VeriExodus: Verify the Correctness of Exodus

Da Yu, Charles Yeh, Silao Xu

Final Project - CSCI 2950-u - Spring 2013

Abstract

Software-defined networking (SDN) separates the interwined control and data planes by providing abstraction and centralized-control of networks. This allows programmers to focus on high-level logical policies instead of detailed routing configurations. Exodus is a project that migrates a traditional network to an SDN such that the generated network should perform the same policies as those defined in the original network. However, the results of Exodus have not been extensively tested or verified. Since many events will make the network state keep changing, we propose a framework to prove the correctness of Exodus based on the snapshot of a given network state. In doing that, we decouple every step that an Exodus router and a Cisco router process packets and abstract them into transfer functions. After that, we merge these transfer functions to get the header space changes when packet pass through the router. By comparing the header spaces, VeriExodus can successfully verify the migrated router run the same polices as the original Cisco router in our evaluation.

1 INTRODUCTION

As conventional networks become more complicated, routers and switches need to carry and deploy many protocols or mechanisms simultaneously including routing algorithms, ACLs, VLANs, etc. To achieve this, network administrators need to write distributed policies on the routers in various router-configuration languages, a process that is very tedious and error-prone[2, 3, 6]. Software-defined networking (SDN)[3] separates the intertwined control-plane and data-plane by providing layering abstractions and centralized-control of the network. Instead of forcing users to program at the lowest network level and manually handle router-configuration languages, SDNs allows them to focus on high-level logical policies[9, 8, 4].

Exodus[1] is a project that migrates traditional networks to SDNs. It consumes router configuration files and translates them into high-level programs which can be compiled into OpenFlow[7] switches by the controller. Generated SDNs should perform the same policies as those defined in the original networks. However, the only method of verification is to send sample packets on both networks then check the fate of these packets, a method that is neither sound nor complete. We intend to test and verify that Exodus-generated networks are equivalent to the original networks in a given network state.

Since networks continue to change when events occur, such as link failures, NAT, ARP, etc., we will first verify the equivalence of the Exodus-generated network and the original network given a static network state from both. In this paper, we define two networks to be equivalent in a given network state if they implement the same policy, i.e. if all possible packets have the same fate in the two networks. To verify dynamic network state is left for future work.

Initially, we found two options for verifying equivalence between an Exodus-generated network and its original Cisco network. One approach was to search for differences in their flow tables. However, this is hard to implement because of Exodus's output pattern. Exodus generates six virtual single-table switches and wires them in series to simulate each router in the original network. So for this approach, we need to

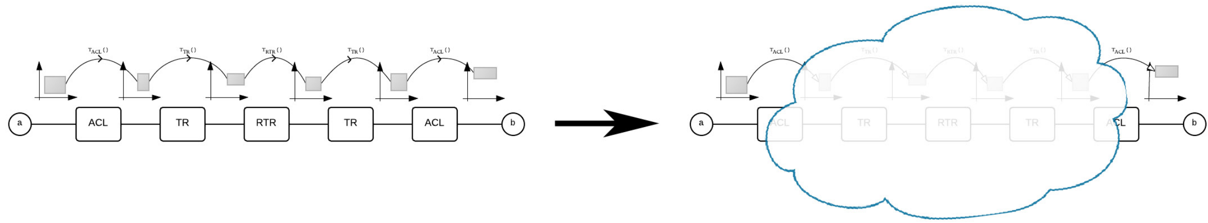

Figure 1: The OpenFlow translator translates 5 virtual switches(exclude NAT), each representing a table, into one logical transfer function

know how to combine the six virtual switches into a single flow table. We also need to know how to spot differences between two flow tables.

Another approach was to regard the networks as black boxes. Because the fate of packets is determined by the rules installed in switches and routers, we could conclude that two networks are equal by verifying that the fate of all packets is the same between the two networks. However, several questions needed to be answered: Since it is not possible to simulate all possible packets, how many packets do we need to cover every rule in the two networks? Do different packet sequences cause different results?

