Splendid Isolation: A Slice Abstraction for Software-Defined Networks

Stephen Gutz
Cornell

Alec Story
Cornell

Cole Schlesinger
Princeton

Nate Foster
Cornell

ABSTRACT

The correct operation of many networks depends on keeping certain kinds of traffic isolated from others, but achieving isolation in networks today is far from straightforward. To achieve isolation, programmers typically resort to low-level mechanisms such as Virtual LANs, or they interpose complicated hypervisors into the control plane. This paper presents a better alternative: an abstraction that supports programming isolated *slices* of the network. The semantics of slices ensures that the processing of packets on a slice is independent of all other slices. We define our slice abstraction precisely, develop algorithms for compiling slices, and illustrate their use on examples. In addition, we describe a prototype implementation and a tool for automatically verifying formal isolation properties.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Operations—*Network management*; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

Keywords

Isolation, software-defined networking, OpenFlow, network programming languages, Frenetic.

1. INTRODUCTION

Networks are designed to be shared—after all, having some shared infrastructure is a necessary prerequisite for communication. But the correct operation of many networks depends on keeping certain kinds of traffic isolated from others. For example, universities must restrict access to the servers that manage student records to comply with data protection laws; intelligence organizations often maintain a physical "airgap" between the devices that process packets classified at different levels of confidentiality; and datacenter operators typically ensure that traffic generated by one tenant cannot flow to the machines leased by another tenant.

In networks today, isolation is usually achieved through a variety of mechanisms, many of them ad hoc: virtual LANs (VLANs) provide a way to separate the processing of different classes of packets in the network; special-purpose devices such as firewalls prevent packets from flowing onto certain segments of the network; and systems such as Flowvisor [16] allow multiple programs to control an OpenFlow [11] network without interfering with each other. But although each of these mechanisms provides a kind of isolation, none is a completely satisfactory solution. VLANs provide traffic isolation, but add extra complexity to the already difficult task of writing network configurations. Firewalls provide physical isolation but require purchasing and deploying special devices at appropriate locations in the topology. Systems like Flowvisor provide control isolation but require interposing a hypervisor into the management plane and placing trust in a large and potentially buggy piece of software. Furthermore, these mechanisms do not provide a way to formally verify that a given network has the required isolation properties.

We believe that isolation should be provided at the language level. Instead of relying on low-level mechanisms (*e.g.*, VLANs), special-purpose devices (*e.g.*, firewalls), or complicated hypervisors (*e.g.*, Flowvisor), we argue that languages for programming networks should come equipped with intuitive and composable constructs that can be used to establish a range of isolation properties including traffic, physical, and control isolation. There are numerous advantages to treating isolation at the language level. The compiler can handle all of the tedious details related to implementing isolation, freeing programmers from having to reason about VLANs or tricky issues such as the placement of firewall boxes into the topology. It can also automatically apply optimizations that make efficient use of limited low-level resources such as VLAN tags. Unlike a hypervisor, which must intercept and analyze every event and control message at run-time, the compiler only needs to be executed once—before the program is deployed in the network—which streamlines the control plane and reduces latency. Finally, obtaining isolation through language abstractions provides opportunities for obtaining assurance using formal verification tools.

This paper presents a slice abstraction that makes it easy to isolate network programs from each other. Slices allow a single physical network to be used by multiple programs without harmful interference. They can also be used within a single program to obtain a kind of modularity—*e.g.*, ensuring that administrative traffic does not interfere with the processing of ordinary packets. Formally, a slice is defined in terms of a graph that represents a restricted version of the

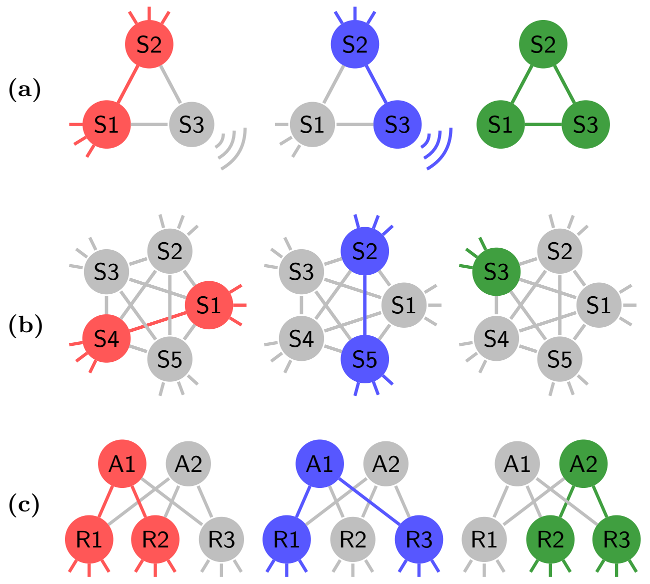

Figure 1: Example networks and associated slices: (a) campus, (b) intelligence, and (c) datacenter.

