SUPERFLUIDITY

A SUPER-FLUID, CLOUD-NATIVE, CONVERGED EDGE SYSTEM

Research and Innovation Action GA 671566

DELIVERABLE I6.3B:

MODELLING AND DESIGN FOR SYMBOLIC EXECUTION AND MONITORING TOOLS

Deliverable Type:	Report
Dissemination Level:	PU
Contractual Date of Delivery to the EU:	01.07.2017
Actual Date of Delivery to the EU:	21.07.2017
Workpackage Contributing to the Deliverable:	WP6

Editor(s):	Costin Raiciu (UPB)
Author(s):	Costin Raiciu (UPB), Radu Stoenescu (UPB), Matei Popovici (UPB), Dragos Dumitrescu (UPB), Omer Gurewitz (BGU)
Internal Reviewer(s)	

Abstract:	This internal deliverable includes the modelling and design for Task 6.3: the symbolic execution checking tool and the automatic monitoring and anomaly detection
Keyword List:	

1. Introduction

In the first year, Superfluidity has shown it is possible to perform exhaustive symbolic execution for network verification via the SEFL modelling language and the Symnet symbolic execution tool. Since then, work has continued to apply SEFL/Symnet verification to real-life problems and verify actual networks.

The present document is a snapshot of the work undertaken by Superfluidity in the area of security, correctness and verification, that has focused on extending our prior work in a few key directions:

- Designing NetCTL, a policy language that operators can use to express their requirements, and an accompanying verification tool that takes the policy as input and drives the symbolic execution (with Symnet) to check whether the policy holds (Section 2).
- Continuing the work of integrating Symnet in Openstack Neutron (Section 3).
- A novel data-structure that allows cost-optimal modeling of match-action tables and very small incremental costs for rule updates (Section 4).
- A reactive monitoring approach that complements our verification work (Section 5).

2. NetCTL: Policy language to drive symbolic execution

NetCTL is an adaptation of CTL - Computation Tree Logic, a language designed an successfully deployed for program verification.

In CTL, temporal operators such as **`F`** (i.e. sometime in the future) and **`G`** (i.e. always in the future) are combined with the path quantifiers **`exists`** (on some path) and **`forall`** (on all paths). NetCTL borrows these operators and adds SEFL code to describe state-based properties. For instance, the policy:

```
forall F destTCP = 80
```

evaluated at some node A of the network, expresses that on all possible packet paths from A, **`destTCP`** will eventually become 80. The policy combines the quantifier **`forall`** with the temporal operator **`F`**. The construction **`destTCP = 80`** is a SEFL code which describes a network property evaluated at a hop in the topology.

Unlike other, more expressive temporal languages such as CTL*, CTL requires that each path quantifier be directly preceded by a temporal operator. This syntactic restriction ensures CTL's desirable computational properties: model-checking is linear with respect to the size of the model and that of the formula. NetCTL benefits from the same computational properties - it allows verifying a network topology by doing a *single-pass* (i.e. a single symbolic execution).

More formally, the syntax of NetCTL is given below:

```
φ ::= SEFL | ~φ | φ ∧ φ | XY φ | X φ until φ
```

where **X ∈ {exists, forall}, Y ∈ {F,G}**. The syntax includes the standard boolean operators, the CTL operators already discussed, and the **until** operator : the formula
p until q expresses that **p** is true until **q** is true.

Unlike other policy languages, NetCTL is compositional: it allows expressing more complicated policies, starting from simpler ones. For instance, the following composed policy:

```
forall G (ip != 192.168.0.0/16 → exists F port == Internet)
```

evaluated in a network node A, expresses that on all paths starting from A, at each hop, if the IP destination of a packet becomes public, then there exists a path which reaches Internet. We have found that many interesting network policies can be naturally expressed in \netcheck. We illustrate a few such examples:

Traffic isolation
An example of a traffic isolation policy is: ``*all traffic destined for A must pass through an IDS*". The NetCTL policy which expresses this behaviour is:

```
forall (port !=A until port == IDS)
```

The policy states that on all paths, the current port is either different from A (hence A has not yet been reached) or is equal to IDS (and henceforth no other restrictions apply). In order words, the policy expresses that a packet cannot reach A before it reaches the IDS.

TCP end-to-end connectivity

Suppose we have nodes A and B of the network. TCP connectivity ``at equilibrium" (for simplicity, we ignore connection establishment or timeouts) between A and B means that any TCP packet from A destined to B will indeed reach B; also, a reply from B which will eventually reach back to A.

