows that could end up at that switch, each combined with the following node that packets for that flow should be forwarded to from that switch.

```
Definition routing_tables := Switch → list (flow * Node).
```

We define a set of routing tables to correctly implement a `dec_next_node` function if the following two constraints hold:

1. The tables match the given `dec_next_node` function; a switch has a particular flow/node pair in its tables iff the `dec_next_node` function dictates that the switch will forward the given flow to the given node.

2. The tables are well-formed (with each flow corresponding to at most one forwarding destination).

```
Record routing_tables_valid
  (tables : routing_tables)
  (dec_next : dec_next_node)
:= {
  no_duplicate_entries : ∀ here,
    NoDup (tables here);

  entries_match_next_node_result : ∀ here current_flow hop_target,
    dec_next here current_flow = Some hop_target
      ↔ In (current_flow, hop_target) (tables here)
}.
```

Note that the `routing_tables_valid` condition only ensures that a set of routing tables correctly implements a `dec_next_node` function, but does not ensure that the corresponding routing decisions are semantically reasonable according to any policy. (We continue to use the `dec_next_node_valid` condition to show semantic correctness at this point.)

To generate routing tables from `dec_next_node` functions, we simply enumerate all possible flows in the network at each switch, and add an entry to the corresponding table in each case where the `dec_next_node` function would produce an output for that flow. Of course, enumerating all pairs of flows in the network requires for the first time that the network has a finite number of hosts. (Hosts and switches are defined as inhabitants of the `Set` type in Gallina, which in general can be infinite. However, we have no practical use for abstract topologies whose hosts are defined

as, say, the set of natural numbers.) To generate all flows, we therefore require the compiler frontend to provide an explicit enumeration of all members of the `Host` set.

4.5 OpenFlow Actions

At this point, our policy is suitable for direct conversion to OpenFlow. In order to accomplish this, we need to formally model some of the OpenFlow specification. Since we assume that the compiler has full control over all switches in the network, we also have full control over all of the OpenFlow data being sent through the network. As a result, we only need to formally model the subset of OpenFlow that the compiler actually produces; the remainder of the OpenFlow spec will not appear on the network and is therefore not necessary to model here.

At a high level of abstraction, a network using OpenFlow typically works as follows:

- Each switch in the network is connected to a *controller*. Connections between a switch and a controller can exist independently of the network topology between switches. (The graph of switch-controller connections comprises the *control plane*, whereas the graph of switch-switch and switch-host connections comprises the *data plane*.)
- At any given point in time, a switch has a set of *flow entries* that determine how the switch should respond to a particular type of incoming packet.
- Each flow entry contains of a set of *matchers* that determine whether the rule should apply to any given packet based on data in the packet header. A filter typically takes the form of "match all IP packets", or "match all IP packets with a specific host IP address".
- Each flow entry also contains a set of *actions* that determine what the switch should do with a packet if it determines that the current rule applies to that packet. An action typically takes the form of "forward this packet out a particular port", or "forward this packet to the controller".

- At any point in time, a controller can send a message to a connected switch instructing it to modify the set of flow entries in use.

Our formal model of OpenFlow flow entries roughly matches the given description, with a few modifications. First, we only consider matchers that use exact matches on source and destination IP addresses. For the purposes of this model, we ignore all non-IP packets.

```
Definition ipv4_address : Type := word 8 * word 8 * word 8 * word 8.

Record ipv4_packet := {
  IpSrc : ipv4_address;
  IpDest : ipv4_address
}.

Record header_fields_matcher := {
  IpSrcMatcher : option ipv4_address;
  IpDestMatcher : option ipv4_address
}.
```

Second, instead of allowing a list of zero or more actions in a flow entry, our formal model requires exactly one action of three possible "pseudo-actions" corresponding to a subset of possible OpenFlow behaviors.

```
Inductive openflow_pseudo_action :=
| ForwardToSwitch : Port → openflow_pseudo_action
| ForwardToDest : Port → openflow_pseudo_action
| Drop
.

Record openflow_flow_entry := {
  header_fields : header_fields_matcher;
  action : openflow_pseudo_action
}.
```

These three constructors for `openflow_pseudo_action` correspond to three possible lists of concrete actions produced by the compiler. The `Drop` pseudo-action

produces an empty list of actions (i.e. indicating that a given packet should not be forwarded anywhere). The `ForwardToSwitch` pseudo-action produces a list with one action, which forwards the packet to the given port. The `ForwardToDest` pseudo-action produces a list with two actions, where one of the actions forwards the packet to the given port, and the other action forwards the packet to the controller (to trigger a state transition in the current dynamic policy). The conversion from a pseudo-action to an action occurs in the unverified OCaml layer of the compiler; for the purposes of our formal model, we treat the three pseudo-actions as routing primitives that provide the given behavior by construction.

