Machine-Verified Network Controllers

Arjun Guha
Cornell University
arjun@cs.cornell.edu

Mark Reitblatt
Cornell University
reitblatt@cs.cornell.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Abstract

In many areas of computing, techniques ranging from testing to formal modeling to full-blown verification have been successfully used to help programmers build reliable systems. But although networks are critical infrastructure, they have largely resisted analysis using formal techniques. Software-defined networking (SDN) is a new network architecture that has the potential to provide a foundation for network reasoning, by standardizing the interfaces used to express network programs and giving them a precise semantics.

This paper describes the design and implementation of the first machine-verified SDN controller. Starting from the foundations, we first develop a detailed operational model for OpenFlow (the most popular SDN platform) and formalize it in the Coq proof assistant. Building on this model, we then develop a verified compiler and run-time system for a high-level network programming language. We identify bugs in existing languages and tools built without formal foundations and prove that these bugs are absent from our system. Finally, we describe our prototype implementation and our experiences using it to build practical examples.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

Keywords Software-defined networking, OpenFlow, formal verification, Coq, domain-specific languages, NetCore, Frenetic.

1. Introduction

Networks are some of the most critical infrastructure in modern society and also some of the most fragile! Networks fail with alarming frequency, often due to simple misconfigurations or software bugs [7, 17, 28]. The recent news headlines contain numerous examples of network failures leading to disruptions: a configuration error during routine maintenance at Amazon triggered a sequence of cascading failures that brought down a datacenter and the customer machines hosted there; a corrupted routing table at GoDaddy disconnected their domain name servers for a day and caused a widespread outage; and a network connectivity issue at United Airlines took down their reservation system, leading to thousands of flight cancellations and a "ground stop" at their San Francisco hub.

One way to make networks more reliable would be to develop tools for verifying network invariants automatically. These tools would allow network administrators to answer questions such as: "does this configuration provide connectivity to every host in the network?" or "does this configuration correctly enforce the access control policy?" or "does this configuration have a forwarding loop?" or "does this configuration properly isolate trusted and untrusted traffic?" Unfortunately, building such tools today is effectively impossible due to the complexity of current networks. A typical enterprise or datacenter network contains thousands of heterogeneous devices, from routers and switches, to web caches and load balancers, to monitoring middleboxes and firewalls. Moreover, each device executes a stack of distributed protocols and are configured separately through proprietary and idiosyncratic interfaces. To reason formally about such a network, an administrator (or verification tool) must reason about the proprietary programs running on each distributed device, as well as the asynchronous interactions between them. Although formal models of traditional networks exist, they have been either too complex to allow effective reasoning, or too abstract to be useful. Overall, incidental complexity has made reasoning about networks practically infeasible.

Fortunately, recent years have seen growing interest in a new network architecture that could provide a basis for building formal network reasoning tools. In a *software-defined network* (SDN), a program on a logically-centralized *controller machine* defines the overall policy for the network, and a collection of *programmable switches* implement the policy using efficient packet-processing hardware. The controller and switches communicate via an open and standard interface. By carefully installing packet-processing rules in the hardware tables provided on switches, the controller can effectively manage the behavior of the entire network.

Compared to traditional networks, SDNs employ two important simplifications that make them amenable to formal foundations and verification. First, they relocate control from distributed algorithms running on individual devices to a single program running on the controller. Second, they eliminate the various heterogeneous devices used in traditional networks—switches, routers, load balancers, firewalls, etc.—and replace them with stock programmable switches that provide a standard set of features. Together, these simplifications mean that the behavior of the network is determined solely by the sequence of configuration instructions issued by the controller. Hence, to verify that the network has some property, an administrator (or tool) simply has to reason about the states of the switches as they process controller instructions.

In the networking community, there is burgeoning interest in tools that check network-wide properties automatically. Systems such as FlowChecker [1], Header Space Analysis [10], Anteater [15], VeriFlow [11], and others, work by generating a logical representation of switch configurations and using an automatic solver to check properties of those configurations. The configurations are obtained either by "scraping" state off of the switches or by inspecting the sequence of instructions issued by an SDN controller at run-time.

These tools represent a good first step toward making networks more reliable, but they have two important limitations. First, they are based on ad hoc foundations. Although SDN platforms such as

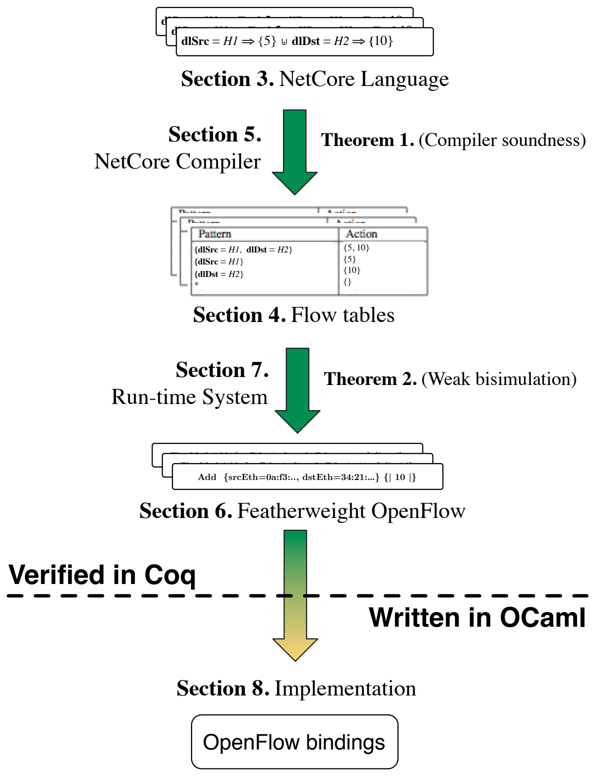

dlSrc = H1 ⇒ {5} ⊎ dlDst = H2 ⇒ {10}
Section 3. NetCore Language
Section 5. NetCore Compiler
Theorem 1. (Compiler soundness)
Pattern
Action
{dlSrc = H1, dlDst = H2}
{5, 10}
{dlSrc = H1}
{5}
{dlDst = H2}
{10}
*
{}
Section 4. Flow tables
Section 7. Run-time System
Theorem 2. (Weak bisimulation)
Add {srcEth=0a:f3:.., dstEth=34:21:...} {| 10 |}
Section 6. Featherweight OpenFlow
Verified in Coq
Written in OCaml
Section 8. Implementation
OpenFlow bindings

Figure 1: System architecture.

OpenFlow [19] have precise (if informal) specifications, the tools make simplifying assumptions that are routinely violated by real hardware switches. For example, some tools assume that switches will process instructions emitted by the controller in sequence, even though actual switches often reorder messages. This means that properties they consider verified do not always hold. Second, the tools are expensive to run and do not scale well. For example, most tools take several minutes to run, even on small to medium-sized networks (VeriFlow [11] is a notable exception). This is too slow to be used in large dynamic networks where configurations change on the order of seconds. Overall, although these tools are useful for finding bugs, they can not provide the rigorous guarantees that networks, as critical infrastructure, require.

Our approach. This paper presents a different approach. Rather than building tools to find bugs in SDN controllers at run-time, we develop a verified SDN controller in the Coq proof assistant and prove it correct against a formal specification and a detailed operational model of SDN. With our controller, programmers specify the behavior of the network using the NetCore programming language [20], which abstracts away from the details of the underlying switch hardware and distributed system, and allows programmers to reason in terms of simple hop-by-hop packet processing steps. The NetCore compiler and run-time system translates programs written in this language down to low-level packet-processing rules. Because its behavior is verified in Coq, we establish the correctness of our controller once and for all, obviating the need for run-time or post hoc verification as in most current tools.

Architecturally, our system is organized as a verified software stack that translates through the following levels of abstraction:

- **NetCore.** The highest level of abstraction is the NetCore language, proposed in prior work by Monsanto *et al.* [20]. NetCore is a declarative language that allows programmers to describe *what* network behavior they want, without specifying *how* it should be implemented. It offers a collection of intuitive constructs for matching, filtering, and transforming packets, as well as natural logical operators for combining smaller programs into bigger ones such as union and domain restriction. Although NetCore programs are ultimately executed in a distributed system—the network—they have a simple semantics that models their behavior as functions from packets to packets.
- **Flow tables.** The intermediate level of abstraction is *flow tables*, a representation that sits between NetCore programs and switch-level configurations. There are two main differences between NetCore programs and flow tables. First, NetCore programs describe the forwarding behavior of a whole network, while a flow table describes the behavior of a single switch. Second, flow tables process packets using a linear scan through a list of prioritized rules. Hence, to translate operators such as union and negation, the NetCore compiler must generate a sequence of rules that encodes the same semantics. However, because flow table matching uses a lower-level packet representation (as nested frames of Ethernet, IP, TCP, etc. packets), flow tables must satisfy a well-formedness condition to rule out invalid patterns that are inconsistent with this representation.
- **Featherweight OpenFlow.** The lowest level of abstraction is *Featherweight OpenFlow*, a new foundational model we have designed that captures the essential features of SDNs. Featherweight OpenFlow models switches, the controller, the network topology, as well as their internal transitions and interactions in a small-step operational semantics. This semantics is non-deterministic, modeling the asynchrony inherent in networks. To implement a flow table in a Featherweight OpenFlow network, the controller run-time system instructs switches to install or uninstall rules as appropriate. This run-time system deals with two important issues: first, switches process instructions concurrently with packets flowing through the network, so it must ensure that at all times the rules installed on switches are consistent with the flow table. Second, switches are allowed to buffer instructions and apply them in any order, so it must ensure that the behavior is correct no matter how instructions are reordered through careful use of synchronization primitives.