physical network topology, a mapping from the nodes in this graph to the nodes in the underlying network, and a collection of predicates that specify which packets are permitted to enter the slice at its perimeter. Programmers specify a separate program for each slice and the compiler takes the overall collection of slices, together with their associated programs, and emits a global configuration for the entire network. The specification for the compiler ensures that the slices are isolated from each other—*i.e.*, that the packets traversing each slice do not interfere with the operation of any other slice.

Overall, the contributions of this paper are as follows:

- We make the case for treating isolation at the language level, using examples inspired by common network scenarios (Section 2).
- We define a simple and elegant programming abstraction for defining slices (Section 3).
- We describe algorithms for compiling slices to OpenFlow switches and present our prototype implementation of these algorithms (Section 4).
- We discuss techniques for verifying formal isolation properties of programs expressed using slices, as well as a tool that implements these techniques using a model checker (Section 5).

We focus in this paper on isolation with respect to packet processing. We believe that our slice abstraction can be extended to handle other important issues such as bandwidth and controller resources, but we defer an investigation of these topics to future work.

2. EXAMPLES

This section introduces a series of examples that motivate the need for several different kinds of isolation: traffic isolation, physical isolation, and control isolation. We show informally how reasoning in terms of slices helps streamline the process of developing programs with these properties.

Traffic isolation. Consider the topology depicted in Figure 1(a), which represents a fragment of a university campus network. The hosts connected to switch S1 are desktop machines for trusted users such as the dean, registrar, and other administrators. The hosts connected to switch S2 are servers that store sensitive information including student records. The hosts connected wirelessly to switch S3 are machines owned by untrusted users such as students and visitors. Informally, the intended policy for the network is as follows: S1 hosts may communicate with S2 servers, but no traffic may flow between S2 servers and S3 hosts; S3 hosts may communicate with web services on S2 hosts, but not with any other services provided by those machines; and network operators may send packets to monitor the health of internal links, but those probes must not reach the hosts connected to S1, S2, and S3.

It is possible to configure the network so that it implements this policy—*e.g.*, introducing distinct VLAN tags for trusted, untrusted, and monitoring packets, and installing appropriate forwarding and filtering rules for each class of traffic on all three switches—but the details are tricky to get right, and even simple errors could easily lead to security breaches. For example, installing the wrong forwarding rule on S2 could allow an untrusted S3 host to communicate with and potentially compromise an S2 server.

Using slices, it is straightforward to write a program that correctly implements the overall policy. We simply create a slice for each class of traffic and program the slices separately. Figure 1(a) depicts the three slices. The red slice, shown on the left, handles traffic between S1 hosts and S2 servers. The blue slice, shown in the middle, handles traffic between S3 hosts and S2 web servers. The formal definition of this slice (given in the next section) restricts traffic on S2 to packets with TCP source port 80 and on S3 to packets with TCP destination port 80. The green slice, shown on the right, handles all traffic between S1, S2, and S3, but does not include the hosts connected to those switches. The program running on each slice can implement forwarding within the slice however it likes without worrying about violating the overall security policy—the semantics of the slice abstraction ensures traffic isolation. Overall, the program written using slices is significantly simpler than a corresponding program written using explicit VLANs or other low-level mechanisms.

Physical isolation. For the next example, consider a network that carries classified information in an intelligence organization. Suppose that the security policy for this organization mandates physical isolation—an “airgap”—between the devices and links that process packets classified at different levels of confidentiality. As in the campus example, we could carefully construct a policy that maintains this invariant, but doing this would require performing explicit manual reasoning about low-level switch configurations and would be very easy to get wrong.

Using slices, the situation is much simpler. We create a separate slice for each level of confidentiality and check that the required airgap exists by verifying that the sets of physical devices used to implement each slice are disjoint. Figure 1(b) depicts one possible arrangement of slices. The red slice, shown on the left, connects S1 and S4 and handles unclassified traffic. The blue slice, shown in the middle, connects S2 and S5 and handles secret traffic. The green slice, shown on the right, connects hosts on S3 and handles top-secret traffic. These slices could be programmed separately without worrying about traffic on any given segment escaping to a different segment. As each program can only reference switches and ports included in the virtual topology for the slice, it is easy for an administrator to check that the overall policy is satisfied—the semantics of slices ensures that programs cannot even reference the ports adjacent hosts at different levels, let alone direct packets to them!