With this model, we can now compile our routing tables directly to OpenFlow, by simply converting each entry in a routing table to a flow entry.

4.6 Correctness of Generated OpenFlow Entries

4.6.1 Modeling the Network

The correctness of the compiler is evaluated on an abstract model of a packet traveling through a network.

1. A host `src` can arbitrarily attempt to send a packet to another host `dest`. Initially, the packet resides at `src` in the `NotYetSent` state.

2. From the `NotYetSent` state, `src` then emits the packet by sending it to any adjacent switch `sw`. The packet then resides at `sw` in the `EnRoute` state.

3. From the `EnRoute` state, a switch `sw` can either:

    - Forward the packet to another switch `sw'`, provided that an edge exists from `sw` to `sw'` in the topology. The packet then resides at `sw'` in the `EnRoute` state.

    - Forward the packet to a host `endpoint`, provided that an edge exists from `sw` to `endpoint` in the topology. The packet then resides at `endpoint` in the `ReceivedAtHost` state.

- Drop the packet, causing it to move to the terminal `Dropped` state.

4. Finally, from the `ReceivedAtHost` state, a host `endpoint` can move the packet into the `Arrived` state, provided that `endpoint` is the intended destination of the packet (`endpoint = dest`).

```
Inductive openflow_network_packet_state :=
| NotSentYet
| EnRoute : Switch → openflow_network_packet_state
| ReceivedAtHost : Host → openflow_network_packet_state
| Arrived
| Dropped
.

Inductive openflow_network_step :
  openflow_network_packet_state → openflow_network_packet_state → ℙ :=
| EmitPacketFromSrc : ∀ port src_host first_switch,
  src_host.(host_ip) = packet.(IpSrc)
    → topology (HostNode src_host) (SwitchNode first_switch)
      = Some port
    → openflow_network_step NotSentYet (EnRoute first_switch)
| ForwardPacketToSwitch : ∀ port current_switch new_switch,
  get_matching_action packet current_switch.(entries)
    = ForwardToSwitch port
    → topology (SwitchNode current_switch) (SwitchNode new_switch)
      = Some port
    →
      openflow_network_step
        (EnRoute current_switch)
        (EnRoute new_switch)
| ForwardPacketToDest : ∀ port current_switch dest_host,
  get_matching_action packet current_switch.(entries)
    = ForwardToDest port
    → topology (SwitchNode current_switch) (HostNode dest_host)
      = Some port
    →
      openflow_network_step
        (EnRoute current_switch)
        (ReceivedAtHost dest_host)
| DropPacketAtSwitch : ∀ current_switch,
  get_matching_action packet current_switch.(entries) = Drop
```

```
    → openflow_network_step (EnRoute current_switch) Dropped
| AcceptPacket : ∀ dest_host,
  dest_host.(host_ip) = packet.(IpDest)
    → openflow_network_step (ReceivedAtHost dest_host) Arrived
.
```

Each switch uses only the source and destination IP address of a packet in order to determine where the packet should be sent, and whether it should be allowed. We assume that each host has a single, unique IP address which is already known to all other hosts that might attempt to contact it. (In effect, this would typically require the existence of a working DNS infrastructure, which is largely orthogonal to our requirements.)

For example, when `hostA` sends `hostB` a packet, our model dictates that it starts out by creating a packet in the `NotYetSent` state which has a source IP address of `hostA`'s IP, and a destination IP address of `hostB`'s IP. As the packet travels through the network, the switches would need to make a determination about how to process the packet based on its IP. Since network policies express constraints on hosts rather than constraints on IP addresses, we require the user to provide an injective mapping from hosts to IP addresses.

Given this model, a set of routing rules on switches is defined to be correct if the following conditions hold for all packets:

1. The packet system never gets "stuck" or ends up in a cycle, i.e. it always reaches the `Arrived` or `Dropped` state after a bounded number of steps.

2. If the flow represented by a packet is allowed by the static policy, the packet always reaches the `Arrived` state. If the flow is disallowed, the packet always reaches the `Dropped` state.

```
Fixpoint always_reaches_state_after_bounded_steps
  desired_state
  num_steps
  current_state
```

```
: ℙ :=
  desired_state = current_state ∨
  match num_steps with
  | 0 ⇒ ⊥
  | S num_steps' ⇒
    (∃ new_state, openflow_network_step current_state new_state) ∧
    ∀ new_state,
      openflow_network_step current_state new_state
        → always_reaches_state_after_bounded_steps
          desired_state
          num_steps'
          new_state
  end.

Record valid_openflow_entries
  (topology : network_topology)
  (policy : static_network_policy)
  (host_ip : host_ip_map)
  (entries : switch_openflow_entry_map)
: ℙ := {
  packets_arrive_iff_allowed : ∀ packet src_node dest_node,
    src_node.(host_ip) = packet.(IpSrc)
      → dest_node.(host_ip) = packet.(IpDest)
      → ∃ num_steps,
        always_reaches_state_after_bounded_steps
          host_ip
          entries
          topology
          packet
          (
            if policy {| Src := src_node; Dest := dest_node |}
            then Arrived
            else Dropped
          )
          num_steps
          NotSentYet;
  existent_ports : ∀ switch,
    all_ports_exist topology switch switch.(entries)
}.
```