Figure 1 depicts the architecture of our system, and also provides an outline for this paper. Overall, the main contributions of this paper are as follows:

- We present the first machine-verified SDN controller, which gives network programmers robust static guarantees backed by machine-checked proofs against a foundational model.
- We develop Featherweight OpenFlow, the first formal model of OpenFlow. It includes all sources of asynchrony and nondeterminism mentioned in the informal specification, as well as a precise model of the switch flow table semantics.
- We formalize NetCore, flow tables, and Featherweight OpenFlow in Coq, and develop machine-checked proofs of correctness for the translations between them.
- We present our prototype implementation, obtained by extracting our Coq development to OCaml, and present experimental results comparing the performance of our system against an unverified controller on simple benchmark applications.

Besides their use in our system, we hope that the abstractions and theorems presented in this paper will be useful to others. Flow tables are a canonical representation of switch state that appear in many other systems. Likewise, Featherweight OpenFlow is a comprehensive model that captures the essential forwarding behavior of SDNs in a minimal core calculus. Our design and Coq

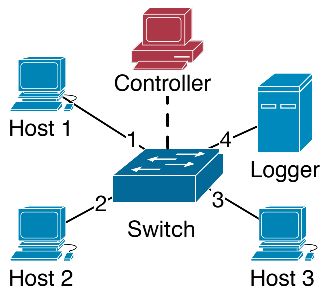


Figure 2: Example network topology.

formalization of flow tables and Featherweight OpenFlow provide a starting point for developing extensions that model additional SDN features and a foundation for other verified SDN systems.

2. Overview

To motivate the need for verified SDN controllers, consider the simple network depicted in Fig. 2. It consists of a single switch connected to four hosts: three clients and a middlebox that monitors HTTP requests. The ports on the switch are numbered 1 to 4; the clients are connected to ports 1 to 3 and the middlebox to port 4. In addition, the switch has a dedicated link to the controller that is not considered part of the data network.

Now imagine we want to build an SDN controller that implements the following high-level network policy: block SSH traffic, log HTTP requests, and allow clients to send non-SSH traffic to each other. It is straightforward to formalize this policy as a packet-processing function that maps input packets to (possibly several) output packets: the function drops SSH packets, forwards HTTP packets to their destination and the middlebox, and forwards all other packets to their destination alone.

To implement this function, however, we would need to specify several additional low-level details, since switches cannot implement general packet-processing functions directly. First, the controller would need to encode the function as a *flow table*—a set of prioritized pattern-action pairs. Second, it would need to send the switch a series of control messages to add individual entries from the flow table, incrementally building up the complete table.

More concretely, the controller could first send a message instructing the switch to add a flow table entry that blocks SSH traffic:

$$\textbf{Add}\ 10\ \{\textbf{tpDst} = 22\}\ \{\!|\,|\!\}$$

Here 10 is a priority number, $\{\textbf{tpDst} = 22\}$ is a pattern that matches SSH traffic (TCP port 22), and $\{\!|\,|\!\}$ is an empty set of ports, which drops packets, as intended. Next, the controller could add an entry to process HTTP requests:

$$\textbf{Add}\ 9\ \{\textbf{dlDst} = \mathit{H1}, \textbf{tpDst} = 80\}\ \{\!|1, 4|\!\}$$

Note that this rule duplicates HTTP (TCP port 80) packets, sending them to the monitor and their destination.[1] Finally, the controller could add an entry to forward other packets to their destination:

$$\textbf{Add}\ 1\ \{\textbf{dlDst} = \mathit{H1}\}\ \{\!|1|\!\}$$

Note that this rule does not apply to SSH and HTTP traffic, since those packets are handled by the higher-priority rules.

After these control messages have been sent, it would be natural to expect that the network correctly implements the packet-processing function described above. But the situation is actually more complicated: switches have substantial latitude in how they process messages from the controller, and packets may arrive at any time during processing. Establishing that the network correctly implements this function—in particular, that it blocks SSH traffic and logs HTTP traffic—requires additional reasoning.

[1] The controller would actually need to create rules for each client. To save space, we have only given the rules for *H1* here.

Packet	pk	$::=$	**Eth** *dlSrc dlDst dlTyp* $nwPk$
Network layer	$nwPk$	$::=$	**IP** *nwSrc nwDst nwProto* $tpPk$
		$\mid$	**Unknown** $payload$
Transport layer	$tpPk$	$::=$	**TCP** *tpSrc tpDst payload*
		$\mid$	**Unknown** $payload$

Figure 3: Logical packet structure.

Controller-switch consistency. SDN switches process packets and control messages concurrently. In our example, the switch may receive an HTTP request packet before the flow table entry that handles HTTP packets arrives. In this situation, the switch will send the packet to the controller for further processing. Since the controller is a general-purpose machine, it can implement the packet-processing function directly, apply it to the incoming packet, and send the results back to the switch. However, this means that SDN controllers typically have two *different* implementations of the function: one residing at the controller and another on the switches. A key property we verify is that these two implementations are consistent.

Message reordering. SDN switches may process control messages in any order, and many switches do to maximize performance. But unrestricted reordering can cause implementations to violate their intended specifications. For example, if the rule to drop SSH traffic is installed after the final, low-priority rule that forwards all traffic, then SSH traffic will temporarily be forwarded by the low-priority rule, breaking the intended security policy. To ensure that such reorderings do not occur, a controller must carefully insert *barrier messages*, which force the switch to process all outstanding messages. A key property we verify in that controllers use barriers correctly (several unverified controllers ignore this issue).

Natural patterns. Another complication is that the patterns presented earlier in this section, such as $\{\textbf{tpDst} = 22\}$, are actually invalid. To match SSH traffic, it is not enough to simply state that the destination port must be 22. The pattern must also specify that the Ethernet frame type must be IP, and the transport protocol must be TCP. Without these additional constraints, switches will interpret the pattern as a wildcard that matches all packets. Several earlier controller platforms did not properly account for this behavior, and had bugs as a result. We develop a semantics for patterns and identify a class of *natural patterns* that are closed under the algebraic operations used by our compiler and flow table optimizer.

Roadmap. The rest of this paper develops techniques for establishing that a given packet-processing function is implemented correctly by an OpenFlow network. More specifically, we tackle the problem of verifying high-level programming abstractions, using NetCore [20] as a concrete instance of a high-level network language. The next section presents NetCore in detail. The following sections describe general and reusable techniques for establishing the correctness of SDN controllers, including NetCore.

3. NetCore

This section presents the highest layer of our verified stack: the NetCore language. A NetCore program specifies how the switches process packets at each hop through the network. More formally, a

Switch ID	sw	$\in \mathbb{N}$	
Port ID	pt	$\in \mathbb{N}$	
Headers	h	$::=$ **dlSrc** | **dlDst**	MAC *address*
		| **dlTyp**	*Ethernet frame type*
		| **nwSrc** | **nwDst**	IP *address*
		| **nwProto**	IP *protocol code*
		| **tpSrc** | **tpDst**	*transport port*
Predicate	pr	$::= \star$	*wildcard*
		| $h = n$	*match header*
		| **at** sw	*match switch*
		| **not** pr	*predicate negation*
		| pr_1 **and** pr_2	*predicate conjunction*
Program	pg	$::= pr \Rightarrow \{\!\mid pt_1 \cdots pt_n \mid\!\}$	*basic program*
		| $pg_1 \uplus pg_2$	*program union*
		| **restrict** pg **by** pr	*program restriction*

$$\boxed{[\![pr]\!]\ sw\ pt\ pk}$$

$$
\begin{array}{l}
[\![\star]\!]\ sw\ pt\ pk = \mathbf{true}\\
[\![\mathbf{dlSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ dlSrc\ \_\ \_\ \_) = dlSrc{=}n\\
[\![\mathbf{nwSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ \_\ \_\ \_\ (\mathbf{IP}\ nwSrc\ \_\ \_\ \_)) = nwSrc{=}n\\
[\![\mathbf{nwSrc}{=}n]\!]\ sw\ pt\ (\mathbf{Eth}\ \_\ \_\ \_\ (\mathbf{Unknown}\ \_)) = \mathbf{false}\\
\qquad\qquad \cdots\\
[\![\mathbf{at}\ sw']\!]\ sw\ pt\ pk = sw{=}sw'\\
[\![\mathbf{not}\ pr]\!]\ sw\ pt\ pk = \neg([\![pr]\!]\ sw\ pt\ pk)\\
[\![pr_1\ \mathbf{and}\ pr_2]\!]\ sw\ pt\ pk = [\![pr_1]\!]\ sw\ pt\ pk \wedge [\![pr_2]\!]\ sw\ pt\ pk
\end{array}
$$

$$\boxed{[\![pg]\!]\ sw\ pt\ pk = \{\!\mid (pt_1, pk_1) \cdots (pt_n, pk_n) \mid\!\}}$$

$$
\begin{array}{l}
[\![pr \Rightarrow \{\!\mid pt_1 \cdots pt_n \mid\!\}]\!]\ sw\ pt\ pk =\\
\quad \mathbf{if}\ [\![pr]\!]\ sw\ pt\ pk\ \mathbf{then}\ \{\!\mid (pt_1, pk) \cdots (pt_n, pk) \mid\!\}\ \mathbf{else}\ \{\!\mid\,\mid\!\}\\
[\![pg_1 \uplus pg_2]\!]\ sw\ pt\ pk =\\
\quad [\![pg_1]\!]\ sw\ pt\ pk \uplus [\![pg_2]\!]\ sw\ pt\ pk\\
[\![\mathbf{restrict}\ pg\ \mathbf{by}\ pr]\!]\ sw\ pt\ pk =\\
\quad \{\!\mid (pt', pk') \mid (pt', pk') \in [\![pg]\!]\ sw\ pt\ pk \wedge [\![pr]\!]\ sw\ pt\ pk \mid\!\}
\end{array}
$$

Figure 4: NetCore syntax and semantics (extracts).

program denotes a total function from port-packet pairs to multisets of port-packet pairs. The syntax and semantics of a core NetCore fragment are shown in Fig. 4. To save space, we have elided several fields and operators not needed for the examples in this paper.

