OpenStack networking for humans: symbolic execution to the rescue

Radu Stoenescu, Dragos Dumitrescu, Costin Raiciu
University Politehnica of Bucharest
Email: firstname.lastname@cs.pub.ro

***Abstract*—Neutron is the OpenStack component that implements networking and it has been mocked and derided the weakest link in OpenStack [11]. We propose to use network symbolic execution to improve Neutron's ability to correctly implement tenant policies and to provide tenant traffic isolation.**

We propose to apply symbolic execution on two different OpenStack layers: the tenant view of the network and the actual deployment. Analyzing the tenant view is useful in many ways; first, it helps the tenant better understand its configuration's behavior before deployment. Secondly, its outputs can be compared to the analysis of the deployment to check if they are equivalent. We have built a prototype implementation and conducted preliminary evaluation, finding that we can verify our department's OpenStack deployment in seconds and detect certain common Neutron problems.

I. Introduction

The cloud is taking over the world of computing. Public clouds such as Amazon EC2, Microsoft Azure or Rackspace are widely used, and smaller clouds are being built pretty much everywhere. Network operators are building miniature clouds in their core networks (e.g. DT is deploying racks collocated with PoPs) while mobile operators are deploying processing close to the edge to enable mobile edge computing [7]. Deploying a cloud is no easy task. Major public clouds providers have each developed their own custom cloud management software, but the software is deployment-specific and not available publicly. New cloud players are very numerous and eyeing smaller deployments; having each of them develop cloud software makes no sense.

OpenStack is the leading community effort to build a production-quality, open source platform that enables building public and private clouds with ease. OpenStack has a lot of momentum, with major companies investing human and capital resources, and is reaching maturity. Hundreds of OpenStack clouds have already been deployed [2]. We focus on Neutron, the networking component of OpenStack, that allows users to specify their high-level networking configuration and deploys it. Neutron is notoriously unreliable, to the point where it has become known as the weakest link in OpenStack and bashed in popular media by company executives [11]. Neutron has certainly improved recently, but it is still far from perfect.

In this position paper we propose to use network static analysis, in particular symbolic execution [13], to improve Neutron. Rather than provide a definitive solution, we provide a high level approach to solving Neutron's woes. Our key idea is to use symbolic execution to analyze the properties of a) the tenant virtual network configuration before deployment and b) the actual network dataplane after deployment.

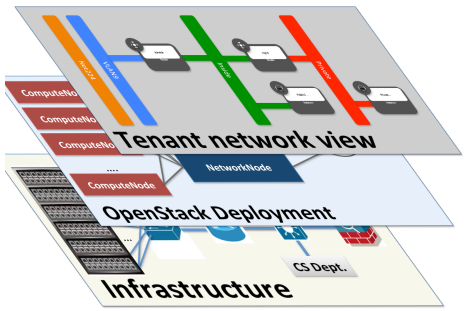

Fig. 1. OpenStack Networking Layers.

We have built a prototype to showcase our approach and check its validity, and performed a preliminary evaluation. Our initial results are promising: verification is fast (seconds) and can detect common problems with Neutron deployments.

II. OpenStack Networking with Neutron

Network virtualization in OpenStack is enabled by Neutron [8]. It offers an API to tenants allowing the creation of virtual networks that are decoupled from the underlying networking topology and the configuration chosen for deployment. The tenant network is then instantiated on the physical topology, and this mapping is influenced by the way the cloud provider has deployed OpenStack. Neutron layering is captured in Figure 1 and it aims to achieve the following goals:

- **Policy Compliance.** Neutron aims to allow tenants to configure networks that satisfy their high-level security policies, for instance separation of public and back-end traffic, reachability, etc.
- **Implementation Correctness.** The properties of the virtual network configured by the tenant should be matched by the actual deployment. For instance, Internet packets that can reach an instance in the virtual network should also reach the instance in the instantiated network.
- **Traffic Isolation.** A tenant's traffic should not reach other tenants unless Neutron is explicitly configured to do so.
- **Performance.** Neutron must allow cloud providers and tenants to effectively utilize fast interconnects including 40Gbps and 100Gbps Ethernet.

