Exodus: Toward Automatic Migration of Enterprise Network Configurations to SDNs

Tim Nelson Andrew D. Ferguson Da Yu Rodrigo Fonseca Shriram Krishnamurthi
Brown University
{tn, adf, dyu, rfonseca, sk}@cs.brown.edu

Abstract

We present the design and a prototype of Exodus, a system that consumes a collection of router configurations (e.g., in Cisco IOS), compiles these into a common, intermediate semantic form, and then produces corresponding SDN controller software in a high-level language. Exodus generates networks that are functionally similar to the original networks, with the advantage of having centralized programs that are verifiable and evolvable. Exodus supports a wide array of IOS features, including non-trivial kinds of packet-filtering, reflexive access-lists, NAT, VLANs, static and dynamic routing. Implementing Exodus has exposed several limitations in both today's languages for SDN programming and in OpenFlow itself. We briefly discuss these lessons learned and provide guidance for future SDN migration efforts.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network management; D.2.4 [**Software/Program Verification**]: Formal Methods; D.3 [**Programming Languages**]: Miscellaneous

General Terms

Design, Languages, Management

Keywords

Software-Defined Networking, OpenFlow, SDN Migration

1. INTRODUCTION

Managing enterprise networks is notoriously challenging [2, 6, 9, 18, 29, 30]. Software-defined networking (SDN) holds the promise of making this problem easier by centralizing configuration and management, thereby easing the evolvability of the network, and enabling the use of novel languages and verification techniques.

However, migrating from an existing, working network environment to an SDN presents a formidable hurdle [14]. Enterprises and administrators quite likely depend upon the behavior of their existing configurations. Unfortunately, these networks can be large and complex [18, 20, 29, 31], with behavior defined by myriad distributed policies, usually specified for each individual device, in a variety of configuration languages. *The scale and complexity of the aggregate behavior of these rules means the process of creating a controller program for an equivalent SDN is non-trivial.* A common deficiency of current SDN migration paths [6, 8, 20] is the need to rewrite network policies and configurations from scratch; for some, this may be reason enough not to migrate.

This paper addresses the problem of migrating existing network configurations to corresponding SDN controller programs, and presents Exodus, a system for performing this conversion. Exodus consumes configurations (Sec. 2) using a significant subset of Cisco IOS, and generates SDN controller software (Sec. 3) that mimics to a large extent the behavior of the original network. Exodus uses Flowlog [24], a high-level language for SDN programming that provides a compiler and run-time system for controlling OpenFlow switches. Atop this, Exodus generates both a single, unified controller program and a specification for the OpenFlow switches it controls—maintaining the existing topology to ease the initial transition. The resulting SDN controller can then run in the production network, or be used to evaluate the new configuration in a laboratory or emulation environment.

It is not a goal of Exodus to produce a network that is *provably equivalent* to the original, but rather one comparable in policy and functionality. As we discuss in Sec. 4, we used HSA [17] to show the equivalence of static behavior (*e.g.* ACLs), and formally verified some correctness properties of dynamic behavior. Not only is equivalence difficult to prove, it may not be productive in many cases. The centralized approach of SDNs, for example, obviates many of the complications of distributed routing protocols. NAT on OpenFlow, on the other hand, requires the controller to be involved, and will necessarily behave differently. While this paper focuses on converting IOS configurations, Exodus can handle other configuration languages as well, such as Linux iptables and Juniper JunOS configurations, limited only by parsing and translation into Flowlog.

2. OVERVIEW AND BACKGROUND

Our goal is the workflow of Fig. 1: Exodus accepts a set of IOS configurations (possibly involving multiple routers) and produces an SDN system implementing the network behavior they dictate. Exodus specifies a per-router set of OpenFlow tables that are instantiated on the network, (Sec. 3.1) and a single Flowlog controller program (Sec. 3.2) that uses these tables. This program is synthesized from standard Flowlog modules, such as ARP cache or VLAN forwarding, which are independent of the configuration, and from modules (Tab. 1) that are specialized to the network, Exodus generates the latter by feeding information from the parsed configurations to Flowlog templates. Exodus also initializes Flowlog with configuration data,

SOSR2015, June 17 – 18, 2015, Santa Clara, CA USA

DOI: http://dx.doi.org/10.1145/2774993.2774997 .