We can build a NetCore program that implements the example from the previous section by composing several smaller NetCore program fragments. The first fragment forwards traffic to *H1*:

$$pg_1 \triangleq \mathbf{dlDst}{=}H1 \Rightarrow \{\!\mid 1 \mid\!\}$$

This basic program consists of a predicate pr and a multiset of actions $\{\!\mid pt_1 \cdots pt_n \mid\!\}$. The predicate denotes a set of port-packet pairs, and the actions denote the ports (if any) where those packet should be forwarded on the next hop. In this instance, the predicate denotes the set of all packets whose Ethernet destination (**dlDst**) address has the specified value, and the actions denote a transformation that forwards matching packets to port 1. Note that we represent packets as nested sequences of frames (Ethernet, IP, TCP, etc.) as shown in Fig. 3. NetCore provides predicates for matching on well-known header fields as well as logical operators such as **and** and **or**, unlike hardware switches, which only provide prioritized sets of rules.

The next two fragments are similar to pg_1, but forward traffic to *H2* and *H3* instead of *H1*:

$$pg_2 \triangleq \mathbf{dlDst}{=}H2 \Rightarrow \{\!\mid 2 \mid\!\}$$
$$pg_3 \triangleq \mathbf{dlDst}{=}H3 \Rightarrow \{\!\mid 3 \mid\!\}$$

Using the union operator, we can combine these programs into a single program that implements forwarding between all clients:

$$pg_{\mathit{fwd}} \triangleq pg_1 \uplus pg_2 \uplus pg_3$$

Semantically, the $\uplus$ operator produces the (multiset) union of the results produced by each sub-program. Using the union operator again, we can extend this program to one that also forwards HTTP requests to the middlebox:

$$pg_{\mathit{fwd}} \uplus\ \mathbf{tpDst}{=}80 \Rightarrow \{\!\mid 4 \mid\!\}$$

Note that this program duplicates packets sent to port `80`, forwarding to their destination and also to the logging machine. Finally, we can add the security policy using the **restrict by** operator, which restricts a program by a predicate:

$$\mathbf{restrict}\ (pg_{\mathit{fwd}} \uplus\ \mathbf{tpDst}{=}80 \Rightarrow \{\!\mid 4 \mid\!\})\ \mathbf{by}\ (\mathbf{not}\ \mathbf{tpDst}{=}22)$$

This program is similar the previous one, but drops SSH traffic.

The advantages of using a declarative language such as NetCore should be clear: it provides abstractions that make it easy to establish network-wide properties through compositional reasoning. For example, simply by inspecting the final program and using the denotational semantics (Fig. 4), we can easily verify that the network blocks SSH traffic, forwards HTTP traffic to the middlebox, and provides pair-wise connectivity between the clients. In particular, even though a controller, switches, flow tables, forwarding rules, are all involved in implementing this program, we do not have to reason about them! This is in contrast to lower-level controller platforms, which require programmers to explicitly construct switch-level forwarding rules, issue messages to install those rules on switches, and reason about the asynchronous interactions between switches and controller. Of course, the complexity of the underlying system is not eliminated, but relocated from the programmer to the language implementors. This is an efficient tradeoff: functionality common to many programs can be implemented just once, proved correct, and reused broadly.

4. Flow Tables

The first step toward executing a NetCore program in an SDN involves compiling it to a prioritized set of forwarding rules—a *flow table*. Flow tables are an intermediate representation that play a similar role in NetCore to register transfer language (RTL) in traditional compilers. Flow tables are more primitive than NetCore programs because they lack the logical structure induced by NetCore operators such as union, intersection, negation, and restriction. In addition, the patterns used to match packets in flow tables are more restrictive than NetCore predicates. Finally, unlike NetCore programs, which denote total functions, flow tables are partial: switches redirect unmatched packets to the controller.

As defined in Fig. 5, a *flow table* consists of a multiset of rules (n, pat, pts) where n is an integer priority, pat is a pattern, and pts is a multiset of ports. A *pattern* is a record that associates each header field to either an integer constant n or the special *wildcard* value $\star$. When writing flow tables, we often elide headers set to $\star$ in patterns as well as priorities when they are clear from context.

Pattern semantics. The semantics of patterns is given by the function $pk\#pat$, as defined in Fig. 5. This turns out to be subtly complicated, due to the representation of packets as sequences of nested frames—a pattern contains a (possibly wildcarded) field for every header field, but not all packets contain every header field. Some fields only exist in specific frame types (**dlTyp**) or protocols (**nwProto**). For example, only IP packets (**dlTyp** = `0x800`) have IP source and destination addresses. Likewise, TCP (**nwProto** = `6`) and UDP (**nwProto** = `17`) packets have source and destination ports, but ICMP (**nwProto** = `1`) packets do not.

$$\begin{array}{lll}
\text{Wildcard} & w & ::= n \mid \star \\
\text{Pattern} & pat & ::= \{\textbf{dlSrc} = w, \textbf{dlDst} = w, \textbf{dlTyp} = w, \\
 & & \quad \textbf{nwSrc} = w, \textbf{nwDst} = w, \textbf{nwProto} = w, \\
 & & \quad \textbf{tpSrc} = w, \textbf{tpDst} = w\} \\
\text{Flow table} & FT & \in \{\!| n \times pat \times \{\!| pt |\!\} |\!\}
\end{array}$$

$$\boxed{[\![FT]\!]\ pt\ pk \rightsquigarrow \{\!| pt_1 \cdots pt_n |\!\} \times \{\!| pk_1 \cdots pk_m |\!\}}$$

$$\frac{\begin{array}{c}\exists(n, pat, \{\!| pt_1 \cdots pt_n |\!\}) \in FT \qquad pk\#pat = \textbf{true} \\ \forall(n', pat', pts') \in FT \text{ if } n' > n \text{ then } pk\#pat' = \textbf{false}\end{array}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!| (pt_1) \cdots (pt_n) |\!\}, \{\!||\!\})}$$
(FORWARD-MATCHED)

$$\frac{\forall(n, pat, pts) \in FT \qquad pk\#pat = \textbf{false}}{[\![FT]\!]\ pt\ pk \rightsquigarrow (\{\!||\!\}, \{\!| (pt, pk) |\!\})} \ \text{(TOCONTROLLER)}$$

$$\boxed{pk\#pat}$$

$$\begin{array}{l}
(\textbf{Eth}\ dlSrc\ dlDst\ dlTyp\ nwPk)\#pat = \\
\quad dlSrc \sqsubseteq pat.\textbf{dlSrc} \wedge dlDst \sqsubseteq pat.\textbf{dlDst} \wedge \\
\quad dlTyp \sqsubseteq pat.\textbf{dlTyp} \wedge \\
\quad (pat.\textbf{dlTyp} = \texttt{0x800} \Rightarrow nwPk\#_{nw}pat)
\end{array}$$

$$\boxed{nwPk\#_{nw}pat}$$

$$\begin{array}{l}
(\textbf{IP}\ nwSrc\ nwDst\ nwProto\ tpPk)\#_{nw}pat = \\
\quad nwSrc \sqsubseteq pat.\textbf{nwSrc} \wedge nwDst \sqsubseteq pat.\textbf{nwDst} \wedge \\
\quad nwProto \sqsubseteq pat.\textbf{nwProto} \wedge \\
\quad (pat.\textbf{nwProto} = \texttt{6} \Rightarrow tpPk\#_{tp}pat) \\
(\textbf{Unknown}\ payload)\#_{nw}pat = \textbf{true}
\end{array}$$

$$\boxed{tpPk\#_{tp}pat}$$

$$\begin{array}{l}
(\textbf{TCP}\ tpSrc\ tpDst\ payload)\#_{tp}pat = \\
\quad tpSrc \sqsubseteq pat.\textbf{tpSrc} \wedge tpDst \sqsubseteq pat.\textbf{tpDst} \\
\textbf{Unknown}\ payload\#_{tp}pat = \textbf{true}
\end{array}$$

$$\boxed{n \sqsubseteq w}$$

$$m \sqsubseteq n = m{=}n \qquad n \sqsubseteq \star = \textbf{true}$$

Figure 5: Flow table syntax and semantics.

To match on a given field, a pattern must specify values for all other fields it depends on. For example, to match on IP addresses, the pattern must also specify that the Ethernet frame type is IP:

$$\{\textbf{dlTyp} = \texttt{0x800}, \textbf{nwSrc} = \texttt{10.0.0.1}\}$$

If the frame type is elided, the value of the dependent header is silently ignored and the pattern is equivalent to a wildcard:

$$\{\textbf{nwSrc} = \texttt{10.0.0.1}\} \equiv \{\}$$

In effect, patterns not only match packets, but also determine how they are parsed. This behavior, which was ambiguous in early versions of the OpenFlow specification (and later fixed) has lead to real bugs in existing controllers (Section 5). Although unintuitive for programmers, this behavior is completely consistent with how packet processing is implemented in modern switch hardware.