Control isolation. As a final example, consider a multi-tenant cloud datacenter consisting of a collection of hosts running client virtual machines and a network organized into a "fat-tree" topology—*i.e.*, the physical hosts are connected to top-of-rack switches, which are connected to aggregation switches one level up. To allow tenants to use the network in the most efficient way possible, we would like to allow them to customize the network—*e.g.*, writing a program that implements a custom multicast protocol and efficiently moves data between their machines. But for security, we also need to ensure that a program written by one tenant does not affect traffic for other tenants.

Unlike the previous examples, we cannot construct a program with the desired isolation property—the programs are being provided by the tenants! We could use a hypervisor to monitor and check the configurations generated by each client program, but interposing a hypervisor into the network has numerous disadvantages. For one, it requires processing every control message using the hypervisor, which adds latency to one of the critical paths for performance. For another, a hypervisor is a large application (*e.g.*, the Flowvisor hypervisor is a non-trivial 16K lines of Java code), and bugs could break important invariants related to isolation.[1]

Using our slice abstraction, we can create a slice for each tenant as shown in Figure 1(c). Each slice contains the machines leased by each client as well as the top-of-rack and aggregation switches connecting them. We assume that it is possible to identify each client's traffic using fields in packet headers such as IP addresses. Note that individual machines and switches can be included in multiple slices. It is safe to let tenants program their slices because the compiler constructs a whole-network configuration that keeps the traffic generated by each tenant's machines separate. This configuration can be validated against a formal specification that captures the intended isolation policy, so even though the compiler is a large piece of software, it need not be trusted to obtain assurance. Overall, slices provide effective control isolation, even in scenarios where the network must be programmed by multiple parties.

3. THE SLICE ABSTRACTION

One reason that isolation can be difficult for programmers to reason about is that it is a global property—*e.g.*, no packets originating in one region of the network can reach devices in some other region of the network. Slices provide a means for programmers to limit the scope of a network program, restricting the devices that can be involved with the execution of the program as well as the packets that can be processed by it. The encapsulation provided by slices helps facilitate compositional reasoning about program behavior.

[1]This concern is not hypothetical: a recent bug in the Flowvisor hypervisor caused port statistics to be incorrectly sent to *all* slices [5].

Definition. Formally, a slice is defined in terms of the following ingredients:

- a *topology* that comprises switches, ports, and links,
- a *mapping* from switches, ports, and links in the slice to switches, ports, and links in the underlying network,
- and a collection of *predicates* on packets, one for each of the outward-facing edge ports in the slice.

The topology is a graph with switches as nodes, ports as annotations on nodes, and links as edges. It specifies the network elements contained in the slice. The mapping specifies how elements in the slice topology relate to corresponding elements in the physical network. The mapping is required to satisfy some straightforward conditions to ensure that it is compatible with the underlying network. For instance, every switch in a slice must map to a unique physical switch, and every pair of ports connected by a link in a slice must map to physical ports also connected by a physical link in the underlying network. We believe it should be possible to relax some of these conditions—*e.g.*, allowing many-to-one switch mappings—but our implementation does not yet support these generalizations. Finally, the predicates specify the set of packets that may enter the slice at edge ports.

Semantics. A slice extends the network with new logical switches that can be configured just like an ordinary switch. The semantics of a slice can be understood in terms of a few simple principles:

- A packet enters a slice if it arrives an external port for the slice and matches the predicate associated with that port.
- Packet processing on each slice is dictated exclusively by the program for that slice, and is not affected by the programs for any other slices.

From these principles, it is straightforward to show that slices do not interfere with each other, except possibly by sending packets from an edge port on one slice to an edge port on the other slice. Even this form of indirect interference can also be ruled out if the slices have disjoint predicates and extend all the way to the edge of the underlying network (as in the datacenter example in the preceding section). Moreover, these guarantees hold even if the slices are implemented using the same physical switches. Taken together, they serve as a strong specification for the compiler, ensuring that it translates the programs written for each slice into physical rules that implement the same forwarding behavior while providing isolation.