To verify this behaviour, we must add the following code snippet at the input port of A:

```
allocate(x); allocate(y);
x = TCPSrc; y = TCPDst
```

which stores the source and destination TCP fields in variables x and y. We also add the following code at the output port of B:

```
allocate(tmp);
tmp = TCPSrc; TCPSrc = TCPDst; TCPDst = tmp;
deallocate(tmp); Forward (B);
```

which swaps the source and destination TCP fields, thus emulating a reply. Finally, the NetCTL policy which captures end-to-end connectivity is:

```
exists F (port == B ⋀ forall G (port == A ⋀ TCPSrc ==y ⋀ TCPDst == x ))
```

To verify it, we introduce a fully-symbolic packet at node A of the network. The policy expresses that:

- there exists a path from A on which the port becomes B (hence B is reachable)
- on all paths starting from A, if A is again reachable, then the source and destination TCP fields are flipped.

Tunnel invariance

Suppose we would like to check that, between nodes A and B of the network, a certain field **`header_val`** is unchanged. This behaviour would ensure that a tunnel between A and B works correctly. We first add the following code at A which introduces a fully symbolic value:

```
allocate(crt_val);
crt_val = *
```

Our policy for tunnel invariance is:

```
forall G ( port == A; crt_val = header_val} →

          forall G (port == B → header_val == crt_val ))
```

We note that the code snippet **`port == A; crt_val = header_val`** performs a verification (the port must be A) as well as a *state-change*: **`crt_val`** becomes equal to **`header_val`**. Thus, the policy expresses that on all paths, at each hop, if the port becomes A:

- store the contents of **`header_val`** in **`crt_val`**
- on all subsequent paths, at each hop, if the port becomes B, then **`header_val`** must have the same constraints as **`crt_val`**, i.e. its original value.

Verifier implementation

Our NetCTL verifier extends Symnet and has been written in Scala. It performs policy verification in time O(n*m) where n is the size of the model (the total number of instructions) and m is the size of the policy.

The implementation differs from the standard CTL model checking algorithm and exploits the fact that, conceptually, SEFL models have a tree-like structure.

When trying to satisfy an existential or universal path quantifier (i.e. a formula such as `X` φ where `X` $\in$ `{exists, forall}`) at a branching instruction, the verifier will selectively branch program execution. If the policy φ is false on a program path, then forall φ is false the branching point --- symbolic execution need not continue. Conversely, if φ is true on a path, then exists φ is true and the verifier will not explore the other paths.

When trying to satisfy a temporal operator (e.g. `F, G`), the verifier will behave in an analogous fashion. For instance, when verifying a policy of the form `F` φ (resp. `G` φ), symbolic execution will stop with success (resp. failure) whenever a state where φ holds (resp. does not hold) is found.

The entire verification process corresponds to a (partial) depth-first traversal of the symbolic execution tree of a model, starting from an initial node.

Discussion

NetCTL is *at least as expressive as* existing network policy languages, to the extent of the authors' knowledge. We can easily embed any policy from e.g. Merlin, Procera, NetPlummer (HSA) into NetCTL. Moreover, by relying on SEFL, our language can verify network properties which: (i) are difficult to model by existing approaches, (ii) are not (easily) expressible in existing policy languages.

NetCTL is fully-automated, and relies on the SEFL & Symnet ecosystem to generate models from existing networks. NetCTL **guarantees** network correctness over such models.

The NetCTL-based verification procedure can offer strong correctness guarantees to the extent to which the SEFL network models are accurate. While SEFL is more expressive than most existing modelling languages (e.g. HSA), it cannot capture certain box transformations such as: (i) packet reordering, (ii) packet fragmentation/assembly, (iii) traffic shaping, (iv) selective/random dropping of packets.

Also, SEFL is not designed to capture certain implementation bugs from middleboxes, which may affect now packets are handled by the network.

Thus, NetCTL delivers a *best-effort* verification. If a policy is found to be false, acounter-example (i.e. the packet trace which violates the policy) is provided. The existence of a gap between the network model and the actual network infrastructure means that a true policy does not guarantee correct behaviour in the implementation, in all situations.

One alternative to addressing the model-implementation gap, is to *generate executable network processing* from SEFL models. Thus - model verification renders network *correctness by design*. We are currently exploring this direction, for networks with simple (layer 2/3) traffic processing.

3. Openstack Neutron verification with symbolic execution

In this section we discuss how we can verify the network configurations of OpenStack, the leading cloud management software, OpenStack is an open-source cloud platform software deployed as an Infrastructure-as-a-Service architecture. OpenStack abstracts away the complexity of a computing environment and provides the user with a set of services to create, manage and use compute stacks in rich network topologies transparently. The interaction of the user with OpenStack is ensured using a set of public REST APIs and a graphical user interface. Internal communication within the cloud data-center is ensured via a set of private APIs implemented as RPCs (remote procedure calls).

OpenStack exposes a set of services to the users. The most important one, the compute service (nova) provides the user with the capability of launching machines as needed to be run in the provider data center. OpenStack also exposes image, block storage, authentication and authorization, service discovery, quota management and remote access services.