Flow table semantics. The semantics of flow tables is given by the relation $[\![\cdot]\!]$. The relation has two cases: one for matched packets and another for unmatched packets. Each flow table entry is a tuple containing a priority n, pattern pat, and a multiset of ports $\{\!| pt_1 \cdots pt_n |\!\}$. Given a packet and its input port, the semantics forwards the packet to all ports in the multiset associated with the highest-priority matching rule in the table. Otherwise, if no matching rule exists, it diverts the packet to the controller. In the formal semantics, the first component of the result pair represents forwarded packets while the second component represents diverted packets. Note that flow table matching is non-deterministic if there are multiple matching entries with the same priority. This has serious implications for a compiler—*e.g.*, naively combining flow tables with overlapping priorities could produce incorrect results. In the NetCore compiler, we avoid this issue by always working with unambiguous and total flow tables.

$$\boxed{\mathcal{P} : sw \times pr \to [(pat, \textbf{bool})]}$$

$$\begin{array}{lcl}
\mathcal{P}(sw, \textbf{dlSrc} = n) & = & [(\{\textbf{dlSrc} = n\}, \textbf{true})] \\
\mathcal{P}(sw, \textbf{nwSrc} = n) & = & [(\{\textbf{dlTyp} = \texttt{0x800}, \textbf{nwSrc} = n\}, \textbf{true})] \\
 & \cdots & \\
\mathcal{P}(sw, \textbf{at}\ sw) & = & [(\star, \textbf{true})] \\
\mathcal{P}(sw, \textbf{at}\ sw') & = & [(\star, \textbf{false})] \quad \text{where } sw \neq sw' \\
\mathcal{P}(sw, \textbf{not}\ pr) & = & [(pat_1, \neg b_1) \cdots (pat_n, \neg b_n), (\star, \textbf{false})] \\
\multicolumn{3}{l}{\quad \text{where } [(pat_1, b_1) \cdots (pat_n, b_n)] = \mathcal{P}(sw, pr)} \\
\mathcal{P}(sw, pr\ \textbf{and}\ pr') & = & \\
\multicolumn{3}{l}{\quad [(pat_1 \cap pat'_1, b_1 \wedge b'_1) \cdots (pat_m \cap pat'_n, b_m \wedge b'_n)]} \\
\multicolumn{3}{l}{\qquad \text{where } [(pat_1, b_1) \cdots (pat_m, b_m)] = \mathcal{P}(sw, pr)} \\
\multicolumn{3}{l}{\qquad \text{where } [(pat'_1, b'_1) \cdots (pat'_n, b'_n)] = \mathcal{P}(sw, pr')}
\end{array}$$

$$\boxed{\mathcal{C} : sw \times pg \to [(pat, pt)]}$$

$$\begin{array}{l}
\mathcal{C}(sw, pr \Rightarrow pt) = [(pat_1, pt_1) \cdots (pat_n, pt_n), (\star, \{\!||\!\})] \\
\quad \text{where } [(pat_1, b_1), \cdots, (pat_n, b_n)] = \mathcal{P}(sw, pr) \\
\quad \text{where } pt_i = pt \text{ if } b_i = \textbf{true} \\
\quad \text{where } pt_i = \{\!||\!\} \text{ if } b_i = \textbf{false} \\
\mathcal{C}(sw, pg \uplus pg') = \\
\quad [(pat_1 \cap pat'_1, pt_1 \uplus pt'_1), \cdots, (pat_m \cap pat'_n, pt_m \uplus pt'_n)] +\!\!+ \\
\quad [(pat_1, pt_1) \cdots (pat_m, pt_m)] +\!\!+ \\
\quad [(pat'_1, pt'_1) \cdots (pat'_n, pt'_n)] \\
\qquad \text{where } [(pat_1, pt_1) \cdots (pat_m, pt_m)] = \mathcal{P}(sw, pg) \\
\qquad \text{where } [(pat'_1, pt'_1) \cdots (pat'_n, pt'_n)] = \mathcal{P}(sw, pg')
\end{array}$$

Figure 6: NetCore compilation.

5. Verified NetCore Compiler

With the syntax and semantics of NetCore and flow tables in place, we now present a verified compiler for NetCore. The compiler takes programs as input and generates a set of flow tables as output, one for every switch. The compilation algorithm is based on a previous algorithm [20], but we have verified its implementation in Coq. While building the compiler, we found two serious bugs in the original algorithm related to the handling of (unnatural) patterns in the compiler and flow table optimizer.

The compilation function $\mathcal{C}$, defined in Fig. 6, generates a flow table for a given switch sw. It uses the auxiliary function $\mathcal{P}$ to compile predicates. The compiler produces a list of pattern-action pairs, but priority numbers are implicit: the pair at the head has highest priority and each successive pair has lower priority.

Because NetCore programs denote total functions, packets not explicitly matched by any predicate are dropped. In contrast, flow tables divert unmatched packets to the controller. The compiler resolves this discrepancy by adding a catch-all rule that drops unmatched packets. For instance:

$$\mathcal{C}(sw, \textbf{dlSrc} = H1 \Rightarrow \{\!| 5 |\!\}) = [(\{\textbf{dlSrc} = H1\}, \{\!| 5 |\!\}), (\star, \{\!||\!\})]$$

The key operator used by the compiler constructs the cross-product of the flow tables provided as input. This operator can be used to compute intersections and unions of flow tables. Note that implementing union in the "obvious" way—by concatenating flow tables—would be wrong. The cross-product operator performs an element-wise intersection of the input flow tables and then merges their actions. To compile a union, we first use cross-product to build a flow table that represents the intersection, and then concatenate the flow tables for the sub-policies at lower priority. For example, the following NetCore program,

$$\mathbf{dlSrc} = \mathit{H1} \Rightarrow \{\!|5|\!\} \uplus \mathbf{dlDst} = \mathit{H2} \Rightarrow \{\!|10|\!\}$$

compiles to a flow table:

Priority	Pattern	Action
4	$\{\mathbf{dlSrc} = \mathit{H1}, \mathbf{dlDst} = \mathit{H2}\}$	$\{\!|5, 10|\!\}$
3	$\{\mathbf{dlSrc} = \mathit{H1}\}$	$\{\!|5|\!\}$
2	$\{\mathbf{dlDst} = \mathit{H2}\}$	$\{\!|10|\!\}$
1	$\star$	$\{\!||\!\}$

The first rule matches all packets that match both sub-programs, while the second and third rules match packets only matched by the left and the right programs respectively. The final rule drops all other packets. The compilation of other predicates uses similar manipulations on flow tables.

We have built a large library of flow table manipulation operators in Coq, along with several lemmas that state useful algebraic properties about these operators. With this library, proving the correctness theorem for the NetCore compiler is simple—only about 200 lines of code in Coq.

Theorem 1 (NetCore Compiler Soundness)**.** *For all NetCore programs pg, switches sw, ports pt, and packets pk we have* $[\![\mathcal{C}(sw, pg)]\!]\ pt\ pk = [\![pg]\!]\ sw\ pt\ pk$.

Intuitively, this theorem states that a flow table compiled from a NetCore policy for a switch sw, has the same behavior as the NetCore policy evaluated on packets at sw.

Compiler bugs. In the course of our work, we discovered that several unverified compilers from high-level network programming languages to flow tables suffer from bugs due to subtleties in the semantics of patterns. Section 4 described inter-field dependencies in patterns. For example, to match packets from IP address `10.0.0.1`, we write

$$\{\mathbf{nwSrc} = \texttt{10.0.0.1}, \mathbf{dlTyp} = \texttt{0x800}\}$$

and if we omit the **dlTyp** field, the IP address is silently ignored. This unintuitive behavior has led to bugs in the PANE [5] and Nettle [25] systems as well as an unverified version of the NetCore compiler [20]. To illustrate, consider the following NetCore policy:

$$\mathbf{nwSrc} = \texttt{10.0.0.1} \Rightarrow \{\!|5|\!\}$$

In NetCore, this policy matches all IP packets from `10.0.0.1` and forwards them out port 5. But the original NetCore compiler produced the following flow table for this program:

Priority	Pattern	Action
2	$\{\mathbf{nwSrc} = \texttt{10.0.0.1}\}$	$\{\!|5|\!\}$
1	$\star$	$\{\!||\!\}$

Because the first pattern is equivalent to the all-wildcard pattern, this flow table sends *all* traffic out port 5. Both PANE and Nettle have similar bugs. Nettle has a special case to handle patterns with IP addresses that do not specify $\mathbf{dlTyp} = \texttt{0x800}$, but it does not correctly handle patterns that specify a transport port number but not the **nwProto** field. PANE suffers from the same bug. Even worse, these invalid patterns lead to further bugs when flow tables are combined and optimized by the compiler.

Natural patterns. The verified NetCore compiler does not suffer from the bug above. In our formal development, we require that all patterns manipulated by the compiler be what we call *natural patterns*. A natural pattern has the property that if the pattern specifies the value of a field, then all of that field's dependencies must be met. This rules out patterns such as $\{\mathbf{nwSrc} = \texttt{10.0.0.1}\}$, which omits the Ethernet frame type necessary to parse the IP address. Natural patterns are easy to define using dependent types in Coq. Moreover, we can calculate the cross-product of two natural patterns by intersecting fields point-wise. Hence, it is easy to prove that natural patterns are closed under intersection.