Note that these definitions do not preclude having multiple overlapping slices defined over the same physical switches. If a packet arrives at a port that connects to multiple slices, a copy of the packet is sent to each slice whose predicate is matched by the packet. This feature is important in scenarios like the campus network example, where traffic from the web servers must be sent to both the red and blue slices.

```
# topology
topo = nxtopo.NXTopo()
topo.add_switch(name="X",ports=[1,2,3,4])
topo.add_switch(name="Y",ports=[1,2,3,4])
topo.add_link(("X", 4),("Y", 4))
# mappings
s_map = { "X":"S2", "Y":"S3" }
p_map = identity_port_map(topo,s_map)
maps = (s_map,p_map)
# predicates
preds = \
 ([ (p, header("srcport",80))
     for p in topo.edge_ports("X") ] +
  [ (p, header("dstport",80))
     for p in topo.edge_ports("Y") ])
# slice constructor
slice = Slice(topo,phys_topo,maps,preds)
```

Figure 2: Blue campus slice.

Example. To illustrate the use of slices, consider the Python code shown in Figure 2 that implements the blue slice from the campus network example. The first few lines of code define the topology, represented as a NetworkX [9] graph with two switches: X and Y. The switches have three edge ports each and are connected by a link. The next few lines of code define the switch and port mappings from the slice down to the underlying network: switch X maps to S2, Y maps to S3, and the port mapping is the identity function. The subsequent lines associate a predicate with each edge port. These predicates map web traffic into the slice but exclude other traffic. The `nc` module used in this code provides an implementation of NetCore, a high-level language for writing predicates on packets that can be compiled to OpenFlow [12]. The final line in the program invokes the `Slice` constructor and builds the actual slice.

As an example of a program we might run on this network, consider the following NetCore program, which implements a simple broadcast protocol connecting the hosts on the blue slice:

```
  (inport("X",[1,2,3])
      |then| forward(4))
+ (inport("X",4)
      |then| forward([1,2,3]))
+ (inport("Y",[1,2,3])
      |then| forward(4))
+ (inport("Y",4)
      |then| forward([1,2,3]))
```

We will use this example in the next section to explain the details of the compilation algorithm.

4. THE SLICE COMPILER

We have implemented a prototype compiler for slices. The code for our compiler is available at the following URL:

`https://github.com/frenetic-lang/slices`

The compiler takes as input a collection of slice definitions and their associated programs, expressed in NetCore, and emits as output a list of OpenFlow forwarding rules for each switch in the physical network.

In general, a slice compiler has significant flexibility in how it implements isolation. The only requirement is that it must correctly implement the semantics of slices described in the preceding section. In cases where the slices are defined over disjoint sets of switches, the compiler can often simply rewrite each program using the switch and port mappings in the corresponding slice definition. But in general, the compiler must instrument the programs to ensure isolation.

Our compiler uses a simple strategy for instrumenting programs using VLAN tags. It works by creating a program for each slice that is—by construction—isolated from every other slice. More formally, compilation proceeds as follows. For each slice s, the compiler applies the following transformations to p, the NetCore program associated with s:

- Allocate a fresh VLAN tag v.
- Create a program p_v by restricting p so that it only applies to packets whose VLAN field is v.
- Create a program p_{in} by restricting p to only apply to packets at edge ports that match the associated predicate in s and have no VLAN tag, and add an action that pushes the VLAN tag v onto every such packet.
- Create a program p_{out} by restricting p to only apply to packets with VLAN tag v being forwarded out edge ports, and add an action to remove the VLAN tag from every such packet.
- Create a program p_{hop} by restricting p to only apply to packets at edge ports that match the associated predicate in s with no VLAN tag where p yields a forwarding action that immediately forwards the packet out an edge port.
- Apply the switch and port mappings to the program formed by taking the union of p_v, p_{in}, p_{out}, and p_{hop}.

Intuitively, the program p_v handles packets traversing the interior of the slice, programs p_{in} and p_{out} handle packets entering and exiting the slice respectively, and p_{hop} handles packets that enter and exit the slice in a single hop. Each of these programs are straightforward to construct in NetCore, because (unlike raw OpenFlow forwarding rules) the language supports powerful set-theoretic operators such as union, intersection, negation, etc. To generate the final result, the compiler forms the union of the instrumented programs for each slice, and uses the NetCore compiler to convert the result into a list of OpenFlow rules.