A user may be assigned to one or more tenants. A tenant (or project) provides the abstractions to handle separation between multiple customers in the same cloud. For instance, a company may choose to acquire an OpenStack tenant from a cloud provider and assign a number of users to it as they see fit. Thus, the internal services provided for the company are hidden from those of a different company.

Using the compute service, the user may launch a number of machines. It is the task of OpenStack's scheduler to handle assignment of each virtual machine launched by a user to a given hypervisor host within the topology. The hypervisor hosts are known as compute nodes. It is the purpose of the networking service to provide connectivity between multiple machines within a given tenant and to the Internet.

From a historical perspective, previous releases of OpenStack used nova networking as a service for providing the basic network functionalities required by OpenStack. However, the customers' needs to create rich network topologies within their tenants has led to the development of Neutron (previously Quantum) service. It provides the main abstractions that allow networking availability to the user and offers the possibility to create logically separated networked environments within tenants.

Initially, the nova networking service was using a that scheme, consisting of one or more shared networks (provider networks) using a common IP addressing scheme and some segmentation features meant to keep traffic between tenants separate. However, there was no possibility to create private networks within a given tenant, nor were other services available, such as VPN,firewall etc. In the following, the main features of Neutron are described.

The main services that Neutron offers to the user are: networks, subnets, routers, VPNs, load balancers,firewalls, ports, DHCP, security groups. The concepts underpinned by the

aforementioned services are well-known in classic networking infrastructures and provide well- defined functions to the user.

For the scope of the current project, an OpenStack deployment can be seen from two distinct perspectives: the tenant perspective and the deployment perspective. The tenant perspective refers to all abstractions that are provided via OpenStack public APIs to the end-user (e.g. routers, networks etc.). The deployment perspective refers to all components that make up the underlying implementation of OpenStack (e.g. agents, plugins, drivers etc.) and the configurations deployed on the machines that represent the OpenStack environment (e.g. OpenFlow tables, iptables tables etc.)

As described previously, it is important to underline that modeling the behavior of an OpenStack system from one perspective or the other may yield different results. The bottom line of this research endeavor is proving equivalence of the two symbolic executions.

Neutron Architecture

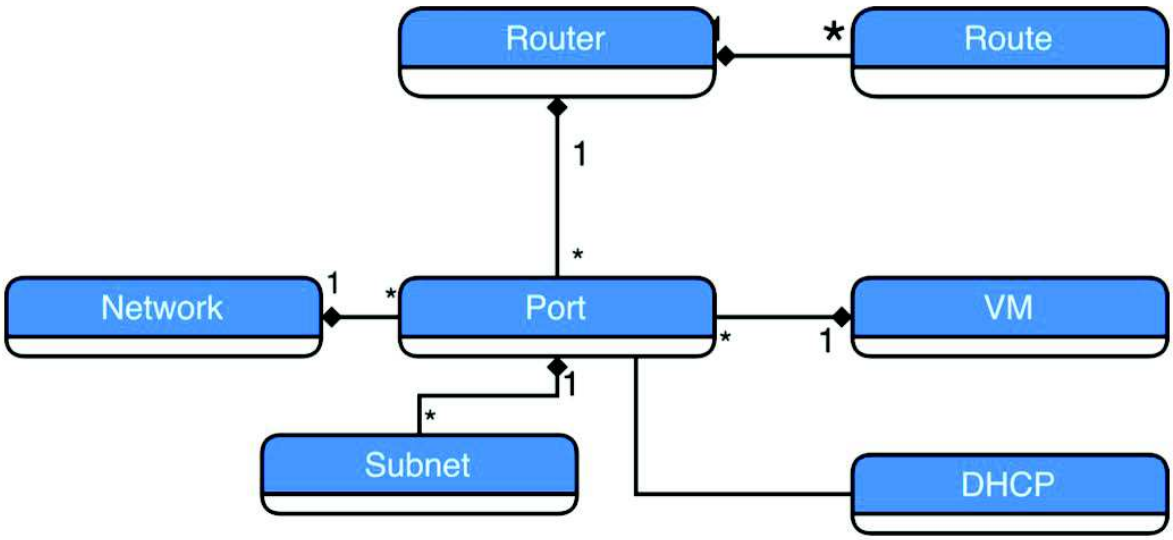
The OpenStack networking services of interest are: routing, floating IPs, self-service (per-tenant) networks, subnetworks. The metadata and DHCP services are not of interest for this matter and will not be analyzed further on.

As specified in the OpenStack Networking Guide, Neutron has a number of basic components:

- A Neutron Server - handles API service requests (e.g. at tenant level, allows for creating, retrieving, deleting and updating information on tenant level resources)
- Neutron Agents - provide Layer-2, Layer-3, DHCP and Metadata functions
- Neutron plug-ins - manage agents

A commonly used implementation for Neutron resides in the use of ML2 (Modular Layer 2) plug-in to handle agent management. Simply put, the ML2 provides a framework for deploying a number of drivers which handle connectivity and correct implementation for tenant-defined networks, as well as offering the possibility for tenant-based separation and segmentation.