Lemma 1. *If pat_1 and pat_2 are natural patterns, then $pat_1 \cap pat_2$ is also a natural pattern.*

Another important property is that all patterns can be expressed as some equivalent natural pattern (where patterns are equivalent if they denote the same set of packets). This property tells us that we do not lose expressiveness by restricting to natural patterns.

Lemma 2. *If pat is an arbitrary pattern, then there exists a natural pattern pat', such that $pat \equiv pat'$.*

These lemmas are used extensively in the proofs of correctness for our compiler and flow table optimizer.

Flow table optimizer. The basic NetCore compilation algorithm described so far generates flow tables that correctly implement the semantics of the input program. But many flow tables have redundant entries that could be safely removed. For example, a naive compiler might translate the program $(\star \Rightarrow \{\!|5|\!\})$ to the flow table $[(\star, \{\!|5|\!\})(\star, \{\!||\!\})]$, which is equivalent to $[(\star, \{\!|5|\!\})]$. Worse, because the compilation rule for union uses a cross-product operator to combine the flow tables computed for sub-programs, the output can be exponentially larger than the input. Without an optimizer, such a naive compiler is essentially useless—*e.g.*, we built an unoptimized implementation of the algorithm in Fig. 6 and found that it ran out of memory when compiling a policy consisting of just 9 operators!

Our compiler is parameterized on a function $\mathcal{O} : FT \to FT$, that it invokes at each recursive call during compilation. Because even simple policies can see a combinatorial explosion during compilation, this inline reduction is necessary. We require that $\mathcal{O}$ produces equivalent flow tables, *i.e.* $[\![\mathcal{O}(FT)]\!] = [\![FT]\!]$.

We have built an optimizer that eliminates low-priority entries whose patterns are fully subsumed by higher-priority rules and proved that it satisfies the above condition in Coq. Although this optimization is quite simple, it is effective in practice. In addition, earlier attempts to implement this optimization in NetCore had a bug that incorrectly identified certain rules as overlapping which we did not discover until developing this proof. The PANE optimizer also had a bug—it assumed that combining identical actions is always idempotent. Both of these bugs led to incorrect behavior.

6. Featherweight OpenFlow

The next step towards executing NetCore programs is a controller that configures the switches in the network. To prove that such a controller is correct, we need a model of the network. Unfortunately, the OpenFlow 1.0 specification, consisting of 42 pages of informal prose and C definitions, is not amenable to rigorous proof.

This section presents Featherweight OpenFlow, a detailed operational model that is designed to model the essential features of OpenFlow networks, while still fitting on a single page. The model elides a number of details such as error codes, counters, packet modification, and advanced configuration options such as the ability to enable and disable ports. But it does include all of the features relating to how packets are forwarded and how flow tables are mod-

Switch	S	$::= \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, out_m)$
Controller	C	$::= \mathbb{C}(\sigma, f_{in}, f_{out})$
Link	L	$::= \mathbb{L}((sw_{src}, pt_{src}), pks, (sw_{dst}, pt_{dst}))$
Link to Controller	M	$::= \mathbb{M}(sw, SMS, CMS)$

Devices

Ports on switch	pts	$\in \{pt\}$
Input/output buffers	in_p, out_p	$\in \{\!\mid (pt, pk) \mid\!\}$
Msgs from controller	in_m	$\in \{\!\mid SM \mid\!\}$
Msgs to controller	out_m	$\in \{\!\mid CM \mid\!\}$

Switch Components

Controller state	σ
Controller input relation	$f_{in} \in sw \times CM \times \sigma \rightsquigarrow \sigma$
Controller output relation	$f_{out} \in \sigma \rightsquigarrow sw \times SM \times \sigma$

Controller Components

From controller	SM	$::= \textbf{FlowMod}\ \delta \mid \textbf{PktOut}\ pt\ pk \mid \textbf{BarrierRequest}\ n$
To controller	CM	$::= \textbf{PktIn}\ pt\ pk \mid \textbf{BarrierReply}\ n$
Table update	δ	$::= \textbf{Add}\ n\ pat\ act \mid \textbf{Del}\ pat$

Abstract OpenFlow Protocol

Msg queue from controller	$SMS \in [SM_1 \cdots SM_n]$
Msg queue to controller	$CMS \in [CM_1 \cdots CM_n]$

Controller Link

$$\frac{[\![FT]\!](pt, pk) \rightsquigarrow (\{\!\mid pt'_1 \cdots pt'_n \mid\!\}, \{\!\mid pk'_1 \cdots pk'_m \mid\!\})}{\begin{array}{ll} & \mathbb{S}(sw, pts, FT, \{\!\mid (pt, pk) \mid\!\} \uplus in_p, out_p, in_m, out_m) \\ \xrightarrow{(sw, pt, pk)} & \mathbb{S}(sw, pts, FT, in_p, \{\!\mid (pt'_1, pk) \cdots (pt'_n, pk) \mid\!\} \uplus out_p, in_m, \{\!\mid \textbf{PktIn}\ pt\ pk'_1 \cdots \textbf{PktIn}\ pt\ pk'_m \mid\!\} \uplus out_m) \end{array}} \quad (\textsc{Fwd})$$

$$\frac{}{\begin{array}{ll} & \mathbb{S}(sw, pts, FT, in_p, \{\!\mid (pt, pk) \mid\!\} \uplus out_p, in_m, out_m) \mid \mathbb{L}((sw, pt), pks, (sw', pt')) \\ \longrightarrow & \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, out_m) \mid \mathbb{L}((sw, pt), [pk] +\!\!+ pks, (sw', pt')) \end{array}} \quad (\textsc{Wire-Send})$$

$$\frac{}{\begin{array}{ll} & \mathbb{L}((sw', pt'), pks +\!\!+ [pk], (sw, pt)) \mid \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, out_m) \\ \longrightarrow & \mathbb{L}((sw', pt'), pks, (sw, pt)) \mid \mathbb{S}(sw, pts, FT, \{\!\mid (pt, pk) \mid\!\} \uplus in_p, out_p, in_m, out_m) \end{array}} \quad (\textsc{Wire-Recv})$$

$$\frac{}{\mathbb{S}(sw, pts, FT, in_p, out_p, \{\!\mid \textbf{FlowMod Add}\ m\ pat\ act \mid\!\} \uplus in_m, out_m) \longrightarrow \mathbb{S}(sw, pts, FT \uplus \{\!\mid (m, pat, act) \mid\!\}, in_p, out_p, in_m, out_m)} \quad (\textsc{Add})$$

$$\frac{FT_{rem} = \{\!\mid (n', pat', act') \mid (n', pat', act') \in FT \text{ and } pat \neq pat' \mid\!\}}{\mathbb{S}(sw, pts, FT, in_p, out_p, \{\!\mid \textbf{FlowMod Del}\ pat \mid\!\} \uplus in_m, out_m) \longrightarrow \mathbb{S}(sw, pts, FT_{rem}, in_p, out_p, in_m, out_m)} \quad (\textsc{Del})$$

$$\frac{pt \in pts}{\mathbb{S}(sw, pts, FT, in_p, out_p, \{\!\mid \textbf{PktOut}\ pt\ pk \mid\!\} \uplus in_m, out_m) \longrightarrow \mathbb{S}(sw, pts, FT, in_p, \{\!\mid (pt, pk) \mid\!\} \uplus out_p, in_m, out_m)} \quad (\textsc{PktOut})$$

$$\frac{f_{out}(\sigma) \rightsquigarrow (sw, SM, \sigma')}{\mathbb{C}(\sigma, f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS) \longrightarrow \mathbb{C}(\sigma', f_{in}, f_{out}) \mid \mathbb{M}(sw, [SM] +\!\!+ SMS, CMS)} \quad (\textsc{Ctrl-Send})$$

$$\frac{f_{in}(sw, \sigma, CM) \rightsquigarrow \sigma'}{\mathbb{C}(\sigma, f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS +\!\!+ [CM]) \longrightarrow \mathbb{C}(\sigma', f_{in}, f_{out}) \mid \mathbb{M}(sw, SMS, CMS)} \quad (\textsc{Ctrl-Recv})$$

$$\frac{SM \neq \textbf{BarrierRequest}\ n}{\begin{array}{ll} & \mathbb{M}(sw, SMS +\!\!+ [SM], CMS) \mid \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, out_m) \\ \longrightarrow & \mathbb{M}(sw, SMS, CMS) \mid \mathbb{S}(sw, pts, FT, in_p, out_p, \{\!\mid SM \mid\!\} \uplus in_m, out_m) \end{array}} \quad (\textsc{Switch-Recv-Ctrl})$$

$$\frac{}{\begin{array}{ll} & \mathbb{M}(sw, SMS +\!\!+ [\textbf{BarrierRequest}\ n], CMS) \mid \mathbb{S}(sw, pts, FT, in_p, out_p, \{\!\mid \mid\!\}, out_m) \\ \longrightarrow & \mathbb{M}(sw, SMS, CMS) \mid \mathbb{S}(sw, pts, FT, in_p, out_p, \{\!\mid \mid\!\}, \{\!\mid \textbf{BarrierReply}\ n \mid\!\} \uplus out_m) \end{array}} \quad (\textsc{Switch-Recv-Barrier})$$

$$\frac{}{\begin{array}{ll} & \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, \{\!\mid CM \mid\!\} \uplus out_m) \mid \mathbb{M}(sw, SMS, CMS) \\ \longrightarrow & \mathbb{S}(sw, pts, FT, in_p, out_p, in_m, out_m) \mid \mathbb{M}(sw, SMS, [CM] +\!\!+ CMS) \end{array}} \quad (\textsc{Switch-Send-Ctrl})$$

$$\frac{Sys_1 \longrightarrow Sys'_1}{Sys_1 \mid Sys_2 \longrightarrow Sys'_1 \mid Sys_2} \quad (\textsc{Congruence})$$

Figure 7: Featherweight OpenFlow syntax and semantics.

ified. Many existing SDN bug-finding and property-checkings tools are based on similar (informal) models of OpenFlow [2, 10, 11]. Hence, we believe Featherweight OpenFlow could also serve as a foundation for these tools.