Example. As an example to illustrate the compilation algorithm, consider the broadcast program described in the previous section and the blue campus slice. Because there are no non-trivial internal paths, the interior program p_v is empty. Likewise, because there are no non-trivial one-hop paths, the program p_{hop} is also empty. The only interesting programs are the input program p_{in},

```
((inport("S2",[1,2,3]) &
  header("srcport",80) &
  header("vlan",0))
    |then| action([4], {"vlan":1}))
| ...
```

and the output program p_{out}:

```
((inport("S2",4) & header("vlan",1)
    |then| action([1,2,3], {"vlan":0}))
| ...
```

The input program for S2 takes all packets that match the slice predicate with VLAN tag 0, re-tags them with VLAN 1, and forwards them out port 4. The output program for S2 takes incoming packets from S3 with VLAN tag 1, re-tags them with VLAN 0, and forwards them out its edge ports.

It is worth noting that the use of VLANs by our compiler is not essential; the reference implementation sketched here is not the only possibility. We are currently working to develop other compilation algorithms for slices that make more prudent use of low-level switch resources.

5. VERIFICATION

One of the main advantages of developing abstractions at the language level is that having a program to analyze provides a means for establishing correctness using language-based tools. We have built a verification tool that checks isolation properties of slices expressed using logic formulas.

Our tool implements an approach to compiler correctness known as *translation validation* [13]. Instead of verifying the correctness of the compiler directly—something that would be quite difficult to do, as it would entail developing complete a formal specification of the intended behavior of the compiler—we analyze its output and check that it has the required properties. This approach removes the compiler from the trusted computing base and replaces it with a widely-used and well-tested verification tool. Hence, bugs in the compiler do not invalidate properties established using our tool. Of course, the tool itself may have errors, but it is extremely unlikely that a single bug would manifest in two pieces of software developed independently.

Intuitively, isolation involves restricting the paths that packets may take through the network. For example, traffic isolation restricts the set of destinations that a packet may reach and physical isolation restricts the switches and links it may traverse. Temporal logics are a natural tool for expressing such properties, since their formulas describe the paths a system can take as it evolves over time. Our tool supports properties expressed in Computation Tree Logic (CTL) [3] and uses the NuSMV model checker [2] to verify formulas against models extracted from OpenFlow configurations.

Before presenting the specific isolation properties we verified with our system, let us briefly review the basic syntax of CTL. Properties of individual packets can be expressed using equality constraints and boolean connectives, as in `dstip = 10.0.0.1 & switch = S`. This formula is satisfied by all packets located at switch S whose destination IP address is `10.0.0.1`. CTL's temporal operators provide a way to express properties of packets as they flow through the network. The formula AF ϕ states that on all paths, the formula ϕ holds at some point in the future. For example, `AF(switch = S)` states that packets can always reach switch S from the current location. Similarly, AG ϕ states that on all paths from the current position, ϕ must hold globally. Using logical implication, we can produce more interesting formulas, such as `(switch = S1) -> AF(switch = S2)`, which states that all packets at switch S1 must eventually be forwarded to S2.

Traffic isolation. To provide traffic isolation, a slice must ensure that every packet that arrives at one its edge ports (and matches the predicate associated with that port) only ever traverses switches, ports, and links belonging to the same slice. Consider the campus example from Section 2, and suppose that ports P1 and P2 represent the ports on S1 and S2 respectively. The following formula is satisfied only if the red slice enforces traffic isolation:

```
   (loc = P1 | loc = P2) ->
 AF(loc = P1 | loc = P2 | loc = DROP)
```

Intuitively, this formula states that any packet arriving at one of the edge ports of the slice must eventually reach one of the edge ports in the same slice or be dropped. The variable `loc` mentioned in this formula refers to the location of the packet in the network on a switch or a host and `DROP` is a special location for dropped packets.

Physical isolation. In some networks, it is important to ensure that all switches and links are only ever used to process packets for at most one slice. Consider the blue slice from the intelligence organization example in Section 2. The following formula is satisfied only if the blue slice enforces physical isolation:

```
   (switch = S5 & port = 1) ->
 AG(switch = S2 | switch = S5 | loc = DROP)
```

Intuitively, it says that any packet entering the slice must only traverse switches S2 and S5 or be dropped.

Deploying verification. Language-based verification has a rich history in traditional software systems, where analysis tools are often applied at compile time to detect bugs early in the development cycle. We believe that applying the same techniques could help increase the reliability of network programs too. Although verification tools such as model checkers can take a long time to complete, network configurations are often written well in advance, so programmers could compile and verify their programs before they are deployed. Verifying the behavior of a hypervisor, on the other hand, could only take place at run-time, and global invariants would need to be re-verified on every network event and control message.