To show that the compiler outputs a valid set of OpenFlow entries given a static policy, we prove the stronger claim that any valid `dec_next_node` function will result in valid OpenFlow entries. (Recall that we previously showed the compiler generates

valid `dec_next_node` functions via the all-pairs-paths mechanism.) We separate the proof into two cases depending on whether a given packet is allowed by the policy.

4.6.2 Correctness for Allowed Packets

First, we show that allowed packets always eventually arrive at their destinations. We use the fact that if a `next_node` relation is valid, then there must exist a path through the network from the packet source to the packet destination, for which which each hop is a valid routing decision under the given `next_node` relation. Since each hop under the `dec_next_node` function is deterministic, there must be exactly one such path for each starting switch adjacent to the source host, and the packet will simply move along that path until it reaches the destination. (This requires a fairly straightforward conversion from the semantics of routing tables to the modeled semantics of OpenFlow.)

4.6.3 Correctness for Disallowed Packets

Next, we show that disallowed packets are always eventually dropped. Analogously to the previous case, we use the fact that if a `next_node` relation is valid, then the `next_node` relation does *not* map any path through the network from the source to the destination. We can use this to show that disallowed packets never reach the `Arrived` state. (A packet can also never reach the `ReceivedAtHost` state because a valid `next_node` relation never forwards a packet to the wrong host.) Since the `next_node` relation never produces a cycle in a topology, a packet can never appear at the same switch multiple times on a route. Since there are a finite number of switches in the network, we can show that the packet must eventually be dropped by induction on the number of unvisited switches.

An alternate proof strategy could use the fact that `next_node` relations never contain cycles to achieve the same goal without relying on deterministic routing choices. This strategy could make the proofs more amenable to future modifications to the compiler. However, since the compiler relies on routing choices being deterministic

for other reasons, we did not pursue this proof strategy.

4.7 Modeling Stateful Policies

4.7.1 Transition Systems for Dynamic Policies and Controllers

At this point, we have shown that the compiler produces correct results for any given static network policy. We now expand the system to be able to support dynamic policies in addition to static policies.

Recall that *dynamic policies* can specify that the set of allowed flows must change at runtime, based on packets that have been sent through the network in the past. This necessitates that the OpenFlow controller occasionally update the set of routing rules on switches. We formally model a dynamic policy as a deterministic labelled transition system, where each state transition is triggered by a packet being sent in the network. We allow the dynamic policy to keep track of any arbitrary state, and require only that a static policy can be generated from that state at any given point in time.

```
Record flow_transition_system {state} := {
  Initial : state;
  Step : state → flow → state
}.

Record dynamic_network_policy {policy_state : Set} := {
  policy_system : flow_transition_system policy_state;

  policy_state_decider : policy_state → static_network_policy
}.
```

In our model, we assume that whenever a switch forwards a packet to a host (i.e. when the packet enters the `ReceivedAtHost` state), the switch simultaneously and instantaneously forwards the packet to the OpenFlow controller. The controller

receives the packet, updates some internal state, and generates a new set of OpenFlow entries based on the updated internal state. Finally, the controller sends the updated OpenFlow entries to all of the switches in the network.

More formally, we define a controller to be another transition system with its own internal state:

```
Record network_controller {controller_state} := {
  controller_system : flow_transition_system controller_state;

  controller_state_decider :
    controller_state → switch_openflow_entry_map
}.
```

4.7.2 Policy Validity

Some combinations of network topologies and static policies are trivially unsatisfiable. For example, if a static policy specifies that a flow must be allowed between two hosts, and the hosts are disconnected in the topology, it is trivially impossible to generate a network configuration that routes packets between the hosts.

More precisely, a topology-static policy pair is satisfiable iff every for every flow `(src, dest)`, the topology contains a path from `src` to `dest`. A user can prove that a given static policy is valid by generating a valid `all_pairs_paths` instance for that policy.

Checking the validity of a dynamic policy is somewhat more involved, because the set of flows allowed by the policy can change over time. As a result, the user is required to prove that a given `all_pairs_paths` instance is valid under every possible state of the dynamic policy.