6.1 OpenFlow Semantics

Initially, every switch has an empty flow table that diverts all packets to the controller. Using **FlowMod** messages, the controller can insert new table entries to have the switch process packets itself. A non-trivial policy may compile to several thousand flow table entries, but **FlowMod** messages only add a single entry at a time. In general, many **FlowMod** messages will be needed to fully configure a switch. However, OpenFlow is designed to give switches a lot of latitude to enable efficient processing, often at the expense of programmability and understandability:

- **Pattern semantics.** As discussed in preceding sections, the semantics of flow tables are non-trivial: patterns have implicit dependencies and flow tables can have multiple, overlapping entries. (The OpenFlow specification itself notes that scanning the table to find overlaps is expensive.) Therefore, it is up to the controller to avoid overlaps that introduce non-determinism.
- **Packet reordering.** Switches may reorder packets arbitrarily. For example, switches often have both a "fast path" that uses custom packet-processing hardware and a "slow path" that processes packets using a slower general-purpose CPU.
- **No acknowledgments.** Switches do not acknowledge when **FlowMod** messages are processed, except when errors occur. The controller can explicitly request acknowledgements by sending a *barrier request* after a **FlowMod**. When the switch has processed the **FlowMod** (and all other messages received before the *barrier request*), it responds with a *barrier reply*.
- **Control message reordering.** Switches may process control messages, including **FlowMod** messages, in any order. This is based on the architecture of switches, where the logical flow table is implemented by multiple physical tables working in parallel—each physical table typically only matches headers for one protocol. To process a rule with a pattern such as $\{\textbf{nwSrc} = \texttt{10.0.0.1}, \textbf{dlTyp} = \texttt{0x800}\}$, which matches headers across several protocols, several physical tables may need to be reconfigured, which takes longer to process than a simple pattern such as $\{\textbf{dlDst} = \texttt{H2}\}$.

Figure 7 defines the syntax and semantics of Featherweight OpenFlow, which faithfully models all of these behaviors. The rest of this section discusses the key elements of the model in detail.

6.2 Network Elements

The model has four kinds of elements: switches, controllers, links between switches (carrying data packets), and links between switches and the controller (carrying OpenFlow messages). The semantics is specified using a small-step relation, with elements interacting by passing messages and updating their state non-deterministically.

Switches. The most interesting elements of the model are switches. A switch $\mathbb{S}$ comprises a unique identifier sw, a set of ports pts, and input and output packet buffers in_p and out_p. The buffers are multisets of packets tagged with ports, (pt, pk). In the input buffer, packets are tagged with the port on which they were received. In the output buffer, packets are tagged with the port on which they will be sent out. Since buffers are unordered, switches can process packets in any order. Switches also have a flow table, FT, which determines how the switch processes packets. As detailed in Section 4, the table is a collection of flow table entries, where each entry has a priority, pattern and, a set of output ports. Each switch also has a multiset of messages to and from the controller, out_m and in_m. There are three kinds of messages from the controller:

- **PktOut** $pt\ pk$ instructs the switch to emit packet pk on port pt.
- **FlowMod** δ instructs the switch to add or delete entries from its flow table. When δ is **Add** $n\ pat\ act$, a new entry is created, whereas **Del** pat deletes all entries that match pat exactly. In our model, we assume that flow tables on switches can be arbitrarily large. This is not the case for hardware switches, where the size of flow tables is often constrained by the amount of silicon used, and varies from switch-to-switch. It would be straightforward to modify our model to bound the size of the table on each switch.
- **BarrierRequest** n forces the switch to process all outstanding messages before replying with a **BarrierReply** n message.

Controllers. A controller $\mathbb{C}$ is defined by its local state σ, an input relation f_{in}, and an output relation f_{out}. The local state and these relations are application-specific, so Featherweight OpenFlow can be instantiated with any controller whose behavior can be modeled in this way. The f_{out} relation sends a message to a switch while f_{in} receives a message from a switch. Both relations update the state σ. There are two kinds of messages a switch can send to the controller:

- **PktIn** $pt\ pk$ indicates that packet pk was received on pt and did not match any entry in the flow table.
- **BarrierReply** n indicates that sw has processed all messages up to and including a **BarrierRequest** n sent earlier.

Data links. A data link $\mathbb{L}$ is a unidirectional queue of packets between two switch ports. To model bidirectional links we use symmetric unidirectional links. Featherweight OpenFlow does not model packet-loss in links and packet-buffers. It would be easy to extend our model so that packets are lost, for example, with some probability. Without packet loss, a packet traces paths from its source to its destinations (or loops forever). With packet loss, a packet traces a prefix of the complete path specified by our idealized model.

Control links. A control link $\mathbb{M}$ is a bidirectional link between the switch and the controller that contains a queue of controller messages for the switch and a queue of switch messages headed to the controller. Messages between the controller and the switch are sent and delivered in order, but may be processed in any order.

7. Verified Run-Time System

So far, we have developed a semantics for NetCore (Section 3), a compiler from NetCore to flow tables (Section 4), and a low-level semantics for OpenFlow (Section 6). To actually execute NetCore programs, we also need to develop a run-time system that installs rules on switches and prove it correct.

7.1 NetCore Run-Time System

There are many ways to build a controller that implements a NetCore run-time system. A trivial solution is to simply process all packets on the controller. The controller receives input packets as **PktIn** messages, evaluates them using the NetCore semantics, and emits the outputs using **PktOut** messages.

Of course, we can do much better by using the NetCore compiler to actually generate flow tables and install those rules on switches using **FlowMod** messages. For example, given the following program,

$$\textbf{dlDst} = \texttt{H1}\ \textbf{and not}(\textbf{dlTyp} = \texttt{0x800}) \Rightarrow \{\!|1|\!\}$$

Location	loc	$::= sw \times pt$
Located packet	lp	$::= loc \times pk$
Topology	T	$\in loc \rightharpoonup loc$

$$\boxed{pg, T \vdash \{\!|lp|\!\} \xRightarrow{lp} \{\!|lp|\!\}}$$

$$\frac{lps' = \{\!|(T(sw, pt_{out}), pk) \mid (pt_{out}, pk) \in [\![pg]\!]\ sw\ pt\ pk|\!\}}{\begin{array}{l} pg, T \vdash \quad \{\!|((sw, pt), pk)|\!\} \uplus \{\!|lp_1 \cdots lp_n|\!\} \xRightarrow{(sw,pt,pk)} \\ \qquad lps' \uplus \{\!|lp_1 \cdots lp_n|\!\} \end{array}}$$

Figure 8: Network semantics.

the compiler might generate the following flow table,

Priority	Pattern	Action
5	$\{$**dlDst** $=$ `H1`, **dlTyp** $=$ `0x800`$\}$	$\{\!\|\,\|\!\}$
4	$\{$**dlDst** $=$ `H1`$\}$	$\{\!\|1\|\!\}$
3	$\star$	$\{\!\|\,\|\!\}$

and the controller would emit three **FlowMod** messages:

Add 5 $\{$**dlDst** $=$ `H1`, **dlTyp** $=$ `0x800`$\}$ $\{\!|\,|\!\}$
Add 4 $\{$**dlDst** $=$ `H1`$\}$ $\{\!|1|\!\}$
Add 3 $\star$ $\{\!|\,|\!\}$

However, it would be unsafe to emit just these messages. As discussed in Section 6, switches can reorder messages to maximize throughput. This can lead to transient bugs by creating intermediate flow tables that are inconsistent with the intended policy. For example, if the **Add** 3 $\star$ $\{\!|\,|\!\}$ message is processed first, all packets will be dropped. Alternatively, if **Add** 4 $\{$**dlDst** $=$ `H1`$\}$ $\{\!|1|\!\}$ is processed first, traffic that should be dropped will be incorrectly forwarded. Of the six possible permutations, only one has the property that all intermediate states either (i) process packets according to the program, or (ii) send packets to the controller (which can evaluate them using the program). Therefore, to ensure the switch processes the messages in order, the run-time system must intersperse **BarrierRequest** messages between **FlowMod** messages.

Network semantics. The semantics of NetCore presented in Section 3 defines how a program processes a single packet at a single switch at a time. But Featherweight OpenFlow models the behavior of an entire network of inter-connected switches with multiple packets in-flight. To reconcile the difference between these two, we need a *network semantics* that models the processing of all packets in the network. In this semantics (Fig. 8), the system state is a bag of in-flight located packets $\{\!|lp|\!\}$. At each step, the system:

1. Removes a located packet $((sw, pt), pk)$, from its state,
2. Processes the packet according to the policy to produce a new set of located packets,
$$\{\!|lp_1 \cdots lp_n|\!\} = [\![pg]\!]\ sw\ pt\ pk,$$
3. Transfers these packets to input ports, using the topology, $T(lp_1) \cdots T(lp_n)$, and
4. Adds the transferred packets to the system state.

Note that this approach to constructing a network semantics is not specific to NetCore: any hop-by-hop packet processing function could be used. Below, we refer to any semantics constructed in this way as a *network semantics*.