Achieving all these properties simultaneously is very tricky. Achieving tenant policy compliance appears simple: the tenant only has to correctly configure its virtual network (given

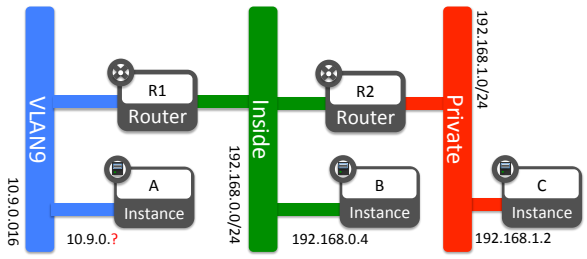

Fig. 2. Example of a tenant network topology in the Horizon GUI.

the appropriate configurationAPI), however tenants are not networking professionals usually, so many configuration errors are possible. Furthermore, the short and inexpensive configure-deploy cycle facilitates the introduction of configuration bugs.

Implementation correctness and traffic isolation are achieved through coding best practices in the open-source community; however this implies that only very popular approaches will be heavily scrutinized, and many bugs will exist in less popular code. Achieving correctness and isolation in the context of proprietary drivers for third-party networking hardware is even more challenging. Even with code review, there is no guarantee that these properties are met in practice. Finally, achieving high networking performance implies using pass-through technologies such as SR-IOV for virtual machines, and relying on networking hardware to implement Neutron. SR-IOV traffic bypasses the local hypervisor stack (e.g. iptables and Openvswitch) and only basic security is provided, meaning that other tenant-specified functionality (such as firewalling) may not be provided. We now provide a more detailed view of OpenStack networking and highlight the difficulties when trying to meet the OpenStack goals.

Tenant network view. Tenants use an virtual network view to configure the way their instances are connected. To this end they can use layer 2 networks (flat or VLANs), routers, firewalls, VPNs and load balancers. In more detail, tenants can use the Horizon GUI or the Neutron API as follows:

- The simplest choice is to use a flat network where virtual machine instances are assigned IP addresses from a DHCP server run by the cloud provider. The DHCP server also provides a gateway for outgoing connections and performs network address translation. Incoming connections are dropped by default.
- Create VLAN(s) and associated subnet(s) where the connected instances are assigned, at configuration time, distinct IP addresses from the subnet's range. One instance can be connected (have interfaces in) multiple VLANs.
- Create routers that can interconnect different VLANs and provide Internet connectivity. All addresses assigned (either with subnets or DHCP) are private and thus not reachable from the Internet. Outgoing Internet connectivity can be provided by using NAT.
- Assign floating IPs if incoming connectivity is desired (e.g. for the tenant to be able to ssh into its instance). A floating IPs is a public IP addresses that is associated to a private IP address of the tenant. Neutron perform address translation between the floating address and the private one, transparent to the VM.
- Specify firewall rules to be applied to specific ports.
- Further constructs include VPNs and traffic load balancing, and this list is likely to increase in the future.

In Figure 2 we show an example tenant networking configuration in the Horizon GUI of our university's OpenStack deployment. The tenant wants to deploy a web server connected to a database server. Its policy is that that its web server should be publicly accessible on port 80, and that the database is only reachable from the web server, and not the Internet.

The tenant configures two VM instances: B will run the web server and is connected to the green VLAN, and C is the database server and is connected to the red VLAN. The two VLANs are connected via router R2, and the green VLAN is connected via router R1 to the Internet. R1 is setup as an Internet gateway. The blue VLAN (VLAN9) is created automatically by this particular OpenStack deployment and is where the Internet gateway resides. The tenant also starts A for testing purposes and attaches it to VLAN9.

Is this network configuration a correct implementation of this tenant's policy? An experienced OpenStack network administrator will, most likely, be able to debug this configuration quite easily and find that, for instance, there is no incoming connectivity to the web server since a floating IP has not been assigned. The average OpenStack tenant, however, will not be a networking specialist. Such a person needs a fairly large set of skills: they need to be able to create a VM image, they need to install and manage various servers (e.g. web and database), setup the tenant network and, finally, develop their application logic. In the pre-cloud era, multiple people were needed maintain such a website: e.g. a networking administrator, a web admin, a web developer. In the cloud era, it is possible (and expected) that a single person will fulfill all these roles, but they will not be networking experts.