We use Header Space Analysis (HSA)[5] to achieve our verification goal, although the original usage of HSA is to debug the correctness of the network policies. In HSA, every network element in the network is regarded as a transfer function. Each transfer function maps a set of located packets to another set of located packets with header information modifications. We extended HSA to support L2 routing and wrote two kinds of translators:

- OpenFlow Translator: Exodus consumes configuration files as input and generates six virtual switches to achieve the same policies(Figure 2) for every router in the original network. For each virtual switch, the translator needs to parse every rule in the flow table and translate them into transfer function rules. After that, the translator merges these transfer function rules to form one cohesive transfer function as shown in Figure 1.

- Cisco Router Translator: Although the original HSA provides a Cisco router translator, we still had to rewrite the parser because: 1)the original HSA doesn't support L2 routing; 2)Exodus can only migrate a limited range of features. So for this project, we only want to verify the migration that Exodus supports within a static network model (static routing and ACLs). The translator parses the IOS configuration files and simulates the whole procedure of a router receiving packets to generate transfer functions.

Once the two translators generate transfer functions, we need to use header spaces to finish the comparison. We proposed a comparator to identify a set of header spaces in the transfer functions that are permitted through a given router. For these header spaces, the comparator will compare rules, which are five tuples, to determine whether the two transfer functions generated by the two networks are equivalent. Each rule consists of in-ports, a matching wildcard, a mask, rewrite bits, and out-ports.

We use our developed translators to generate transfer functions on our toy example. For the "ext" router(Topology as 6), the OpenFlow translator generates 28 rules from the Exodus-generated Openflow rules and the Cisco translator generates 25 rules from the original Cisco configurations. When we translate these transfer function rules into header spaces, both of the transfer functions have 11 rules which are permitted through the router. All of them are perfect matches, which means Exodus successfully migrated the original Cisco router into an SDN for the saturated network state.

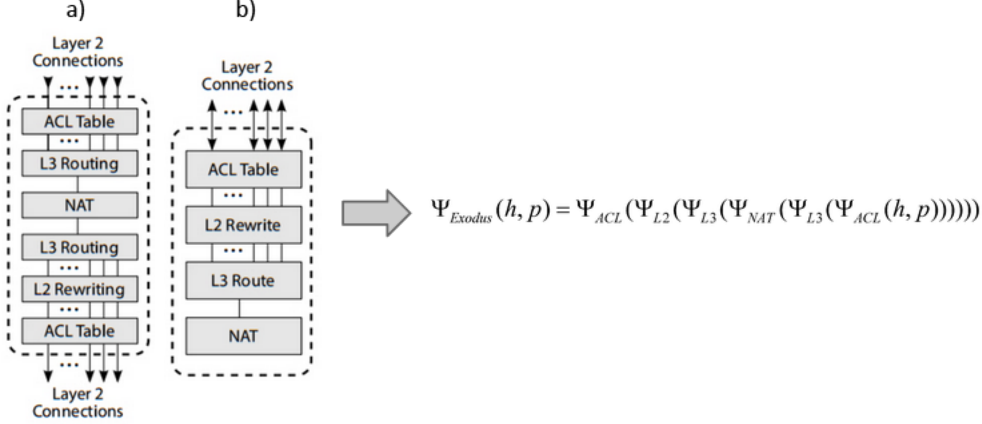

Figure 2: Exodus Router and corresponding Transfer Functions

We organize our paper as follows: In section 2, we will briefly introduce some background information about our work. Then we will introduce our design and detailed implementation of the two translators and the comparator in section 3. After describing our experiment and result analysis in section 4, we propose our future work in section 5 and conclude the study in section 6.

2 BACKGROUND

2.1 Exodus

The setup of tradition networks makes transitioning to SDNs increasingly difficult as network engineers become more entrenched within their growing networks. Exodus is a project that hopes to overcome this problem by migrating traditional networks to SDNs. It accepts router configuration files and translates them into high-level FlowLog programs. Upon execution, these programs are compiled into OpenFlow rules by network controllers and installed onto their corresponding routers and switches. Essentially, the FlowLog programs' output by Exodus have the capability to configure and maintain entire networks. These generated SDNs should perform the same policies as those defined in the original networks.