7.2 Run-Time System Correctness

Now we are ready to prove the correctness of the NetCore run-time system. However, rather than proving this directly, we instead develop a general framework for establishing controller correctness, and obtain the result for NetCore as a special case.

Bisimulation equivalence. The inputs to our framework are: (i) the high-level, hop-by-hop function the network is intended to implement, and (ii) the controller implementation, which is required to satisfy natural safety and liveness conditions. Given these parameters, we construct a *weak bisimulation* between the network semantics of the high-level function and a network of OpenFlow switches instantiated with the controller implementation. This construction handles a number of low-level details once and for all, freeing developers to focus on essential controller correctness properties.

We prove a weak (rather than strong) bisimulation because Featherweight OpenFlow models the mechanics of packet processing in much greater detail than in the network semantics. For example, consider a NetCore program that forwards a packet pk from one switch to another, say *S1* to *S2*, in a single step. An equivalent Featherweight OpenFlow implementation would require at least three steps: (i) process pk at *S1*, moving it from the input buffer to the output buffer, (ii) move pk from *S1*'s output buffer to the link to *S2*, and (iii) move pk from the link to *S2*'s input buffer. If there were other packets on the link (which is likely!), additional steps would be needed. Moreover, pk could take an even more circuitous route if it is redirected to the controller.

Overall, the weak bisimulation states that the NetCore and FeatherWeight OpenFlow implementations are indistinguishable modulo "internal" steps. This implies that any reasoning about the trajectory of a packet at the NetCore level is preserved in FeatherWeight OpenFlow.

Observations. To define a weak bisimulation, we need a notion of observation (called actions in the π-calculus). We say that the NetCore network semantics observes a packet (sw, pt, pk) when it removes the packet from its state—i.e., just before evaluating it. Likewise, a Featherweight OpenFlow program observes a packet (sw, pt, pk) when it removes (pt, pk) from the input buffer on sw to process it using the FWD rule.

Bisimulation relation. Establishing a weak bisimulation requires exhibiting a relation $\approx_{OF}$ between the concrete and abstract states with certain properties. We relate packets located in links and buffers in Featherweight OpenFlow to packets in the abstract network semantics. We elide the full definition of the relation, but describe some of its key characteristics:

- Packets (pt, pk) in input buffers in_p on sw are related to packets $((sw, pt), pk)$ in the abstract state.
- Packets (pt, pk) in output buffers out_p on sw are related to packets located at the other side of the link connected to pt.
- Likewise, packets on a data link (or contained in **PktOut** messages) are related to packets located at the other side of the data link (or the link connected to the port in the message).

Intuitively, packets in output buffers have already been processed and observed. The network semantics moves packets to new locations in one step whereas OpenFlow requires several more steps, but we must not be able to observe these intermediate steps. Therefore, after Featherweight OpenFlow observes a concrete packet pk (in the FWD rule), subsequent copies of pk must be related to packets at the ultimate destination.

The structure of much of the relation itself is largely straightforward and dictated by the nature of Featherweight OpenFlow. However, a few parts are application specific. In particular, packets at the controller and packets sent to the controller in **PktIn** messages may relate to the state maintained in the network semantics in application-specific ways.

Abstract semantics. So far, we have focused on NetCore to build intuitions. But, our bisimulation is general, and can be constructed for any controller that implements a high-level packet-processing function. We now make this notion precise with a few additional definitions.

Definition 1 (Abstract Semantics)**.** *An abstract semantics is defined by the following components:*

1. *A packet-processing function on located packets and observations:*
$$f(lp) = \{\!|lp_1 \cdots lp_n|\!\}$$
2. *A controller implementation defined by a type of controller state, σ, and input and output relations f_{in} and f_{out}, and,*
3. *An abstraction function, $c : \sigma \to \{\!|lp|\!\}$, that identifies the set of packets the controller has received, but not yet processed.*

Note that the type of the NetCore semantics (Fig. 8) matches the type of the function above. In addition, because the NetCore controller simply holds the set of **PktIn** messages, the abstraction function is trivial. Given such an abstract semantics, we lift the packet processing function to a network semantics $\xRightarrow{lp}$ as we did for NetCore.

Abstract semantics have certain well-formedness conditions:

Definition 2 (Well-Formed Abstract Semantics)**.** *An abstract semantics is* well formed *if:*

1. *The controller ensures that all packets are either (i) processed by a switch in accordance with the packet-processing relation or (ii) sent to the controller;*
2. *Whenever the controller receives a packet,*
$$(sw, \textbf{\textit{PktIn}}\ \ pt\, pk, \sigma) \rightsquigarrow \sigma'$$
it applies the packet-processing function f to pk to get a set of located packets and adds them to its state
$$c(\sigma') = c(\sigma) \uplus f(pk)$$
3. *Whenever the controller emits a packet,*
$$\sigma \rightsquigarrow (sw, \textbf{\textit{PktOut}}\ \ pt\, pk, \sigma')$$
it removes the packet from its state:
$$c(\sigma') = c(\sigma) \setminus \{\!|(sw, pt, pk)|\!\}$$
4. *The controller eventually processses all packets (sw, pt, pk) in its state $c(\sigma)$ according to the packet-processing relation, and*
5. *The controller eventually processes all OpenFlow messages.*

The first well-formedness property is fundamental—if it does not hold, then switches can process packets in an arbitrary manner, including contrary to the intended packet-processing relation. Proving this property requires reasoning about the **FlowMod** messages sent to switches by the controller implementation. In particular, since messages can be reordered, barriers must be interspersed appropriately. The second and third properties state that the abstraction function c is compatible with the controller implementation. The fourth property requires the controller to correctly process every packet it receives. The fifth property is a liveness condition requiring the controller to process every OpenFlow message sent by switches. In the absence of failures on the control link, this property holds as long as the controller does not crash or restart.

Given such a semantics, we show that our relation between abstract and Featherweight OpenFlow states, and its inverse, are weak simulations. This implies that the relation is a weak bisimulation, and thus the two systems are weakly bisimilar.

Theorem 2 (Weak Bisimulation)**.** *For all well-formed abstract semantics, Featherweight OpenFlow states s and s', and abstract states t and t':*

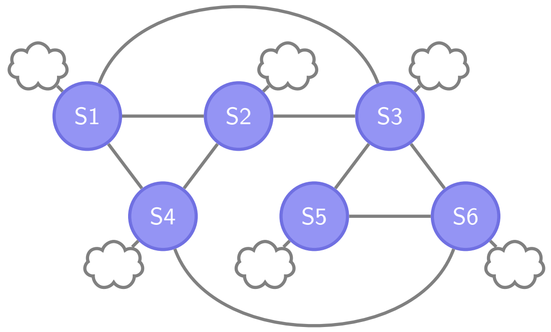


Figure 9: Waxman topology with six nodes.

- *If $s \approx_{OF} t$ and $s \xrightarrow{(sw,pt,pk)} s'$, then there exists an abstract network state t'' such that $t \xRightarrow{(sw,pt,pk)} t''$ and $s' \approx_{OF} t''$, and*
- *If $s \approx_{OF} t$ and $t \xRightarrow{(sw,pt,pk)} t'$, then there exists a Featherweight OpenFlow state s'', and abstract network states s_i, s_i' such that*
$$s \longrightarrow^* s_i \xrightarrow{(sw,pt,pk)} s_i' \longrightarrow^* s''$$
and $s'' \approx_{OF} t'$.

In this theorem, portions of the $\approx_{OF}$ relation are defined in terms of the controller abstraction function, c supplied as a parameter. In addition, the proofs themselves rely on the properties of well-formed abstract semantics (Definition 2).

Finally, we instantiate this theorem for the NetCore controller:

Corollary 1 (NetCore Run-Time Correctness)**.** *The network semantics of NetCore is weakly bisimilar to the concrete semantics of the NetCore controller in Featherweight OpenFlow.*

8. Implementation and Evaluation

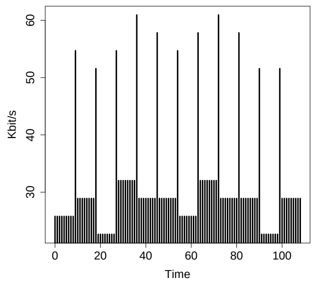
Our machine-verified NetCore controller consists of 12KLOC of Coq, which we extract to OCaml. To run the controller, we need two unverified components:

- A library to serialize OpenFlow data types to the OpenFlow wire protocol. This is a lightly modified version of the Mirage OpenFlow library [14] (1.4KLOC).
- A module to translate between the full OpenFlow protocol and the fragment used by Featherweight OpenFlow (0.2KLOC).

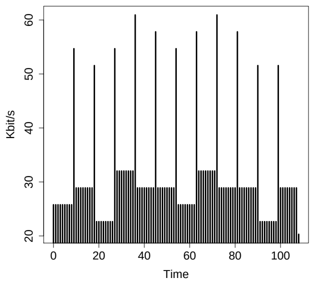
Since our model is *featherweight*, it rules out verifying controllers that use features we have elided (Section 6). However, we can use pieces of our work as verified modules in unverified contexts. For example, we use the verified NetCore compiler and flow table operations in this manner in an OCaml controller, which has rich features beyond the scope of our formal model. For the past month, we've use this partially-verified controller to manage the home network of one of the authors. At home, NetCore monitors network state and dynamically reconfigures the network as mobile devices periodically connect and disconnect.

The verified stack cannot manage dynamic networks. But, we evaluate using several microsbenchmarks described below.