6. RELATED WORK

Networks today typically achieve isolation using low-level mechanisms such as VLANs or firewalls. These mechanisms can be used to effectively provide both traffic and physical isolation, but their use requires careful configuration by expert operators and is prone to errors.

Flowvisor is the most prominent example of a system that provides isolation in OpenFlow networks. It allows multiple controllers to manage a single network [16]. Architecturally, Flowvisor is organized as a hypervisor that sits between the controllers and switches, filtering the events going up to controllers and masking the messages going down to switches. Flowvisor allows an administrator to identify "slices" of the flowspace using topology and packet header characteristics similar to our slices. In addition, Flowvisor also supports bandwidth, switch control plane, and controller isolation for slices, using heuristics to estimate the amount of processing power needed to implement by each slice. Another example of a system that provides isolation using a hypervisor-based approach is XNetMon [4]. The frameworks for virtualizing OpenFlow networks described by Casado et al. [1], Zarifis and Kontesidou [8], and Reich et al. [14] can also be used to provide various forms of isolation.

There is growing interest in applying verification techniques to networks. The verification tool described in this paper builds on one originally developed in the context of work on network updates [15]. Header space analysis [7] provides a formal model of OpenFlow networks as a "transfer function" as well as a tool for checking properties of networks including connectivity, access control, and loop freedom. Anteater [10] verifies network invariants by translating them into SAT instances and using an external solver.

7. FUTURE WORK

Our work on slices is ongoing. We are currently developing a formal semantics for slices, and proving properties such as non-interference with respect to confidentiality (packets do not leak outside of a slice) and integrity (packets generated by other slices do not affect the operation of a slice). We believe that these intuitive and robust guarantees will be a powerful reasoning tool for programmers.

We are also developing additional optimized compilation algorithms that exploit information about slices, programs, and the topology to provide isolation while minimizing the use of VLAN tags. For example, if a single link is only ever used by one slice, then packets traversing that link do not need to be tagged at all.

In the future, we plan to extend our slicing abstraction to handle resources such as bandwidth and controller resources. We also plan to relax the restrictions on switch and port mappings to allow more flexible slice definitions. Finally, we plan to develop a convenient surface syntax for describing slices and integrate them into the Frenetic language [6, 12].

Acknowledgments. We wish to thank Shrutarshi Basu, Josh Reich, Mark Reitblatt, Jennifer Rexford, and David Walker, and the anonymous reviewers for many helpful comments and suggestions. Our work is supported in part by the ONR under grant N00014-09-1-0652 and by the NSF under grants CNS-1111698, CCF-0424422, and SHF-1016937.

8. REFERENCES

[1] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO), Philadelphia, PA*, 2010.

[2] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification (CAV), Copenhagen, Denmark*, pages 359–364, July 2002.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[4] N.C. Fernandes and O.C.M.B. Duarte. XNetMon: A network monitor for securing virtual networks. In *International Conference on Communications (ICC), Kyoto Japan*, pages 1–5, June 2011.

[5] FlowVisor. Bug report, March 2012. See `https://openflow.stanford.edu/bugs/browse/FLOWVISOR-171`.

[6] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 279–291, September 2011.

[7] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Jose, CA*, April 2012.

[8] Georgia Kontesidou and Kyriakos Zarifis. OpenFlow virtual networking: A flow-based network virtualization. Master's thesis, KTH Royal Institute of Technology, 2009.

[9] Los Alamos National Laboratory. NetworkX, November 2011. Available from `http://networkx.lanl.gov`.

[10] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), Toronto, Canada*, pages 290–301, August 2011.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communications Review (CCR)*, 38(2):69–74, 2008.

[12] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, PA*, pages 217–230, January 2012.

[13] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), Lisbon, Portugal*, pages 151–166, March 1998.

[14] Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Toward a language for network virtualization. Draft, April 2012.

[15] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), Helsinki, Finland*, August 2012. To appear.

[16] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, Srinivasan Seetharaman, David Underhill, Tatsuya Yabe, Kok-Kiong Yap, Yiannis Yiakoumis, Hongyi Zeng, Guido Appenzeller, Ramesh Johari, Nick McKeown, and Guru Parulkar. Carving research slices out of your production networks with openflow. *ACM SIGCOMM Computer Communications Review (CCR)*, 40(1):129–130, January 2010.