Ensuring that a tenant policy is met by a network configuration requires more than manual debugging - we need tools that help the tenant quickly find the problems and fix them.

Cloud provider view of networking. When deploying Neutron, the cloud provider has to meet the above goals (correctness, isolation, performance) in the context of its local network policy, and with the added constraint of isolating tenant traffic from local traffic and treating tenant traffic as "outside" traffic when deciding access to local machines.

When the tenant starts the deployment of its configuration, virtual machine instances are placed on the available OpenStack Compute Nodes and the tenant networking configuration is instantiated; the instantiation depends on the way the cloud provider has deployed Neutron.

One of the most popular networking deployments is to use Openvswitch [10] on every Compute Node, use iptables for firewalling and have a Network Node running on one of the servers to implement NAT and routing. Traffic leaving from the Compute Nodes is encapsulated in VXLAN tunnels and carried to the Network Node, which also enables Internet

connectivity. This configuration meets all the requirements, except the performance one.

There is however great flexibility in how a cloud provider can configure its network to work with Neutron, including:

- Using fault-tolerant Network Node implementations called VRRP.
- Deploying routing and firewalling on every Compute Node, in a configuration called DVR.
- Using hardware switch support and VLANs to ensure the tenant traffic isolation instead of VXLAN.
- Using merchant hardware to implement routing and firewalling (called firewall as a service).

A performance-oriented deployment would use SR-IOV for each tenant, bypassing the hypervisor stack, and apply VLAN encapsulation on the NIC. VLAN support in switches would then be used to carry traffic to a hardware firewall (e.g. a CISCO ASA box), where firewall rules belonging to the cloud provider and tenants would be applied. Routing between VLANs and NATting could also be implemented in hardware. In such a deployment, ensuring the isolation and correctness properties hold is trickier: tenant firewall rules could clash or overlap with provider rules, and the correct order in which they should be applied is not obvious. Installing a set of predefined rules for all tenants is feasible, however allowing per-tenant firewall policies is difficult to do.

This versatility of Neutron makes it adaptable for a wide range of uses and requirements. However, it also raises questions about the correctness of any non-trivial individual deployment, especially when less scrutinized third-party software or hardware are used.

III. Symbolic Network Execution

We propose to use static network analysis to improve Neutron. There are many static network analysis tools such as [3], [5], [6], [9], [13], [14]; they all require as input a model of network functionality including the processing performed in different boxes, such as switches and routing, as well as a snapshot of the network state, for instance router forwarding table snapshots or switch dynamic MAC tables. Then, the tools "simulate" what happens when certain packets are injected at different parts of the network: given a packet with specific header fields, static analysis tools tracks the path of the packet through the network and the evolution of its header fields.

The strength of static analysis stems from its ability to quickly test a wide range of possible packets (e.g. all possible headers destined to a server) without having to iteratively test all possible combinations of concrete header fields. How this is achieved depends on the tool being used. The different tools offer different tradeoffs in terms of speed of analysis and properties checked, and an overview of all these tools can be found in [13]. A very good option is to use network symbolic execution, as enabled by the SymNet symbolic execution tool. We use SymNet in this paper and provide a brief overview below; please refer to [13] for more details.

In SymNet, network boxes are modeled as modular elements having an arbitrary number of input and output ports. Network links are modeled as directed edges between the output port of an element to an input port of another one. To describe the functionality of a box, each port has associated a set of instructions that are executed when a packet reaches that port. The set of instructions associated to each port is written in a language called SEFL and described in detail in [13]. SEFL is a simple imperative programming language and offers usual instructions e.g. assignment, if and basic expressions (addition, subtraction). SEFL is optimized to allow scalable network symbolic execution as follows:

- The `constrain` instruction adds restrictions on header fields, such as firewall rules.
- The `forward` instruction makes a packet go to a specified output port.
- The `fork(p1,p2,...)` instruction sends a copy of the packet to each of the specified output ports (p1, p2, ...).
- There are no unbounded loops in SEFL.