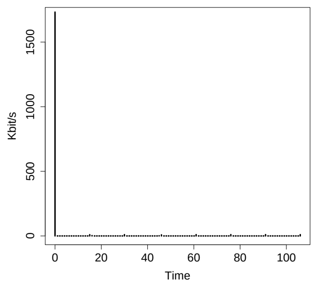
Controller throughput. A key factor that affects network performance is controller throughput as it responds to network events. CBench [24] is a benchmarking tool for measuring controller throughput. It works by flooding the controller with **PktIn** messages and measures the time taken for a **PktOut** in response. This



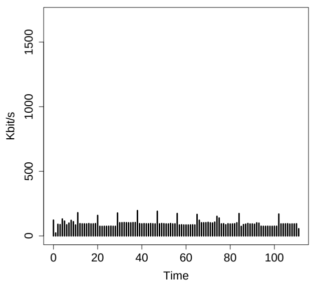
(a) Network traffic with the full controller.



(b) Network traffic with the naive controller.



(c) Control traffic with the full controller.



(d) Control traffic with the naive controller.

Figure 10: Network and control traffic from two controllers running the same NetCore program.

is a crude metric, but effective, since any correct controller must respond to **PktIn** messages.

We use CBench to compare our controller with a few others. To be fair, we program all controllers to implement the same trivial policy: "flood all packets". This helps ensure we test controller throughput, and not the performance of an application. The CBench results, from a 3rd-generation Core i3 at 3.3 GHz, are as follows:

Controller	**Messages/sec**
Unverified NetCore (Haskell)	26,022
NOX (Python and C++)	16,997
Verified NetCore (OCaml)	**9,437**
POX (Python)	6,150

The Haskell-based, unverified NetCore controller is significantly faster than the new verified controller. We attribute this to (i) a more mature implementation that uses an optimized serialization library from Nettle [25] and (ii) Haskell's support for multicore (which the controller exploits heavily). Despite being slower than the original NetCore, the new controller is still faster than the popular POX controller, though POX was designed for rapid prototyping, and not high-performance.

Control traffic. Another key factor that affects controller performance is the volume of OpenFlow traffic it exchanges with switches. Two controller that implement the same policy differently may may have wildly different traffic characteristics.

For this benchmark, we consider two strategies. Let a *full controller* have the implementation strategy presented in this paper, where it sends flow table modifications to configure switches and also processes any packets in receives per the policy. Let a *naive controller* be one that does not send **FlowMod** messages, and thus processes all packets itself. Note that we can trivially transform a full controller to a naive controller by suppressing **FlowMod** messages.

To show the effectiveness of a full controller, we contrast it with a naive controller. To do so, we developed a small suite of NetCore microbenchmarks that correspond to canonical network programs:

- **Routing:** forwards traffic between all pairs of hosts along shortest paths through the network. To test this application, we have all pairs of hosts send six ICMP echo requests ("pings") to each other and check for ICMP replies.
- **Broadcast:** floods traffic sent to a broadcast address along a spanning tree, and also forwards unicast traffic along shortest paths. To test this application, we have all hosts send 10 pings to a broadcast IP address and check for ICMP replies from all other hosts.

We ran these programs on several canonical topologies of varying sizes:

- **Fat tree:** a topology with nodes and edges organized hierarchically with more capacity higher up in the tree, commonly used in datacenters; and
- **Small world:** a topology with random links based on the structure commonly found in social networks, proposed for use in datacenters; and
- **Waxman:** a topology with random links based on the structure commonly found in enterprise networks.

Over more than 25 data points, we find that the naive controller always produces much more control traffic than the full controller. The different is particularly striking on large networks, where the naive controller has to tackle much more traffic overall, which is to be expected.

In constrast, the full controller begins with a burst of control traffic (flow modifications and barriers) that peters out as the switches is configured. We illustrate this behavior in Fig. 10. The figure shows time-series data from two experiments: the left experiment uses a full controller and the right uses a naive controller. Both controllers use the same topology (six-switch Waxman) and run the same NetCore program (broadcast composed with shortest-path routing).

We benchmark each controller with the same test (ping a broadcast address). The upper plots show the volume of ICMP traffic; the spikes occur when several hosts responds simultaneously to the broadcasted ping. As the graphs whoe, both experiments had very similar network traffic.

However, the lower plots, which measure the OpenFlow control traffic are very different. The full controller starts with big spike of traffic as it installs flow table entries, but it quickly subsides to a tiny sliver, which are OpenFlow echo messages between the controller and the switch. In contrast, the naive controller receives a continuous stream of packets. In fact, the graph even shows the controller's traffic spiking when ICMP traffic spikes, since all packets are processed by the controller.

9. Related Work

Verification technology has progressed dramatically in the past decades, making it feasible to prove useful theorems about real systems including databases [16], compilers [9], and even whole operating systems [12]. Compilers have been a particularly fruitful target for verification efforts. Most prominently, the CompCert compiler [13] translates programs in a large subset of C to PowerPC, ARM, and x86 executables. Rocksalt [21], a tool for analyzing machine code, is verified against a detailed model of x86. Another system, Bedrock [4] provides rich libraries for verifying low-level programs in Coq. Much earlier, a compiler for a Pascal-like language was formalized and verified as a part of the CLInc stack [29]. Significant portions of many other compilers have been formalized and verified, including the intermediate representation used in LLVM [31], the F* typechecker [23], and garbage collected languages [18]. Our work builds on these efforts but is the first to verify an SDN controller.

Over the past few years, a number of researchers have proposed high-level programming languages for controlling networks, including COOLAID [3], FML [8], Frenetic [6], NetCore [20], and PANE [5]. This work uses NetCore [20] as a high-level policy language. However, NetCore's original semantics were defined in terms of a complex abstract machine. Here we give NetCore a simple denotational semantics and verify its compilation into Featherweight OpenFlow. While proving our compiler and run-time system correct, we uncovered bugs in the NetCore compiler and runtime. A portion of the PANE compiler was formalized in Coq, but since their proof did not model several subtleties of flow tables, the compiler had bugs. Unlike our system, PANE does not model or verify any portion of its run-time system. We used portions of the PANE proofs during the preliminary development of our system.

Mirage [14] is a language for writing library operating systems that deploy directly to cloud platforms. It includes an OpenFlow interface so that applications can directly control SDNs. Our run-time system uses part of their OpenFlow library to serialize messages.

Formally Verifiable Networking (FVN) [26] is a platform for synthesizing protocol implementations from formal specifications (though the synthesizer is unverified). Our work attacks the problem of generating and deploying correct network-wide configurations, rather than focusing on distributed routing protocols. We employ formal methods to build compilers, shifting the need for expertise with formal methods away from programmers.

Xie et al. introduced techniques for statically analyzing the reachability properties of networks[27]. A number of tools for verifying network configurations have been built using these techniques, including Header Space Analysis [10], Anteater[15], and VeriFlow[11]. These tools check whether the installed network rules have properties specified by the programmer. Our system guarantees that the generated network rules preserve the properties of the input program, enabling higher-level verification.

NICE [2] uses model-checking and symbolic execution to find bugs in OpenFlow controllers written in Python. Portions of our Featherweight OpenFlow model are inspired by the bugs discovered in NICE. Automatic Test Packet Generation [30] analyzes network configurations and constructs packets to achieve complete configuration testing coverage. Retrospective Causal Inference[22] detects minimal input sequences to induce bugs in SDN systems.

Acknowledgements. The authors wish to thank Andrew Ferguson and the PLDI reviewers for many helpful comments, and members of the Mirage project, especially Anil Madhavapeddy, for developing the OCaml CStruct and OpenFlow libraries. This work is supported in part by the NSF under grant CNS-1111698, the ONR under award N00014-12-1-0757, and by a Google Research Award.

References

[1] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[3] X. Chen, Y. Mao, Z. M. Mao, and J. van der Merwe. Declarative configuration managaement for complex and dynamic networks. In *CoNEXT*, 2010.

[4] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, 2011.

[5] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Hierarchical policies for software defined networks. In *HotSDN*, 2012.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[7] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*, 2011.

[8] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *WREN*, 2009.

[9] T. Hoare. The verifying compiler: A grand challenge for computing research. *JACM*, 50(1):63–69, Jan 2003.

[10] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[11] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an OS kernel. In *SOSP*, 2009.

[13] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7): 107–115, Jul 2009.

[14] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *ASPLOS*, 2013.

[15] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[16] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Towards a verified relational database management system. In *POPL*, 2010.

[17] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot. Characterization of failures in an operational IP backbone network. *IEEE/ACM Transactions on Networking*, 16(4): 749–762, Aug 2008.

[18] A. McCreight, T. Chevalier, and A. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ICFP*, 2010.

[19] 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(2):69–74, 2008.

[20] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.

[21] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, faster, stronger SFI for the x86. In *PLDI*, 2012.

[22] R. C. Scott, A. Wundsam, K. Zarifis, and S. Shenker. What, Where, and When: Software Fault Localization for SDN. Technical Report UCB/EECS-2012-178, EECS Department, University of California, Berkeley, 2012.

[23] P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *POPL*, 2012.

[24] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On controller performance in software-dened networks. In *HotICE*, 2012.

[25] A. Voellmy and P. Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

[26] A. Wang, L. Jia, C. Lio, B. T. Loo, O. Sokolsky, and P. Basu. Formally verifiable networking. In *HotNets*, 2009.

[27] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[28] Z. Yin, M. Caesar, and Y. Zhou. Towards understanding bugs in open source router software. In *SIGCOMM CCR*, 2010.

[29] W. Young. Verified compilation in micro-Gypsy. In *TAV*, 1989.

[30] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *CoNEXT*, 2012.

[31] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.