In the previous section, we defined a dynamic policy as a transition system that can change state as a result of any flow. However, it is important to note that some transitions are not actually possible. At any given point in time, if a packet is not allowed by the static policy currently in effect, then the packet will necessarily be dropped by a switch before reaching its destination. As a result, the packet will never

enter the `ReceivedAtHost` state, so it will never trigger a state transition for the dynamic policy or the controller.

Allowing these impossible transitions makes it significantly more difficult (and sometimes impossible) to prove the validity of a policy, because it requires the creator of the policy to prove that the policy would remain valid even under transitions that cannot occur in practice.

To make these proofs more tractable, it is important to refine the transition system model to rule out impossible transitions. We accomplish this by wrapping the transition system in a "filter" that prevents it from changing state in response to certain flows.

```
Definition filter_transition_system
  {state}
  (predicate : state → flow → 𝔹)
  (sys : flow_transition_system state)
:=
  {|
    Initial := sys.(Initial);
    Step := λ current_state next_flow ⇒
      if predicate current_state next_flow
      then sys.(Step) current_state next_flow
      else current_state
  |}.
```

With this abstraction, we can restate our policy validity condition. Rather than requiring the policy to be valid under any transitions of the policy system, we instead require the policy to be valid under any transitions of a filtered policy transition system, where the only possible transition labels are the flows allowed by the current policy.

```
Inductive trc
  {state : Set}
  (step : state → flow → state)
: state → state → ℙ :=
| TrcRefl : ∀ (start : state), trc step start start
```

```
| TrcFront : ∀ start mid dest sent_flow,
  step start sent_flow = mid
    → trc step mid dest
    → trc step start dest
.

Definition invariant
  {state : Set}
  (sys : flow_transition_system state)
  (condition : state → ℙ)
:= ∀ new_state,
  trc sys.(Step) sys.(Initial) new_state
    → condition new_state.

Definition dynamic_policy_valid paths :=
  let filtered_policy_sys :=
    filter_transition_system
      policy.(policy_state_decider)
      policy.(policy_system) in
  invariant filtered_policy_sys (λ current_state ⇒
    all_pairs_paths_valid
      topology
      (policy.(policy_state_decider) current_state)
      paths
  ).
```

We can then model the entire dynamic controller and network by applying the same filter to both the policy and controller system, and have both systems run in lock-step. This is the final model on which we evaluate correctness of the compiler as a whole; the compiler is considered to produce correct output if and only if the current state of the controller generates valid OpenFlow entries for the current static policy, at every possible state of the network.

```
Definition join_transition_systems
  {A}
  {B}
  (sys₁ : flow_transition_system A)
  (sys₂ : flow_transition_system B)
:=
  {|
```

```
    Initial := (sys₁.(Initial), sys₂.(Initial));
    Step := λ current_state next_flow ⇒ (
      Step sys₁ current_state.(fst) next_flow,
      Step sys₂ current_state.(snd) next_flow
    )
  |}.

Definition controller_implements_policy
  {controller_state : Set}
  (controller : @network_controller controller_state)
:=
  let joined_sys :=
    filter_transition_system (λ current_state next_flow ⇒
      policy.(policy_state_decider) current_state.(fst) next_flow
    ) (
      join_transition_systems
        policy.(policy_system)
        controller.(controller_system)
    ) in
  invariant joined_sys (λ policy_and_controller_state ⇒
    let (policy_state, controller_state) :=
      policy_and_controller_state in
    valid_openflow_entries
      topology
      (policy.(policy_state_decider) policy_state)
      node_ip
      (controller.(controller_state_decider) controller_state)
  ).
```

Since we can already generate correct OpenFlow entries for any given static policy, this model makes it relatively simple to create a first-pass implementation of a correct controller. We can simply:

1. Simulate the dynamic policy transition system within the controller in order to determine the static policy that should be in effect at any given point in time.

2. Generate OpenFlow entries for that static policy, and send the entries to all of the switches.

In other words, while our model allows a controller to keep track of state separately from a dynamic policy (e.g. to allow for optimization by omitting irrelevant state),

it is not necessary to have separate state in the controller for a baseline correct implementation.

```
Definition dynamic_controller paths := {|
  controller_system := policy.(policy_system);
  controller_state_decider := λ state ⇒
  (
    generate_openflow_entries
    (
      exhaustive_routing_tables_generator
      (
        all_pairs_paths_dec_next_node_generator
          paths
          topology
          (policy.(policy_state_decider) state)
      )
      all_nodes
    )
    host_ip
    topology
  )
|}.
```

The proof of correctness for this dynamic controller is straightforward via induction on the `trc` relation.

4.8 Limitations of the One-Packet-at-a-Time Network Model

4.8.1 Comparison to Real Networks

Our model of a network has a few shortcomings. These issues could provide an interesting avenue for future work.