We give an example in Figure 3 where we have two network elements: a three-port Openflow switch and a firewall connected to port 2 of the switch. Element input ports are numbered in red and shown as triangles; output ports are numbered in green, and shown as squares. Port 0 is connected to an inside network, and port 1 is connected to the Internet. The configuration aims to ensure that all packets are checked by the firewall, and performs ingress filtering for the ports connected to the two networks. Packets from the firewall port are sent to the local network or the Internet based on their destination address.

```
InputPort(0):
   Constrain(IpSrc in 141.85.37.0/24)
   Constrain(IpDst not in 141.85.37.0/24)
   If (Constrain(TcpDst==80||TcpDst==443),
      Forward(OutputPort(2)),
      Forward(OutputPort(1)))

InputPort(1):
   Constrain(IpSrc not in 141.85.37.0/24)
   Constrain(IpDst in 141.85.37.0/24)
   If (Constrain(TcpSrc==80||TcpSrc==443),
      Forward(OutputPort(2)),
      Forward(OutputPort(0)))

InputPort(2):
   If (Constrain(IpDst in 141.85.37.0/24),
      Forward(OutputPort(0)),
      Forward(OutputPort(1)))
```

Below we provide the model for the firewall which only allows HTTP traffic from our local network and the associated return traffic. To keep per-flow state the model creates a metadata called `FirewallState` and sets in the packet. When the return traffic arrives, it will have the same variable set and will be allowed through.

```
InputPort(0):
If (Constrain(FirewallState==1),
   Forward(OutputPort(0)), //allow seen flows
   InstructionBlock(//outgoing HTTP traffic
     Constrain(IpSrc in 141.85.0.0/16),
     Constrain(IpDst not in 141.85.0.0/16),
```

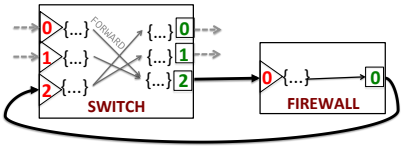

Fig. 3. SymNet network models: elements and interconnections

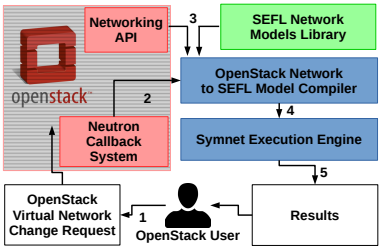

Fig. 4. Verifying the tenant network

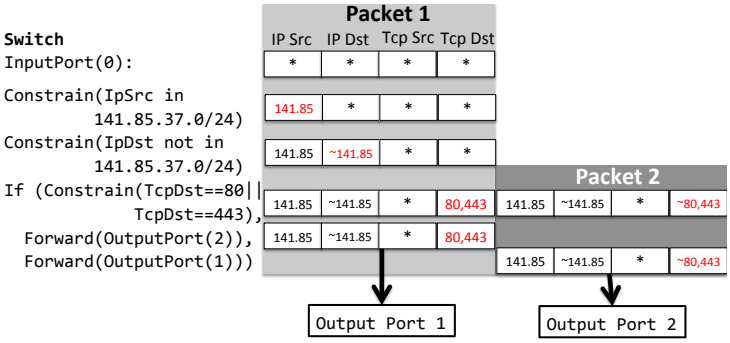

Fig. 5. Symbolic execution example: injecting a symbolic packet in the switch model on input port 0. Two packets are generated with different constraints, one that exits on port 1 and one on port 2.

```
    Constrain(TcpDst==80||TcpDst==443),
    Allocate(FirewallState),
    Assign(FirewallState,1),
    Forward(OutputPort(0))
))
```

To understand symbolic execution, we inject a packet on input port 0 of the switch and trace its evolution in Figure 5 (only switch processing on port 0 is captured in the figure). The packet has all header fields initialized to symbolic values:

1) The values of IpSrc and IpDst are constrained. Then, the If instruction results in two packets (or symbolic execution paths).
2) The "else" branch packet, packet 2, captures non-HTTP traffic and is forwarded to the switch output port 1; at this point symbolic execution of packet 2 stops, as output port 1 is not connected in our model.
3) On the If branch, the TCP destination port is constrained to be HTTP(S), and the packet is forwarded on output port 2 and onto the firewall on port 0.
4) The packet is processed at the firewall. There is no FirewallState in the packet so the else branch runs, and constraints are added for TcpDst. The state is allocated and assigned and the packet sent to output 0.
5) The packet enters the switch via input port 2, the else branch is run and the packet finally reaches output 1.