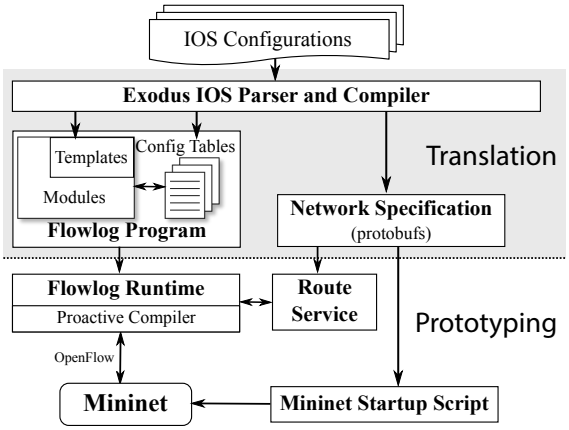

Figure 1: Workflow: Exodus produces Flowlog libraries and a network specification, which can be prototyped in Mininet [19].

such as VLANs, subnets, and OSPF weights.

To implement a centralized replacement for OSPF[1], we built an all-pairs shortest-path engine ("Route Service" in Fig. 1), implemented in under 300 lines of Python, that accepts OSPF weights from the original configuration and produces routes which it provides to Flowlog.

To enable the instantiation of an Exodus network, the compiler produces a network specification that describes the topology of OpenFlow switches with which the controller expects to interact. In our implementation we use this specification to create a Mininet [19] network with the required switches, but this could also be used as a blueprint for a physical network.

IOS provides many different features that together define desired behavior. Many IOS routers are "dual-layer": physical ports can be configured for either traditional layer-3 subnet access or as `switchports` to a VLAN. Accordingly, Exodus's output reflects not only the original static and dynamic IP routing behavior, but also VLAN encapsulation and trunking.

We present the subset of IOS that Exodus supports via the network in Fig. 2. Listing 1 shows the dual-layer configuration on "A", trimmed for brevity. This router provides layer-2 access to VLANs 2 and 3 (lines 1–6), as well as switched virtual interfaces for those VLANs (lines 7–10) so that arriving traffic can be IP routed. Lines 11–14 declare a VLAN trunk. Lines 15–20 define ordinary routing ports to two subnets along with associated OSPF costs. Finally, the router has a default static route to `192.168.2.2`. The example network (Fig. 2) contains 4 other routers; we omit their configurations for space. Routers A and B are connected via a VLAN trunk; A, C, and D form a loop to exercise dynamic routing; and ext uses stateful filtering and NAT to protect the internal network.

2.1 Scope of IOS Support in Exodus

The full IOS language exposes many additional features and a plethora of variant syntax. Rather than attempting to support all the complexities and dialects of IOS, we have focused on a core subset of IOS features. To date, Exodus supports:

1. interface declarations, either with primary subnet or layer-2 `switchport`, including switched virtual interfaces;
2. standard and most extended IOS ACLs, including reflexive access-lists;
3. ACL-based "overload" NAT;

[1]As a proof of concept, we only support a single OSPF area.

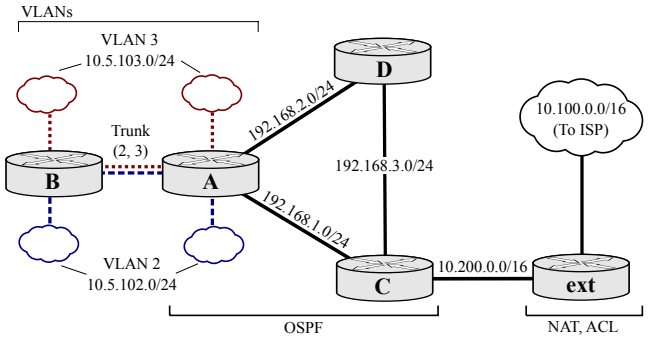

Figure 2: Example Topology. Dotted lines represent VLAN connections; solid lines represent layer-3 connections across subnets.

```
1  interface GigabitEthernet1/1                    ! VLan 2
2   switchport access vlan 2
3   switchport mode access
4  interface GigabitEthernet1/2                    ! VLan 3
5   switchport access vlan 3
6   switchport mode access
7  interface vlan 2                                ! For L3
8   ip address 10.5.102.1 255.255.255.0            ! Routing
9  interface vlan 3
10  ip address 10.5.103.1 255.255.255.0
11 interface TenGigabitEthernet1/1                 ! To B
12  switchport trunk encapsulation dot1q
13  switchport trunk allowed vlan 2,3
14  switchport mode trunk
15 interface GigabitEthernet1/3                    ! To C
16   ip address 192.168.1.1 255.255.255.0
17   ospf cost 20
18 interface GigabitEthernet1/4                    ! To D
19   ip address 192.168.2.1 255.255.255.0
20   ospf cost 5
21 ip route 0.0.0.0 0.0.0.0 192.168.2.2
```

Listing 1: Router **A** configuration

4. trunk and access `switchports` with VLANs; and
5. static and dynamic routing via OSPF (one area).