1. The model only tracks a single packet in the network at a time, and assumes a new packet only enters the network after the previous packet leaves. Of course, in real-life networks, many packets can be in flight simultaneously. As a result,

in practice the set of routing rules on switches could change while a packet is somewhere between two hosts. This has some implications for correctness:

- If the expected path through the network between two hosts is changed while a packet is in flight, the packet could end up getting routed to a black hole and dropped even if it would be allowed by both the old and new static policy (before and after the updates on the switches).
- Of greater concern: If the enforcement mechanism for disallowed packets is moved to an earlier point in the packet's path after the packet has already passed that point, then the packet could be successfully routed even if it would be disallowed by both the old and new static policy.

2. Updates from a controller are assumed to propagate to switches instantaneously. In reality, there is some latency between the time when a packet arrives at a controller, and the time when appropriate updates are installed on switches. This delay could result in a packet being processed by different switches with outdated versions of a particular policy.

   Currently, the compiler implements neither mutable paths nor centralized firewall enforcement, so it is not expected to suffer from the issues described above. However, our definition of correctness is not strong enough to guarantee resilience to these issues. Combining our model with a versioning system for policies, such mechanism introduced by Reitblatt et al. [13], would likely address this issue.

4.8.2 Efficiency

The efficiency of our network model also leaves some room for future improvement. In practice, it is undesirable for a switch to send a packet to the controller for every single packet that reaches its destination, since this creates substantial bandwidth overhead. It would be better for a packet to only be sent to the controller if the packet would actually cause the dynamic policy to transition

to a new state (since the controller does not need to be aware of packets that don't change the state).

This enhancement seems like it would be feasible with only a small change to the network model, although it would introduce some computation complexity for the controller. At any given state, the controller could simulate every possible flow in order to determine which packets, if subsequently sent in the network, would cause the controller to transition a different state. Then the controller could instruct switches to skip sending packets back to the controller unless the packets match one of the flows that would cause a state change. This brute-force approach would have at least quadratic runtime complexity in the number of hosts, since there are a quadratic number of possible network flows.

The quadratic runtime complexity here is not fundamental to the problem; it is simply a side effect of making the user-provided policy system opaque to the compiler backend. By allowing introspection of user-provided dynamic policies, the compiler backend could more efficiently determine which flows cause transitions to different states. Since Gallina terms cannot perform meaningful reflection over other Gallina terms, such a solution would either require the user to use an embedded language to specify policies, or it would require the compiler backend to pattern-match on user policies with something like Ltac.

Chapter 5

Components of the Compiler Frontend

5.1 Overall Architecture and User Interface

In discussing the architecture of the compiler backend, we have identified several cases where the user is required to prove certain properties about their input. (For example, in order to invoke the compiler backend, the user must provide a proof that their topology is valid.)

Proofs can be time-consuming to create, so it is desirable to minimize the amount of work that a user needs to do in order to meet their proof obligations. Of course, we cannot entirely eliminate proof obligations for input validity by proving a general case, because some user-provided inputs are in fact invalid, and the compiler would be unable to generate correct output for invalid inputs. Instead, we provide Ltac scripts that attempt to automatically generate proofs, or large portions of proofs, when given valid inputs. Ideally, these scripts greatly reduce the amount of work that a user needs to do in order to use the compiler.

Recall that the compiler is *proof-generating* rather than *proven correct*. In general, determining whether a proof exists for any given proposition is undecidable. As a result, while our Ltac scripts can produce proofs for a restricted subset of valid inputs, there will always be cases where the input is valid and the compiler frontend

nonetheless fails to generate validity proofs. (However, by construction there are no cases where the input is invalid and the compiler frontend successfully generates a validity proof.)

In the cases where the input is valid but the compiler frontend fails to generate a proof, a user can still run the compiler if they manually construct a validity proof with Ltac.[1] In this scenario, the compiler will still produce a proof of correctness for its output.

As a result, the frontend is best understood as a mechanism that aims to make the compiler easier to use in common cases, but it is not fundamental to the system and can be ignored or augmented by the user if necessary. We will see that the frontend is composed of multiple independent components, allowing a user to use some parts of the frontend but not others.

5.2 Comparing the Compiler Frontend Architecture with Alternative Designs

At first glance, it may seem unappealing that our compiler sometimes requires users to manually write proofs. One might imagine a different design in which we require policies to be provided in a restricted language that allows for decidable proof generation. This alternative approach would allow us to ensure that the compiler always succeeds for valid inputs without any manual proof steps (in effect, allowing the compiler to be proven correct rather than proof-generating).