The output of symbolic execution is 1) a set of packets that have reached unconnected output ports, together with 2) a set of packets that have failed en-route, because some of the constraints applied to their header fields did not hold. The first category is more interesting for network verification. For each path, SymNet reports in a JSON-formatted output file all the instructions executed, all the ports visited, the values and/or constraints for all header fields.

In particular, if we examine the output from SymNet for the example above, we can conclude that outgoing reachability is permitted for all packets with correct IP addresses. We also find that the packet headers are not modified by our configuration: all header fields are bound to the same symbolic variable at the beginning and end of the execution. To gain more insights in this configuration, we also inject a purely symbolic packet at input port 1, and also add a symbolic value for the FirewallState variable. The results show that all HTTP response traffic is dropped unless firewall state is set to 1, and all other traffic is allowed unmodified.

IV. Analyzing OpenStack with SymNet

We provide a high level description of how SymNet can be used to improve Neutron to achieve all the its goals, namely tenant policy compliance, implementation correctness, traffic isolation and performance. To this end, we will rely on SymNet for network verification on two layers: the abstract tenant network and the deployed network.

A. Checking the abstract tenant network.

To check policy compliance, the tenant can use SymNet to run reachability from all instances to all other instances and the Internet before the configuration is deployed. The SymNet output allows the tenant to quickly check whether the reachability matches their expected behavior.

We have implemented support for such testing in Neutron and our implementation is shown in Figure 4. Whenever a tenant uses Neutron to create or modify a virtual network topology (step 1) and prior to the effective deployment, we insert an additional verification step that uses SymNet to analyze the tenant configuration. The deployment process is stalled until the analysis process ends.

To receive information about the creation of new topologies or the modification of existing ones, SymNet registers to the Neutron callback system. Every time such a callback is executed (step 2), SymNet queries the OpenStack HTTP Networking API [1] for the Neutron network configuration currently in place. Next, we need a SEFL model of the tenant network. We have modeled the functionality offered by Neutron including routers, firewalls, NATs and created a library of SEFL models. For every network function used by the tenant, we obtain the corresponding SEFL element by configuring the generic SEFL library component with the parameters provided by the tenant (step 3). The element's input and output ports are then interconnected according the virtual links provided by the tenant.

Finally, symbolic execution is performed by injecting symbolic packets at all the instances' network attachment points, as well at the Internet gateway. The result is detailed reachability for all VMs in the abstract tenant network. Currently, the result of the analysis is provided to the tenant in JSON format, which manually checks whether it obeys its policy. In the future, we plan to automatically check simple popular policies, such as:

- All-to-all VM communication for ICMP, UDP and TCP traffic.
- Outgoing ICMP and TCP reachability from all instances.
- Incoming SSH reachability (TCP on port 22) for all instances.

B. Checking the deployed network dataplane.

The second step of our verification happens *after the tenant's configuration is deployed*, and its goal is to ensure *isolation of tenant traffic* and *correctness of implementation*. To this end, we use two complementary approaches: guided testing and symbolic execution.

Guided testing [13], [15] is very simple: for every path resulting from the tenant network analysis, generate a matching packet and inject it in the actual network, observing the packets reachable at other instances or in the Internet. The big advantage for guided testing is that it can be run by the tenant, without cloud provider support and is independent of the deployed network. We have implemented a simple version of guided testing by using SymNet to generate test packets for the provided paths, generating packets using the Click modular router [4] and using `tcpdump` for reception.

Guided testing is not exhaustive: even if it reports success for the tested packets, there are no guarantees the deployment is indeed correct, or that tenant traffic is correctly isolated.