In order to gurantee the same policies after the migration, Exodus wires six logical OpenFlow tables corresponding to the router in the original configuration. Exodus actually performs the composition by wiring four single-table OpenFlow 1.0 switches in series(Left of Figure 2). These tables are folded around the NAT symmetrically and packets flow in both directions to minimize the required number of switches.

2.2 Header Space Analysis

Header Space Analysis (HSA) provides an abstract way to define networks. It regards the whole network as composition of two kinds of transfer functions. Each network element is represented as a Network Transfer Function(Ψ), which describe the network element's behaviors and packet header modifications. Topology Transfer Functions(Γ) represent connectivity among the network elements. For example, a packet with header (h) from Host A (port p) to Host B with k hops should be $\Phi^k(h,p) = \Psi(\Gamma(...(\Psi(\Gamma(h,p)...)$.

More specifically, the network Transfer Functions are defined as a 5-tuple: inport lists, header match, rewrite mask, and rewrite bits wildcard fields and outport lists. The match field describes the packet headers a rule will apply to, and the rewrite mask and bits describe the modifications to headers that are subject to

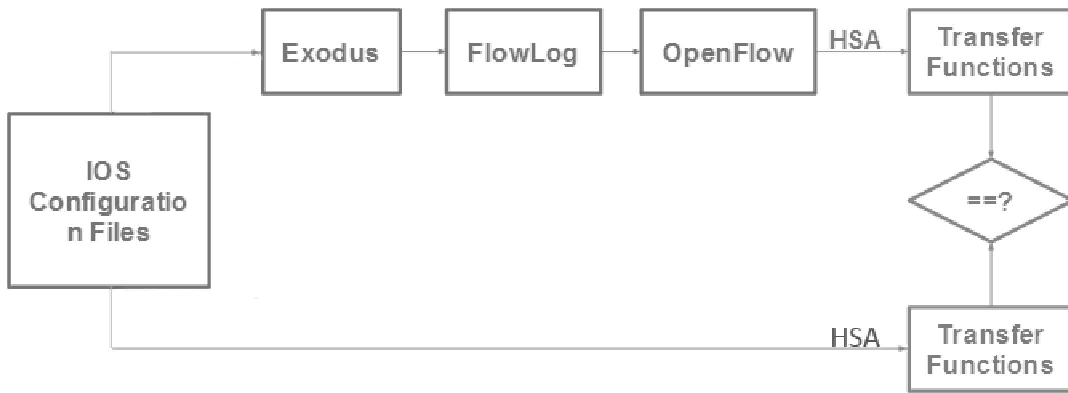

Figure 3: WorkFlow of Exodus

the rule. The inport list describes the interfaces that a rule is attached to, while the out-port list describes the ports that resulting packets will exit on. HSA provides a sound method for network comparison by converting network definitions into abstract representations.

3 IMPLEMENTATION

Converting the two network definitions into similar abstract representations allows them to be more precisely compared. The two-process route is depicted in Figure 3. Thus, we utilize HSA to create two transfer functions for each router, one from the router's Cisco IOS configuration and the other one from the Exodus-generated OpenFlow switches' implementation rules. Among these transfer function rules, the comparator will identify a set of header spaces that are permitted through a given router.By comparing these header spaces, the comparator can determine whether the two routers in different networks are equivalent. Figure 5 shows how to compare the header spaces.

3.1 OpenFlow Translator

For each OpenFlow router, Exodus creates three tables (exclude NAT) and connects them in a reflective pattern. Packet paths are folded through the tables such that a packet entering the router must pass through the tables sequentially as shown: ACL$\rightarrow$ Layer 2 Routing $\rightarrow$ Layer 3 Routings $\rightarrow$ Layer 2 Routing $\rightarrow$ ACL. Figure 2(a)(b) illustrates the OpenFlow wiring and the Exodus seperately. To emulate Exodus's wiring, we generate five transfer functions, corresponding to each table, then merge them into one that represents the entire router. The merged transfer function composition is also given as its mathematical HSA expression in Figure 2.