However, we consider it to be important that users can declare policies in a high-level functional language such as Gallina, for several reasons:

- First, it is useful for the policy language to have substantial expressive power. By necessity, some network systems have policies depending on complex invariants, which might not be expressible in a restricted language. Our approach allows users to specify policies as arbitrary Gallina predicates. In effect, we trade

[1] A user can alternatively declare the validity of their input as an axiom instead of proving validity, but this is not recommended because it somewhat defeats the point of using a verified compiler.

a small amount of convenience for greater expressive power. (We note that the "simple" cases where such expressive power is unneeded are also roughly the cases where the compiler frontend can successfully automate proofs anyway.)

- Second, user-provided policies are part of the trusted computing base for our system, making it prudent to minimize the risk of bugs in policies. While we can prove that the output of the compiler satisfies a given policy, we cannot prove that a given policy accurately expresses the user's intent. We believe that our use of a high-level functional language reduces the risk of specification error, because it allows policies to be declared at a level of abstraction that closely corresponds to the user's mental model. If we instead required policies to be provided in a restricted lower-level language, a user would first need to convert their mental model of the correct behavior into the low-level policy language, and manually reason about whether the conversion was correct.

  In effect, the undecidable "compilation" from a high-level mental model to a proof of input validity still needs to occur regardless of our input language. By requiring the user to construct input validity proofs in Coq, we simply automate some of the logical reasoning that would otherwise have been performed mentally in an error-prone manner. The result is that in comparison to many existing systems with untyped input languages, our user interface substantially reduces the number of cases where input bugs can appear.

5.3 Proof Obligations

5.3.1 Topology Validity

We require the user to provide a network topology, as well as a proof that the topology is valid. Recall that a topology is considered to be valid if no host has two outgoing connections on the same port, and each host is connected to at least one switch.

In order to show that no host has two outgoing connections on the same port, we can simply use a brute-force approach of enumerating all hosts, enumerating all

nodes adjacent to each given host, and verifying that the outgoing port numbers are not equal. Similarly, we can show that each host is connected to at least one switch by simply using "nested loops" to enumerate all hosts and switches.

Note that since validity proofs only need to be constructed once at compile time, we are largely unconcerned with the runtime complexity of constructing proofs. (To optimize user experience, it is somewhat desirable for proof construction to take no more than a few seconds. Beyond that metric, however, we have not placed a high priority on performance optimization of proofs beyond informally ensuring that they run in polynomial time. There is likely to be substantial room for improvement in our proof performance.)

5.3.2 Policy and All-Pairs-Paths Validity

We require the user to provide a dynamic network policy and an `all_pairs_paths` implementation, along with a proof that the `all_pairs_paths` implementation is valid for any reachable state of the dynamic policy. Recall that an `all_pairs_paths` implementation is considered to be valid if its paths follow edges in the topology, it provides paths for all allowed flows in the current static policy, and it generates decreasing costs.

We provide an Ltac script that runs the Floyd-Warshall algorithm with path reconstruction on the user's topology in order to generate an `all_pairs_paths` implementation. We allow the user to provide a set of edge costs for use in the Floyd-Warshall algorithm, which can optimize the set of chosen paths to account for real-world link costs. While the Ltac script is expected to generate *shortest* paths (weighted by the provided edge costs) between each pair of nodes, we do not provide a proof that the path costs are indeed optimal.

After generating the `all_pairs_paths` implementation, our Ltac script can prove that its paths follow the edges in the topology by exhaustively verifying that this is the case for every step of every generated path. We can similarly show that the `all_pairs_paths` implementation generates decreasing costs by computing the path cost at each step of each path, and ensuring that the cost decreases after moving to

the next node along the path.

Finally, we need to prove that for every reachable static policy, the `all_pairs_paths` implementation generates a path for every flow allowed by that static policy. (Since our Ltac script will always generate a path if a path exists, this is equivalent to proving that if any reachable policy state allows a particular flow, then there is a path in the graph between the source and destination of the flow.) If the user's topology happens to be strongly connected, then this proof is trivial because a path exists between every pair of nodes.

However, if the topology is not strongly connected, then this proof is significantly more complex. We cannot use exhaustive search because a user-provided dynamic policy can be an arbitrary transition system, with a potentially infinite state space. Instead, we require the user to supply a condition as an invariant for their dynamic policy, as well as a proof that the condition implies the desired goal at any given state, and a proof that the condition is indeed an invariant of the system.

5.3.3 Host-IP Address Mapping

The user is required to provide an injective mapping from hosts to IP addresses. We prove that the user's mapping is injective by exhaustively enumerating each distinct pair of nodes and showing that the corresponding IP addresses are also distinct.