To get hard guarantees we resort to symbolic execution of the deployed network. Compared to the abstract tenant network, the setup needed to symbolically analyze the real network is much more complex: we need accurate SEFL models and snapshots of the dataplane state for all the boxes (hardware and software) deployed in the cloud-operator network that interact with tenant traffic. This includes network ports for *all* instances of all tenants, hypervisor functionality (software switching, tunelling and local filtering), OpenStack network nodes, hardware switches and routers.

Creating a solution that is applicable to all networks is an extremely challenging task, given the heterogeneity of deployed infrastructure and the pervasive use of middleboxes [12]. Generating accurate SEFL models of middlebox functionality is not trivial and requires a lot of expert effort. There is currently no generally applicable recipe to all networks. Modeling real networks, however, is feasible. In prior work we have developed a model of our department's network [13]. This model relies on the following building blocks: a) a switch model that can be automatically created when given a snapshot of the dynamically-learned MAC table; b) a router model created from a snapshot of the forwarding table, obtained via standard commands on Cisco routers, and c) a CISCO firewall model (Application Security Appliance) that is created automatically given the configuration file. We are currently using this model as a basis to implement OpenStack deployment checking in our network.

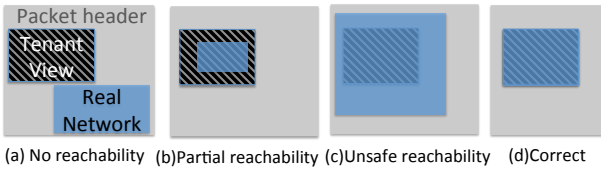

Fig. 6. Checking for equivalence between the tenant abstract network configuration and the deployment

Given an accurate model of a deployed network, we can use symbolic execution to ensure key properties for Neutron. We initiate reachability checks from the new tenant's instances and from the Internet and use the SymNet output to check isolation and correctness, as described next.

Checking isolation. If any VM from any other tenant is reachable from or reaches the instances of the new tenant, we report a violation of the isolation properties. Symbolic execution enables this analysis, but it is complex because its runtime depends grows linearly with the number of tenants/VMs. Optimizations are needed to ensure it scales to large clouds and may include only checking outgoing connectivity from the new tenant, or defining tenant equivalence classes and running reachability between equivalence classes.

Checking correctness. We can compare the sets of paths resulting from the abstract tenant view and the deployment view to decide whether the deployment correctly implements the tenant network. In essence, we want to decide whether the two configurations are equivalent. Equivalence is undecidable for general programs, however we restrict our definition to reachability: we want to ensure the same packets are reachable in the two configurations.

Before we describe the algorithm, we give a few definitions. We assume all packets contain the same set of headers H_1, H_2, etc., for which we care to verify equivalence; all other headers are ignored. Let $C_{H_i}(P_j)$ denote the set of constraints of header field H_i in packet P_j at some reference port in the network. Let $C(P_J) = C_{H_1}(P_J) \cap C_{H_2}(P_J) \cap \cdots$ denote the conjunction of all constraints applied to all headers of packet P_J at the same port.

Output Equivalence Algorithm. *Input:* The set of symbolic packets P_i, $i = 1...N$ and Q_j, $j = 1...M$ obtained by checking reachability between ports a and b in the tenant network and real network, respectively.

Algorithm: To test for equivalence, first we compute the disjunction of constraints of all packets in the two networks at port b: $X = C(P_1) \cup C(P_2) \cdots \cup C(P_N)$ and $Y = C(Q_1) \cup C(Q_2) \cdots \cup C(Q_M)$. The two sets of paths are equivalent if and only if the expression $(X \cap \neg Y) \cup (\neg X \cap Y)$ is not satisfiable.

The intuition for this algorithm is given in Figure 6 where we show a packet with two header fields. The hashed areas corresponds to the reunion of constraints at node b resulting from symbolic execution of the tenant network and the actual

network, respectively. The output equivalence algorithm aims to determine if the two areas overlap perfectly.