Initially, each OpenFlow rule can be translated directly into a transfer function rule. Each set of OpenFlow rules corresponds to a single table's transfer function. We extend HSA to support L2 routing so that OpenFlow rule fields can directly correspond to HSA rule fields, which is very straightforward. So, after the translation, the result is five sets of rules, one set for each table. These sets were created as transfer functions to facilitate merging.

We maintain priority when merging each of the first transfer function's rules with all of the second transfer function's. In other words, we used a depth-first traversal where the first transfer function's rules are the top level of a tree and each of those rules has all of the second transfer function's rules as children

	Match	Mask	Rewrite	In ports	Out ports
Rule 1	1XXX	1110	0001	[1]	[2]
Rule 2	X011	1000	0010	[2]	[3]
Merged rule	101X	1000	0010	[1]	[3]

Table 1: Merging Example

nodes. We only merge two rules if the second rule matches the output of the first one. More specifically, if the first rule's match bits rewritten with its rewrite bits are a superset of the second rule's match bits, then the two rules should be merged.

Actual merging is done separately for each field as follows.

- Match fields follow the flowchart described in Figure 4. The merge is made bit by bit. If a header space matches the merged rule, then the header space matches both rules. We must also consider the mask and rewrite bits to ensure that this is the case. If the first match field's bit is 0 or 1, then a header space must be 0 or 1 to even have the opportunity to match the second match field regardless of rewrites. If the first match field's bit is a wildcard, then whether a header space matches both fields depends on whether there is a rewrite. If there is no rewrite, then the header space must directly match the second match field. If there is a rewrite, then the rewrite forces the header space to match no matter what, so a wildcard is used.

- Mask fields are merged by simply using a logical "and" operator on the first and second rules' mask fields. Any bits rewritten by either rule will be rewritten by sequential rewrites.

- The merged rewrite field is determined by rewriting the first rule's rewrite field with the second rule's. This simulates sequential rewrites on a header space by the two rules.

- Finally, the merged rule's in-ports are the first rule's in-ports and the merged rule's out-ports are the second rule's out-ports. Because the rules are being merged, we can assume that the first rule's out-ports aligns with the second rule's in-ports. We determined this earlier when deciding whether the rules should be merged. A mapping from a rule's out-ports is mapped to the subsequent rule's in-ports makes this decision possible.

Here is a miniature example to illustrate the merging process for a four bit long header(Table 1). Within the table are two rules described as 5-tuples.

Using the flowchart described in Figure 4, we can find the resulting rule's Match field to be 101X. The last bit is a wildcard due to the first rule's rewrite. The merged mask is 1000, a simple "and" of the two rules' masks. Rewriting the second rule's rewrite field on top of the first (0001 & 1000 — 0010) gives a merged rewrite field of 0010. Finally, the merged rule's in-ports and out-ports are respectively the first rule's in-ports and the second rule's out-ports.

Merges are applied sequentially and in pairs, from the in-ACL's transfer function to the out-ACL's transfer function. In other words, the in-ACL is merged with the inwards L2 routing transfer function. The result of this merge is then merged with the L3 routing transfer function, which is subsequently merged with the outwards L2 routing transfer function, the result of which is ultimately merged with the out-ACL transfer function. Four sequential merges ensures that the transfer function stays consistent with a packet's linear traversal of the five virtual switches.

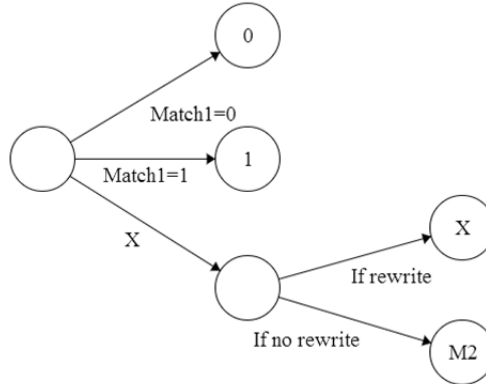

Figure 4: Merging rules for the match wildcard field

3.2 Cisco IOS Translator

In order to translate IOS configuration files, we restructured HSA's implementation. Our final result attempts to simulate Cisco IOS routers as closely as possible. Routing happens in three steps: in-ACLs, layer 3/2 routing, and out-ACLs. The translator actually requires four types of files for each network element translation:

1. The Cisco IOS configuration files that defines the ACLs and interfaces
2. A routing table that describes forwarding behavior on the network element
3. An ARP table that maps each next hop IP address to a MAC address
4. A MAC table that contains MAC address information for each network interface on the element

Actually, in our translator, each step happens in the router will be directly translated as transfer functions. Similar to the OpenFlow translator, our Cisco IOS translator will merge these three transfer functions corresponding to each step in sequence. By parsing the definition of ACLs and interfaces, the Cisco IOS translator can directly generate transfer functions for the in-ACLs and out-ACLs, including which ACLs should be applied on which interfaces. Interfaces which do not have ACLs will simply forward inbound packets. Note here, before generating transfer functions for in-ACLs, we need to confirm that packets entering an interface should have the interface's MAC address as the packet's MAC destination field, which means these kinds of packets can be received by the router.

Our Cisco IOS translater handles Layer 3 and Layer 2 routing simultaneously through a number of arp, mac, and routing table lookups. Next hops which are designated “attached” create a transfer function rule for each attached host. The steps are as follows:

1. Sort routing table entries by prefix length to simulate longest prefix matching.
2. For each entry, find the next hop IP address.
3. Retrieve the next hop's MAC address from the ARP table.
4. Create a transfer function rule which describes the routing table entry and also:
    - rewrites the packet header's MAC source address to the out-going interface's MAC address;
    - rewrites the packet header's MAC destination address to the next hop's MAC address retrieved from the ARP table;

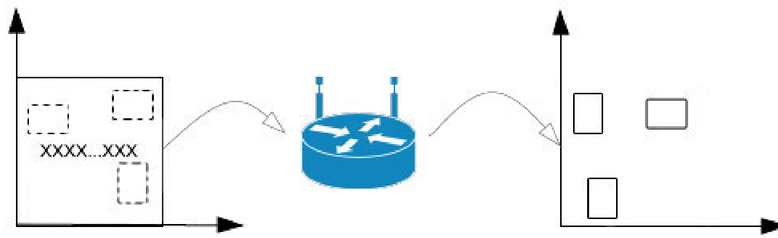

Figure 5: Header Space changes when packets pass through a router

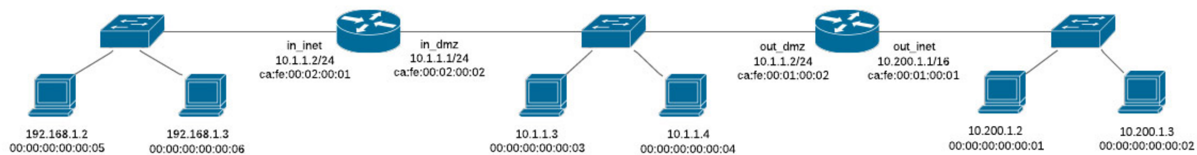

Figure 6: Topology of the toy example

The three transfer functions representing the in-ACLs, routing, and the out-ACLs are then merged. The merging process is exactly the same as the merging which took place during OpenFlow translation. The merged transfer function represents the modifications that a Cisco IOS router will execute, given the input configuration files.

3.3 Comparator

Once we got the two transfer functions that represent the forwarding behaviors of the Cisco router and the Exodus-generated router, we can verify whether the two routers perform the same policies. However, the equivalence verification is more difficult than expected. Exodus automatically simplifies redundant and optimizes policies into a more efficient set of OpenFlow rules. Besides, it doesn't do explicit drop. So, the comparator can not just directly compare each rule in pair from the two transfer functions.