5.3.4 Enumeration of Nodes

The user is required to provide an explicit list of hosts and nodes in the network, as well as a proof that their list indeed contains all nodes, and a proof that their list contains no duplicates. In the simplest case where the `Switch` and `Host` types are provided as a set with only nullary constructors, we can start from the proof goal that every switch and host is in a particular list, and then use refinement with existential variables to enumerate all constructors and add them to a list one at a time. Then all that remains is to show that the generated list contains no duplicates, which is trivial because the list has no free variables.

We do not provide a proof script for cases where `Switch` and `Host` have constructors with arguments. However, such a script seems feasible to create if all constructor argument types are themselves enumerable.

Chapter 6

I/O Translation Layer

The final piece of the compiler is an I/O layer written in OCaml. In order to send messages to external OpenFlow switches, an OpenFlow controller needs to run actions with side effects (e.g. by invoking syscalls). Coq provides no direct mechanism to perform actions with side effects, so we cannot run an OpenFlow controller directly from Coq. Instead, we use Coq's extraction feature, which allows Gallina code to be exported to a language like OCaml with proof terms omitted. We then compile the generated OCaml code with our I/O layer in order to run it on a real network.

Broadly speaking, the I/O layer involves serializing our formal datatype for an OpenFlow controller into a format that can be sent over the wire. The OCaml controller listens for messages from switches, simulates the abstract controller model generated from the Coq code while keeping track of state, and then sends OpenFlow messages back to switches to produce the updated network state. To simplify the controller, we use the set of OpenFlow bindings for OCaml created for use in Frenetic [5].

6.1 Evaluation of the I/O Layer

Our I/O layer is effective as a proof of concept for running our abstract network rules on a real network. However, some modification would likely be necessary before using our controller at scale. For example, in order to update the set of OpenFlow flow

entries on the network at runtime, our OCaml controller simply deletes all flow entries on all switches, then computes a new batch of flow entries for the updated state and sends all of these entries to the appropriate switches.

This behavior was chosen in order to simplify the OCaml controller while developing the Coq model, but it would likely be too inefficient to use in a large network. A better solution would likely involve computing a diff between the old and new set of flow entries at any given switch, and only sending addition and deletion messages for flow entries that changed.

6.2 Impact of Unverified Components on Correctness

Unlike the components of the compiler written in Coq, our OCaml I/O layer is not formally verified. This introduces a risk that the correctness of the compiler could be compromised by a bug in the I/O layer. To minimize this risk, we generally tried to include and verify as much logic as possible in Coq instead of OCaml.

Even with these efforts, there were some cases where it was possible to move logic to Coq but we did not do so. For example, our OCaml code includes a library that performs serialization and deserialization between OpenFlow objects and a binary format, based on the semantics described in the OpenFlow spec. Fundamentally, this serialization does not involve any kind of I/O, and we could have implemented it in Coq and proven its correctness. However, we did not do this in the interest of prioritizing other concerns.

Additionally, the correct behavior of the compiler depends on the correctness of a number of external components which are necessarily external to the controller. For example, we assume the controller's operating system, and the hardware that it runs on, have no bugs that affect the behavior of the controller. We also assume that all of the network switches are actually connected in the manner specified by the user's abstract network topology, and that no one has cut the cables.

A bug in any of these components could, in theory, create incorrect behavior when placed in a real network. There are ongoing efforts in other projects to formally verify an entire software/hardware stack, and we hope these efforts make it easier in the future to create a verified system with fewer assumptions on correct infrastructure. However, we consider these issues to be out-of-scope for our compiler. In general, any formally verified system models some assumptions about components in the real world, and may work incorrectly if those assumptions are violated. Even with these caveats, we believe our compiler still significantly decreases the risk of bugs when configuring a network, by simply reducing the number of places where a bug could feasibly appear.

Chapter 7

Evaluating the Compiler in Practice

To test the behavior of the compiler in practice, we first generated contrived sample network topologies by hand. We modeled the corresponding topologies in Coq, created a basic policy for them, and ran the necessary inputs through the compiler frontend. Then we used the Mininet [8] tool to simulate our network in practice. We ran the compiler backend, compiled the result with the OCaml I/O layer, and verified that the network behaved as expected on a few simple packet communications. These tests were useful for improving the compiler frontend and addressing modeling problems.

We found a few minor issues when evaluating the tool on a real network:

- In a few cases, our model of an OpenFlow network inadvertently differed from the OpenFlow spec. For example, at one point our model allowed two switches to be connected on port number 0 and assumed that forwarding would happen as normal. In reality, port 0 is not a valid port number in OpenFlow,[1] so the network did not behave as expected even though the compiler was ostensibly proven correct. As with any formal-verification project, while we can prove correctness with respect to a particular model, it is important to separately

[1]See Section 3.1 of the errata for OpenFlow v1.0.1 [17]

ensure that the model accurately reflects the real world for the relevant purposes.