This set is non-trivial, incorporating many different commonly-used aspects of router functionality. These features are also common to multiple vendors—although configuration syntax may differ—and are not IOS specific. Supporting new vendors is thus a matter of expanding Exodus's parser, not modifying its core modules.

2.2 Flowlog Overview

Flowlog is a tierless rule-based language for SDN controller programming; programs describe network behavior and the underlying compiler automatically issues updates to switch flow-tables. We chose to use Flowlog for multiple reasons: it supports *mutable state* on the controller, allowing Exodus to translate stateful features such as NAT; it is fairly high-level, resulting in programs that are readable and maintainable after migration; it provides a rich suite of automated verification tools; and its runtime manages OpenFlow rules automatically, simplifying the resulting code.

Nevertheless, our choice of target language is not canonical: some might prefer Java or C++ generated for Floodlight or NOX to programs in a rule-based language. Flowlog does provide a stateful, proactively-compiled (i.e., OpenFlow rules are produced before packets arrive) language which serves as an excellent target for compilation; therefore, *we can view Flowlog as merely an API for its proactive compiler*. The ideas of this paper apply just as well to other compilation targets, though some engineering decisions would likely differ.

To give some intuition about Flowlog, we provide a small example: a program that implements a flood-forward switching policy while keeping track of who is sending packets to a certain subnet:

```
1 TABLE seen(ipaddr);
2 ON ip_packet(pkt):
3   DO forward(new) WHERE
4     new.locPt != pkt.locPt;
5   INSERT (pkt.nwSrc) INTO seen WHERE
6     pkt.nwDst IN 10.0.0.0/16;
```

As tables are a common representation of network data, used for routing, ARP caches, NAT state, and more, Flowlog maintains state in the form of a database. Line 1 declares a one-column table to hold the log of sender IP addresses.

The remainder of the program comprises two *rules*, both of which are triggered by any IP packet arrival (line 2) on the network. The first rule (lines 3-4) implements a basic "flood" forwarding policy; the `pkt.locPt` term represents the incoming packet's arrival port, and the `new.locPt` term the egress port. If multiple valid egress ports exist, the packet will be sent out of all of them. The second rule (lines 5-7) inserts the packet's source IP address into the table if the packet is destined for the `10.0.0.0/16` subnet.

Although the program's text makes it appear that every packet is explicitly processed by the controller, Flowlog's proactive compiler ensures that the only packets that actually reach the controller are ones that will change its internal state (i.e., packets with as-yet-unseen source addresses).

Since Flowlog's core semantics are relational, program behavior can be analyzed using relational model-finding tools such as Alloy [13]; Flowlog's tool-suite includes an automatic compiler to Alloy, enabling verification without requiring the programmer to express their program in Alloy by hand. We make use of this in our evaluation (Sec. 4).

3. DESIGN AND IMPLEMENTATION

We now describe the flow tables used, see how they map to current OpenFlow hardware, and discuss deploying and running the resulting system. We also show how Exodus maps IOS features to Flowlog.

3.1 Router Internals and Configuration

To reflect the semantics of IOS and the requirements [1] for IP routers, Exodus creates eight logical OpenFlow tables per router in the original configuration: two for VLAN-switching, two for access-control, two for routing, and one each for Layer-2 rewriting and NAT, as shown in Fig. 3(a). These tables alone cannot fully implement all features, as some require support from the controller. Rather, they implement the corresponding stage of the packet-processing pipeline as dictated by the controller.

The sequential composition of tables in Fig. 3(a) maps to OpenFlow 1.1+'s pipeline of multiple tables, and echoes the hardware pipelines of traditional routers. The VLAN table handles intra-VLAN traffic before passing packets up to the ACL for layer-3 processing. The ACL filters packets before forwarding them to the routing table, which also determines if the packet needs to be address-translated. If so, it goes through the NAT, and then through a second round of routing. The rewriting stage sets the destination MAC address. The outbound ACL performs a final access check before the VLAN table sends the packet out the appropriate port.