So, we decide to compare the transfer function rules in pair that can pass through the Cisco router and the Exodus Router. Because all possible packets that mismatch with these transfer function rules' match wildcard will be dropped by default in both Cisco router and Exodus router. In Figure 5, we can see that only a set of header spaces that are permitted to pass through the router(dashed parts in the left side of Fig. 5) and will be transfered to solid parts in the right side of Fig. 5. By comparing the modified header spaces of the two networks, the comparator can determine whether these two networks are equivalent.

We also had to make a decision on what to do with ARP OpenFlow rules within the OpenFlow translator. Because Cisco IOS operates entirely on L3 (with the exception of L2 routing), ARP requests are nonexistent in the Cisco transfer function. Ignoring ARP requests seemed most appropriate, especially when considering our project scope limitation to static network state.

4 EVALUATION

We run our implementation using Exodus's "toy example" consisting of two routers. The routers are wired as depicted in Figure 6. Two routers named "ext" and "int" are wired together linearly. Router "ext" has two interfaces: out_dmz and out_inet. Router "int" also has two interfaces: in_dmz and in_inet. Pairs of hosts are

	TF Rules	Controller Rules	Transfer Rules	Drop Rules	Match Rules	Match Transfer Rules
OpenFlow	28	3	11	14	14	11
Cisco IOS	25	0	11	14	14	11

Table 2: The result of VeriExodus

wired at each side of and between the routers. The left side of the external facing router contains the entire 10.1.1.0/24 subnet.

We create our own meta-data files for the Cisco IOS translator including the ARP, MAC, and routing tables. For the "ext" router, the OpenFlow translator results in a transfer function with 28 rules while the Cisco translator results in a transfer function with 25 rules. Between these two transfer functions, 14 rules matched. Among these matched rules, 11 rules describe header spaces permitted through the router via various TCP ports. All of them are perfectly match in pair, which can be concluded as the two network routers are running the same policies. Details are shown in Table 2.

Since the Flowlog program runs in the controller, in this network state, when the OpenFlow switches receive some packets that they don't know how to handle, they need to send to the controller to ask for help. In the OpenFlow transfer function rules, 3 of them represent this kind of senario. The remained 14 unmatched rules are all drop rules, which are caused by two reasons: 1) when we merge transfer functions, we add the dropping rules into the final transfer function directly without any modification; 2) Exodus optimizes redundant rules and uses implicit drops compared with the Cisco router. The implicit drop is expressed by a single all-drop rule at the end of each OpenFlow translated transfer function.

5 FURTHER WORK

We believe that our comparator is very close to full functionality. Our future work is to extend our translatros and the comparator to support stateful network behaviors verification. For instance, VLANs is a Cisco IOS configuration feature that is currently unsupported by VeriExodus, although it is also unsupported by Exodus.

Successful analysis of static network state encourages exploration into comparison of dynamic network state. This is exponentially difficult due to the possible state changing paths that a network can take. Inclusion of dynamic network state would allow us to handle reflexive-acls, dynamic routing and network address translation (NAT).

6 CONCLUSION

In our work, we decouple every step about how an Exodus router and a Cisco router process packets and abstract each step into transfer functions. After that, we merge these transfer functions to know header spaces changes when packets pass through the router. In our evaluation on our toy topology, the comparator succesfully verifies the two networks perform the same policies.

Our comparator is not fully complete, but demonstrates the possibility of equivalence testing between traditional and software defined networks. Unfortunately, each type of network definition comes with its own considerations. For Cisco IOS, this is reflected in explicit drops. For Exodus's OpenFlow, this is noticeable in the simplification of rules as well as implicit drops. It would be optimal to determine a standardized set of characteristics which transfer functions should have. Normalizing these characteristics streamlines the comparison process and allows for abstract and direct comparison of transfer functions.

References

[1] Exodus: Toward automatic migration of enterprise network configurations to sdns(under review).

[2] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[3] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, August 2007.

[4] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. Frenetic: A high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[5] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[6] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. The evolution of network configuration: A tale of two campuses. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 499–514, New York, NY, USA, 2011. ACM.

[7] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[8] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[9] Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, Seattle, WA, 2014. USENIX Association.