- While our topology model allows for one-way links between nodes in a topology, Mininet only supports two-way links out of the box. As a workaround for this issue, we simply represented one-way links in our topology with two-way links in Mininet, potentially influencing the correctness of our tests. (This issue is an inconvenience for testing the compiler, but it does not influence the correctness of the compiler itself because topologies are provided by the user. If a real-world topology only contains two-way links, then the user could simply use two-way links when specifying the topology in Gallina.)

- We found that our network would sometimes unexpectedly drop packets due to delays in controller behavior. For example, at one point we evaluated a network policy where packets from `hostA` to `hostB` were always allowed, and packets traveling in the opposite direction (from `hostB` to `hostA`) were only allowed if `hostB` had previously received a packet from `hostA`. At first glance, it might appear that running something like `ping hostB` from `hostA` should work without any problems, since the echo packet is always allowed, and the echo response packet would start being allowed after the successful transmission of the echo packet. However, in practice we found that the response for the first ping packet would always be dropped, because `hostB` would always emit a ping response before the OpenFlow controller was able to update the routing rules in the network. After the first ping packet, subsequent pings were successful, as expected.

  In other words, our model assumed that switches and controllers communicate synchronously and instantaneously, when in fact they communicate asynchronously and with some delay. In this particular case, we do not consider the loss of a single packet to be a major problem; it is generally understood that networks sometimes unexpectedly drop packets, and hosts should use reliable protocols like TCP to account for the possibility of dropped packets. However, this issue suggests a fundamental limitation of the one-packet-at-a-time,

synchronous network model.

Chapter 8

Conclusions and Future Work

We present a verified compiler from high-level abstract network policies down to low-level routing decisions. This compiler can be used to generate executable routing rules on real-world switches through a very expressive policy language, and a set of proof scripts that ensure the absence of certain types of errors in the input. The executable routing rules are accompanied by a formal, machine-checked proof of correctness, and can be automatically run on a real network through compilation with a prebuilt generic controller.

While we believe the tool is already useful today in many cases, there are substantial avenues for future improvements that could make the tool more expressive, more correct, and more efficient. (Some of these potential improvements are described in more detail in other sections.)

- We could enhance the policy language and our network to allow for policy changes that expire after a certain amount of time.
- We could update the compiler implementation to apply routing enforcement more efficiently, by centralizing routing decisions to a smaller number of switches. We could also generate more efficient sets of routing rules at each given switch (for example, by creating a single routing rule for an IP mask rather than creating a different routing rule for each host). In combination, these changes would likely improve network throughput by reducing the amount

of redundant computation needed at each switch.

- We could update the compiler implementation to allow for routing decisions at each switch that depend on something other than the source/host IP addresses. For example, this could allow a switch to act a load balancer by evenly distributing packets among multiple possible routes.

- We could broaden our network model and correctness analysis to account for the reality that real-world networks process multiple packets simultaneously, and have asynchronous control-plane communication.

- We could enhance our network model to cover additional real-world use cases, such as network address translation or reasoning about the behavior of uncontrolled third-party switches.

- We could make our I/O layer more efficient by avoiding the redundant transmission of routing rules that switches already have.

Ideally, all networking tools would be powerful enough to provide very strong correctness guarantees, and also ergonomic enough to achieve widespread adoption. While significant progress has been made towards this goal in the past decade, there is still a great deal of work to be done. As the world comes to depend on the stability and security of networks, it becomes increasingly important that our networks are robust enough to support the world's needs.

Bibliography

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[2] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

[3] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: Towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 282–293, New York, NY, USA, 2014. ACM.

[4] David Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning*, pages 85–95, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[5] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[6] Inria. Calculus of Inductive Constructions. https://coq.inria.fr/distrib/current/refman/language/cic.html, 2018.

[7] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 15–27, Lombard, IL, 2013. USENIX.

[8] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIG-*

COMM Workshop on Hot Topics in Networks, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[9] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 27:163–27:170, Austin, TX, 2014. FMCAD Inc.

[10] Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. Decentralizing SDN policies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 663–676, New York, NY, USA, 2015. ACM.

[11] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 699–718, Boston, MA, 2017. USENIX Association.

[12] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 109–114, New York, NY, USA, 2013. ACM.

[13] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[14] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the network with Merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 24:1–24:7, New York, NY, USA, 2013. ACM.

[15] Brad Stone. Pakistan cuts access to YouTube worldwide. *The New York Times*, February 2008.

[16] John D. Sutter. GoDaddy: Outage did not result from a hack. *Cable News Network*, September 2012.

[17] The Open Networking Foundation. OpenFlow switch errata. https://www.opennetworking.org/wp-content/uploads/2013/07/openflow-spec-v1.0.1.pdf, 2012.

[18] The Open Networking Foundation. OpenFlow switch specification. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, 2015.