In OpenFlow 1.0, which we use due to its mature support, sequential composition is known to create large numbers of rules due to the necessary cross-products. To keep this in check, the Exodus prototype *physically* performs the composition by wiring single-table (OpenFlow 1.0) switches in series, one for each logical table

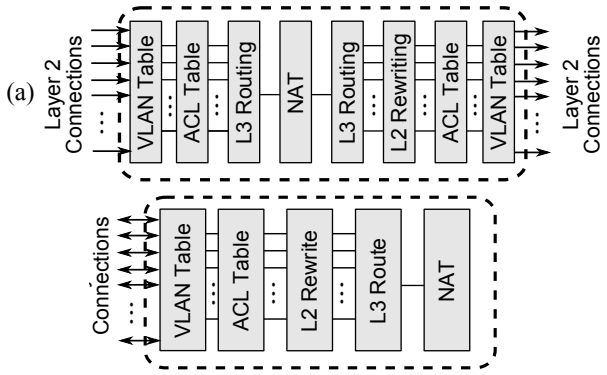

Figure 3: Logical flow tables in an Exodus router implementation (a), and via physical switches in OF 1.0 (b). Arrows denote physical ports. Internal lines show links between tables: one connection to and from the NAT table, and one per subnet otherwise.

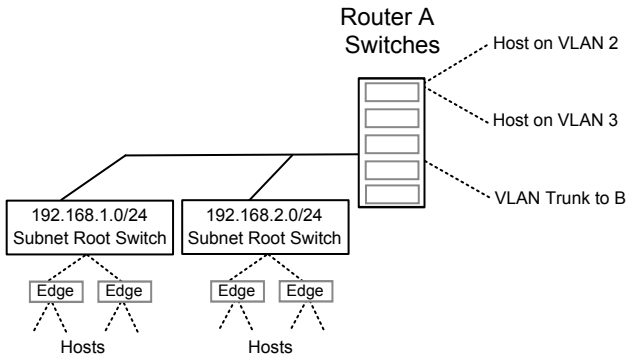

Figure 4: Exodus prototyping subnets and hosts adjacent to router A (other routers not shown). Every non-VLAN subnet is assigned a root switch, and every root has an edge switch per attached router. VLAN access ports are assigned one host each.

(Fig. 3(b)). This pipeline is designed to minimize the number of switches; we "fold" the tables symmetrically around the NAT, and packets flow in both directions. In the inbound direction, the L2 Rewrite table is just a pass-through. This design prepares Exodus for transition to newer versions of OpenFlow with support for multiple tables and makes clear the features needed by each. Of the five types of flow tables used by Exodus, four – ACL, routing, NAT, and VLAN-switching – are managed by code generated from the IOS configuration (Sec. 3.2). The fifth, Layer-2 rewriting, simply sets the destination MAC address on outgoing, routed traffic, and is managed by the ARP Cache module (see Tab. 1).

Exodus produces a description of these tables, the wiring between them, and the connectivity information for the network and outputs it in a custom Google Protocol Buffers format [25]. This serves as a blueprint for a physical network that implements the same policies as the original network. The blueprint allows us to create a running prototype of the network in an emulation environment such as Mininet [19]. We have written a Python script that loads the network into Mininet for experiments, and also creates sample subnets and hosts (Fig. 4). Forwarding within these subnets is provided by Flowlog's MAC Learning module, although any form of Layer-2 connectivity will suffice. Since trunks may connect multiple VLANs and VLAN subnets do not use a root-switch, Exodus accepts an optional list of trunk connections at runtime. Finally, the script launches SSH and web servers on each host to support testing.

3.2 Code Generation

The majority of the Flowlog code produced is unchanged across different configurations. Exodus uses a set of standard Flowlog modules, several of which it then supplements with configuration-specific rules. Tab. 1 lists the modules used in Exodus and their functionality, along with their size. It also indicates which modules are generated per-migration and which are standalone.

Some modules (e.g., L3 ACL) correspond directly to nodes in Fig. 3. Others provide supporting functionality. For instance, the Network Information Base (NIB) feeds up-to-date link-state data to the routing engine, which provides the routes used by the L3 External module. The IOS module is generated afresh for each run of Exodus. It contains configuration information extracted from the original IOS files. Because the module keeps this information in a database, rather than hard-coding it into other modules, operators can make simple changes easily while still retaining the power to make more complex changes in the code itself.