We have four cases: first, the tenant reachability does not overlap at all with the deployment reachability, resulting in a completely broken instantiation of the tenant network: the tenant has no reachability for its traffic, while unwanted traffic is allowed. In the next case we have partial reachability, but at least the configuration is safe: unwanted traffic is stopped. The third configuration allows full reachability, however also allows other packets through—this could indicate that the firewall rules have not been instantiated properly. Finally, the last case is when there is perfect overlap, and the two configurations are equivalent from an output port point of view.

Note that output equivalence is fairly weak in networks that modify header fields. There, additional constraints could be applied on the initial values of the header fields that influence reachability, yet they are not captured by output equivalence. In our future work we plan to explore stricter notions of equivalence such as input-output equivalence.

V. Preliminary Evaluation

We ran reachability analysis for the tenant configuration in Figure 2 by injecting a symbolic packet at all the tenant instances and the gateway. It takes 5 to 7 seconds to run the reachability analysis, for which 3 to 5 seconds are spent in Neutron API calls and 2s to generate the SEFL code and run the reachability analysis.

Our first analysis showed no connectivity at all because the tenant configuration didn't have static IPs assigned and no DHCP server was enabled either. We changed the configuration by assigning static IPs to all the instances; the analysis found that all instances can communicate directly. Further, A and B have outgoing Internet connectivity, and that connections initiated from the Internet are not allowed. Finally, C has no Internet connectivity.

Next, we deployed the configuration and ran our guided testing implementation. Surprisingly, guided testing showed that instance C had Internet connectivity but it couldn't access instances A or B; this is the exact opposite of the tenant configuration, where C has no Internet connectivity but it can reach A and B. It turned out that this problem was transient: when we reran the guided testing scheme, the results were as expected. The culprit was identified to be the propagation latency between high level commands and driver implementation in Neutron, which we measured to be on the order of minutes in our deployment.

VI. Conclusions

OpenStack Neutron is a complex piece of software, providing different views to different stake-holders and incorporating code from multiple parties. It has been anecdotally called the weakest link in OpenStack [11].

We argue this happens because debugging currently only relies on standard best practices for code development, without taking into account the particularities of Neutron. As a complement to existing approaches, we propose using static network analysis to improve OpenStack Neutron. We have shown how network symbolic execution can be used on two levels: to check the abstract tenant network and its deployment in the actual network. We have an initial implementation of these ideas that we have integrated with the Neutron API. Our preliminary evaluation has found interesting nuggets: a propagation delay bug in OpenStack (that was known) and can help tenants more easily deploy their networks.

This is a work in progress, and many refinements are needed to our prototype until it can be applied to a wide range of configurations automatically. On the algorithmic side, we intend to explore stronger version of equivalence. Finally, we intend to fully model our department's network (including Openvswitch, Neutron nodes, etc) and perform full symbolic analysis of the deployed network.

Acknowledgements

This work was partly funded by Superfluidity H2020 (671566).

References

[1] OpenStack Networking API 2.0 Specification. http://developer.openstack.org/api-ref-networking-v2.html.

[2] OpenStack users share how their deployments stack up. http://superuser.openstack.org/articles/openstack-users-share-how-their-deployments-stack-up.

[3] P. Kazemian, G. Varghese, and N. 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.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[5] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *NSDI*, NSDI'15, pages 499–512, Berkeley, CA, USA, 2015. USENIX Association.

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

[7] Michael Till Beck and Martin Werner and Sebastian Feld and Ludwig Maximilian and homas Schimper. Mobile Edge Computing: A Taxonomy. In *AFIN 2014*.

[8] Openstack. Neutron Networking. https://wiki.openstack.org/wiki/Neutron.

[9] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying Isolation Properties in the Presence of Middleboxes. Tech Report arXiv:1409.7687v1.

[10] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *NSDI*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[11] T. Register. HP: OpenStack's networking nightmare Neutron 'was everyone's fault'. http://www.theregister.co.uk/2014/05/13/openstack_neutron_explainer/.

[12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.

[13] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: scalable symbolic execution for modern networks. http://arxiv.org/abs/1604.02847, 2016.

[14] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings of Infocom*, 2005.

[15] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 241–252, New York, NY, USA, 2012. ACM.