Packet-Filtering via Static ACLs ACLs can be trivially represented as forwarding rules in Flowlog. Since Flowlog has no explicit drop action, Exodus embeds higher-priority `deny` rules as negative conditions. IOS allows interfaces with separate ingress and egress filters, and so Exodus produces distinct Flowlog rule-sets for each. This does not add any notable complexity to the output, and conforms to the router pipeline in Sec. 3.1.

Static Routes Exodus maintains static-routing information in its IOS module. Each static-routing directive (e.g., line 21 of Listing 1) produces a database entry on the controller which is then added to the global routing table, using longest-prefix matching to resolve conflicts.

Stateful Filtering Stateful firewalling can be configured in IOS using *reflexive access-lists*, and requires controller interaction to implement via OpenFlow. Exodus uses a state table on the controller for the firewall state. The table contains a row for each hole to be opened in the firewall, indexed by packet-header fields. The L3 ACL module adds rows to this table as outgoing traffic is seen, and uses the table to determine what return traffic to allow.

VLANs Exodus's VLAN module provides connectivity between `switchport`s, using pre-configured values stored in the IOS module. Frames sent on a trunk will be encapsulated with the VLAN tag of their arrival access-port, and de-tagged and sent out to the appropriate VLAN upon exit. To achieve this layer-2 connectivity, the module runs a MAC-learning routine that defaults to sending along the spanning tree (provided by the NIB module) for the VLAN in question.

Dynamic Routing Exodus's external-routing module maintains a table of routes to non-directly attached destinations. The routing service updates this table as the network evolves, re-computing routes *globally* via an all-pairs shortest-path algorithm inspired by OSPF. Edge-costs come from the original configuration.

Network Address Translation Cisco IOS supports three forms of NAT: overload, static, and pooled, which correspond with N-1, N-N, and N-M translation of private to public IP addresses. Exodus implements overload NAT; the translation is similar to that for reflexive ACLs, except that it must modify packet headers. Static NAT is trivial as it requires no controller, and pool NAT would only require a new table of available public IPs.

4. EVALUATION

The feasibility of Exodus's approach depends on both its own methods, and current OpenFlow technology. We ran Exodus over the full example network in Fig. 2, launched it for prototyping in Mininet, and exercised the new program to verify its compliance.

Module	Property
VLAN	Output on a trunk is always tagged.
	Output on an access port is never tagged.
	Intra-VLAN traffic is isolated to its VLAN.
ARP	No requests are generated for cached addresses.
	Reply to requests for cached addresses.
NAT	Private addresses map to ≤ 1 public address.
	Translation is reversed at outside gateway.

Table 2: Example verified properties.

For example, we were able to successfully connect via HTTP to hosts in the `10.100.0.0/16` subnet from hosts in `10.200.1.0/24`.

We also evaluated how the number of OpenFlow rules produced scales as the system grows. To test scalability, we ran Exodus on each of 16 publicly available router configurations from the Stanford network [31], each of which has between 15 and 84 interfaces, with a combined total of 1500 ACL entries. On a 1.7 GHz Core i7 laptop, the translation to Flowlog required under a second for each configuration. Scaling factors differ across the tables. The table implementing IP routing depends mainly on the number of attached subnets and longest-prefix matches in the routing database. The VLAN table, in contrast, scales with the number of VLANs and the structure of their configuration. The size of each ACL table depends on the number of attached subnets and complexity of individual ACL configurations.

With respect to these factors, we have observed both linear and quadratic scaling. Previous runs on the Stanford configurations—before adding dynamic routing or VLANs—yielded rule counts with mostly linear scaling: the switches implementing IP routing each required only two or three more OpenFlow rules than the number of subnets (maximum 86). Layer-2 rewrite tables ranged from 55 to 325 rules, depending on the number of attached subnets. The ACL tables ranged from 31 to 581 OpenFlow rules (2840 in total across all 16 configurations).

Separating functionality into different tables lets hand-optimized tables scale linearly. However, Flowlog's automatic compiler produced a quadratic number of rules in some cases, most notably the layer-2 translator with $O((\mathit{subnets} + \mathit{hosts})^2)$ rules. This is not fundamental, and a hand-optimized layer-2 translator table only requires a rule for each attached subnet, one for each longest-prefix match in the routing table, and one for each attached host. Work is ongoing to further optimize Flowlog's compiler.

Validating Correctness.

We used a modified version of header-space analysis [17] to sanity-check examples similar to the **ext** router (Fig. 2). We confirmed that the generated ACL modules, which vary most by configuration, were translated equivalently. However, since it applies to only an instantaneous view of a network, HSA is not suitable for validating dynamic behavior.

A more appealing option is to statically prove that the compiler's output is always correct. Flowlog's logical underpinnings make it easy to put this work on formal foundations. However, while one might first equate "correctness" with "equivalence", this view is problematic. A formal proof of equivalence would require a comprehensive semantics for IOS, for which there are only partial solutions [5, 23, 32]. Moreover, equivalent behavior can even be undesirable or unattainable in an SDN. Unlike the original devices, OpenFlow switches may interact with the controller—whether to overcome their limitations or to take advantage of the controller's global knowledge. For these reasons, our validation focuses on com-

Module	# Rules	Template?	Description
NIB	27		Network Information Base. Provides up-to-date topology and spanning-tree information.
VLANs	10		Handles inter- and intra-VLAN switching.
L3 ACL	Varies	✓	Applies access-control lists from configuration. Each ACL entry becomes a Flowlog rule.
L3 Router	9	✓	Routing to directly-attached subnets, L2 translation.
L3 External	4	✓	Routing to non-attached subnets via routing-table.
ARP Cache	7		Captures, directs, and responds to ARP requests.
NAT	18	✓	Network Address Translation.
IOS	Varies	✓	Contains static configuration information such as VLAN assignments.

Table 1: Exodus modules written in Flowlog. **# Rules** gives the number of Flowlog rules in each module. A check in the **Template?** column indicates that Exodus fills in pieces for each configuration. Un-templated modules can be used as standalone Flowlog applications.

ponent correctness rather than equivalence, using Flowlog's built-in verification support. Since Flowlog programs abstract out switch-controller interaction, it is straightforward to describe properties that would otherwise involve the intricacies of the OpenFlow protocol. Tab. 2 gives a selection of properties that describe partial correctness of three distinct Exodus modules. Each property has been verified, increasing our confidence that Exodus does in fact implement correct behavior.

Extensibility.

Exodus produces code that has a relatively clear mapping from the original IOS configuration (enhanced by compiler-inserted comments). Modifying the network's global ACL only requires editing the Flowlog rules in the ACL module. The rest of the configuration is governed by static table entries loaded on startup, which can be edited even more easily than ACLs. Moreover, Flowlog's analysis tools can guide operators in their changes. Even if the SDN is eventually reimplemented, the Exodus version is valuable as an oracle for systematic testing of the new controller.

To further evaluate Exodus's extensibility, we created a novel SDN application not present in IOS. This program first blocks mDNS traffic, then implements tunnels, on-demand, for end-users who wish to stream content to registered Apple devices. The application required only seven Flowlog rules and an additional state table.

5. DISCUSSION

The output from Exodus is very clearly a hybrid: a centralized SDN controller with explicit mappings to a set of distributed switches. Exodus does not output a policy expressed over a single "big switch" abstraction [7, 22, 27]. However, armed with the combined policies translated to a high-level SDN language, we can now consider a *range* of SDN designs. Furthermore, Exodus can give insight into the resources required to migrate.

An organization may consider at least two paths for migrating to an SDN. They might leave their existing edge switches in place, and upgrade the network core to support OpenFlow. The recent Panopticon work suggests that even a single, upgraded core switch can be beneficial [20], and the Exodus prototype applies to this scenario. Alternatively, they might upgrade edge switches, in an architecture similar to that proposed by Casado, et al. [8], in which policy moves outward from the core to the edge, and here Exodus's current design can only offer general guidance about resources required.

Implementing the Exodus prototype exposed shortcomings in OpenFlow (which remain in OpenFlow 1.4), as well as in some existing SDN language abstractions:

OpenFlow: Idle Timeout for NAT OpenFlow only offers idle timeouts for single rules, which makes it difficult to correctly implement NAT. A NAT installs two rules per flow, one in each direction, and the timeout should be triggered only when *both* rules have been idle for the specified period. Extending OpenFlow 1.4's FlowMod "bundles," which support atomic transactions, to be a unit over which an Idle Timeout could be set, would solve this problem. OpenFlow switches should ideally also support Idle Timeouts triggered by TCP packets with the FIN or RST flags, as has been the case with an Open vSwitch extension since version 1.5.90.

OpenFlow: Additional ICMP Fields OpenFlow lacks support for matching on and rewriting the identifier field of ICMP queries. RFC 3022 instructs NATs to remap this field, otherwise ICMP queries (such as Echo) cannot be multiplexed [28].Without this, OpenFlow-based NATs must send ICMP traffic to the controller.

Composing Actions without Matches High-level SDN languages can simplify the composition of multiple policies (e.g., via parallel [11], sequential [22], or hierarchical merge [10]). Prior to this work, none of these approaches could compose an arbitrary header modification without first exactly matching on the field being modified. In other words, a (match, action) pair would be required for *every* observed source MAC address. However, wildcard matching is necessary for scalable flow tables. We have removed the exact-match restriction in NetCore (which Flowlog is built atop) by carefully ordering modifications, although correct re-ordering is only possible if the policy composes *at most one* such update without match in parallel. Later versions of OpenFlow provide features to access the original header values, and we believe that high-level SDN languages which offer parallel composition should support this variant.

Suspending Packet Processing We have found the need to occasionally *suspend* execution, process new packets, and later return to the buffered packet. As a concrete example, consider rewriting a packet's destination MAC address when the router does not have a corresponding entry in its ARP table. The router must first emit an appropriate ARP request, and wait for an asynchronous reply before processing the original packet. We encourage designers of high-level SDN languages to support suspending evaluation.

Stable Flow Table Output Semantically-equivalent policies in high-level languages can produce syntactically-different OpenFlow rule sets. While harmless from the packets' perspective, a canonical representation for rules would make debugging SDN applications less onerous. Ideally, automated optimization will improve this situation; we are encouraged by recent efforts [15, 16].

Flowlog Deficiencies The lack of an explicit "drop" action in Flowlog can lead to its compiler creating additional OpenFlow rules in the ACL tables (Sec. 3.2). Because Flowlog currently gives no way to extract just a single port out of potentially many expired ports, our NAT could not reuse the Layer-4 ports assigned to connections it learned had closed.

6. RELATED WORK

Migrating enterprise networks to networks with centralized control is an important topic in the SDN literature. While early proposals, such as 4D [12] or SANE [9], were understandably "clean-slate" designs, with no upgrade path other than starting from scratch, a subsequent strategy was safe co-existence. Ethane [6] required no host modifications, and allowed its switches to be incrementally deployed alongside regular switches. OpenFlow, from the start, introduced hybrid switches that could operate with Layer-2/3 control protocols or be managed by a controller, and had the requirement that OF switches would keep OpenFlow traffic isolated from production traffic [21]. Even in the case of incremental upgrades, these strategies are "dual-stack", meaning that the SDN and the traditional network are independent.

In a fully virtualized environment, one can run virtual SDN switches in the hypervisors in the edge, and provide network virtualization [8]. This approach is not feasible in many enterprise and campus networks where the edge terminates in legacy access switches. Panopticon [20] provides another migration strategy that is more integrated than a dual-stack approach. With strategic switch placement, it can almost match the benefits of a full SDN deployment for any flow that goes through at least one OpenFlow switch. It provides the illusion that the entire network is a single SDN to controller applications. In contrast to these approaches, Exodus performs a partial migration of the existing configuration and does not require the policies for the controller be written afresh.

Another approach to SDN migration is to progressively replace existing routers with functionally equivalent OpenFlow components, and then later benefit from the evolvability of such components. B4 [14] used such a strategy to replace BGP border routers in their WAN with custom OpenFlow switches. They replaced the BGP logic in the routers with a Quagga BGP node and a proxy application between the two. In doing this, they had to migrate the BGP configuration from the routers to Quagga. RouteFlow [26] allows for a similar strategy by running Quagga instances inside Linux containers and translating the routing tables from each instance into OpenFlow rules. RouteFlow thus benefits from Quagga's complete implementation of OSPF, but is tied to existing, distributed, routing algorithms. In Exodus, dynamic routing uses centralized knowledge of the network state, with all that promises for efficiency and customization.

Others (e.g., [3, 5, 17, 23, 32]) have translated IOS router configurations into intermediate logical form. However, these works are largely concerned with analysis or reduction in configuration size. The EDGE [4] tool converts configurations to a database for reporting, analysis, and automated provisioning. Exodus goes further by producing a unified SDN controller program with behavior comparable to the original.

Exodus is built atop Flowlog [24], but required significant enhancements to the original language. To create an ARP cache and proxy, we added a general hierarchy of packet types that provides access to ARP payloads. To translate ACLs and static routes, which can use address masking, we added support for matching on IP address ranges. We added an event type that allows Flowlog programs to react when OpenFlow table entries expire, with corresponding support in NetCore. We enhanced the Flowlog compiler to handle joins over multiple state relations, and added longest-prefix matching, which was previously unsupported.

7. THE ROUTE AHEAD

Exodus is a step toward SDN's overall promise of simplified management. An exciting future prospect is to refactor additional features of the network, such as VLANs or ACLs, to better express operators' high-level policies. Another path is to generate alternative network topologies, *i.e.*, provide the same functionality over a different set of switches. Targeting OpenFlow 1.1+'s chained tables, together with compiler improvements, will greatly improve the scalability and feasibility of Exodus, as well as allow support for more router features, e.g., MPLS and modifying the IP time-to-live value (currently left unchanged by Exodus routers). There is also future work to be done at the border of the Exodus network, e.g. migrating BGP configurations and producing announcement messages, possibly using the same proxy techniques as B4 [14] and RouteFlow [26]. No matter the migration strategy eventually employed, Exodus gives administrators a concrete, working prototype from which to begin discussion and compare solutions. We hope that Exodus will motivate further development of migration tools.

Acknowledgments.

We thank Sanjai Narain, Jennifer Rexford, David Walker, and the anonymous reviewers for useful feedback and discussions. Silao Xu and Charles Yeh contributed to validation efforts. Andrew Ferguson was supported by an NDSEG fellowship. This work was partially supported by the NSF.

8. REFERENCES

[1] F. Baker. Requirements for IP Version 4 Routers. RFC 1812, June 1995.

[2] T. Benson, A. Akella, and D. Maltz. Unraveling the Complexity of Network Management. In *Proc. NSDI*. 2009.

[3] T. Benson, A. Akella, and D. A. Maltz. Mining Policies from Enterprise Network Configuration. In *Proc. IMC*. 2009.

[4] D. F. Caldwell, A. Gilbert, J. Gottlieb, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford. The cutting EDGE of IP router configuration. *Proc. HotNets*, 2003.

[5] V. Capretta, B. Stepien, A. Felty, and S. Matwin. Formal Correctness of Conflict Detection for Firewalls. In *Proc. Workshop on Formal Methods in Security Engineering*. 2007.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. ACM Sigcomm*. 2007.

[7] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In *Proc. PRESTO*. 2010.

[8] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proc. HotSDN*. 2012.

[9] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proc. USENIX-SS*. 2006.

[10] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. ACM Sigcomm*. 2013.

[11] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*. 2011.

[12] A. Greenberg, et al. A clean slate 4d approach to network control and management. *ACM CCR*, 35(5):41–54, October 2005.

[13] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, April 2006. ISBN 0262101149.

[14] S. Jain, et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. ACM Sigcomm*. 2013.

[15] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *Proc. CoNEXT*. 2013.

[16] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing Tables in Software-Defined Networks. In *Proc. IEEE INFOCOM*. 2013.

[17] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. NSDI*. 2012.

[18] H. Kim, T. Benson, A. Akella, and N. Feamster. The Evolution of Network Configuration: A Tale of Two Campuses. In *Proc. IMC*. 2011.

[19] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proc. HotNets*. 2010.

[20] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann. Panopticon: Reaping the benefits of partial SDN deployment in enterprise networks. In *Proc. USENIX ATC*. 2014.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM CCR*, 38(2):69–74, March 2008.

[22] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *Proc. NSDI*. 2013.

[23] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proc. USENIX LISA*. 2010.

[24] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proc. NSDI*. 2014.

[25] Google Protocol Buffers. `https://code.google.com/p/protobuf/`.

[26] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk. Revisiting Routing Control Platforms with the Eyes and Muscles of Software-defined Networking. In *Proc. HotSDN*. 2012.

[27] S. Shenker. The future of networking and the past of protocols. Talk at Open Networking Summit, Oct. 2011.

[28] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, January 2001.

[29] Y. Sung, X. Sun, S. Rao, G. Xie, and D. Maltz. Towards Systematic Design of Enterprise Networks. *Networking, IEEE/ACM Transactions on*, 19(3):695–708, 2011.

[30] M. Yu, X. Sun, N. Feamster, S. Rao, and J. Rexford. A survey of virtual LAN usage in campus networks. *Network & Service Management Series, IEEE Communications Magazine*, July 2011.

[31] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *Proc. CoNEXT*. 2012.

[32] S. Zhang, A. Mahmoud, S. Malik, and S. Narain. Verification and Synthesis of Firewalls using SAT and QBF. *IEEE International Conference on Network Protocols (ICNP)*, 2012.