Within the framework of ML2, the implementation analyzed is composed of multiple OVS (OpenvSwitch) bridges, a set of Linux ip namespaces with multiple interfaces and a set of iptables tables implemented within the aforementioned namespaces.



The object model of interest for the tenant perspective of OpenStack networking can be seen in the Figure above. As can be easily observed the central component in the diagram is the port. Each port connects a distinct piece of equipment, such as a router, a DHCP server or a VM and translates into a tap interface inside a Linux PC.

From the deployment perspective, an OpenStack cloud is composed of a number of machines that perform several functions with respect to their high level speci□cations.

- A machine in the system is called node.
- A node can be either a compute node or a network node.
- Each VM will run on a compute node.
- Each Neutron router will run on a network node.
- Each compute node will handle security groups per machine.
- Each node will have at least two OVS bridges: the br-tun and the br-int.
- Each node may have several bridges pertaining to a provider network.
- Each Neutron port maps to a tap network interface within the system.
- Each tap interface is connected on the br-int bridge.
- Tunneling is performed by br-tun.
- Each OVS bridge may behave like an OpenFlow bridge or like a normal L2 bridge.
- Each node distinguishes Neutron network tra□c based on VLANs which are scoped to the node they are on.
- Tunneling assigns a system-wide unique tunnel id to a Neutron network.
- br-tun translates from tunnel id to scoped VLANs
- br-int and br-tun are directly connected.

The implementation considered for symbolic execution is using the ML-2 plug-in with L3 and OpenvSwitch L2 agents using VXLAN as means to provide spanning topology. Each compute node runs an OpenvSwitch layer 2 agent which performs all the switching and

segmentation logic. The L3 agent is implemented as a standalone network node which performs routing for each router resource declared in the tenant perspective of the topology.

In the following, structural requirements imposed upon some node in the system are presented. In general, assuming some packet flow, one can state that the packet-processing components that make up a node within an OpenStack deployment are:

- A node has one or more namespaces.
- A namespace has one or more interfaces.
- A node has one or more bridges.
- A bridge may connect one or more interfaces.
- A namespace has a routing table.
- A routing table is composed of one or more routes.
- A namespace has one or more iptables tables.
- An iptables table is composed of multiple iptable chains.
- An iptables chain has one or more rules.
- A rule has one or more matches.
- A rule has one action.
- A bridge can be an OVS Bridge or a Linux Bridge.
- An OVS Bridge contains multiple OVS interfaces.
- An OVS interface may be access port or trunk port.
- An access port has an associated VLAN tag.
- A trunk port may transport some or all VLAN tags.
- An OVS Bridge is configured via OpenFlow Configuration.
- An OpenFlow configuration has a set of OpenFlow ports.
- An OpenFlow port maps to an OVS interface.
- An OpenFlow configuration contains a set of OpenFlow tables. • An OpenFlow table is composed of multiple flows.
- A flow may contain multiple matches.
- A flow may contain multiple actions.

Notice that the above list aims at depicting a Linux node from a purely structural perspective. The behavior of blocks, the interaction between components, as well as the order in which packet-processing elements are executed are neglected for the moment and will be described more thoroughly later. The above structure was derived from iptables manuals and user guides, OVS documentation and the OpenFlow specification.

Implementation

In order to achieve integration with the distinct pieces of technology, the following steps were taken:

1. Acquiring configuration data from all sources
2. Interpreting acquired information into internal structures
3. Modeling the packet-processing entities in SEFL.

4. Integrating the functional blocks modeled in SEFL to generate a full packet-processing pipeline.

The body of work pertaining to all the aforementioned items was two-fold. Thus, from the tenant perspective, the steps depicted above involved acquiring the data exposed through Neutron APIs as resources, understanding their semantics and then performing the modeling and integration tasks as prescribed. To that end, Symnet acquires tenant-level data either through the Neutron REST APIs, given knowledge of some valid credentials and a publicly exposed communication endpoint, or by reading and parsing files exported through the command-line utilities provided by Neutron.

From the provider perspective, the configuration data acquisition is performed by dumping configuration files of all components on all nodes in the Neutron deployment. Thus, information regarding Linux IP namespaces, iptables tables, OVS database dumps, OpenFlow dumps for all switches defined in the OVS database, routing tables, IP interfaces and OpenFlow port mappings to actual interfaces are required in order to perform a full symbolic analysis of the system.

Furthermore, internal structures were defined as convenience objects for querying necessary information in the modeling process. The information acquired and enumerated above is translated into packet processing blocks (in SEFL).

Modeling approach

In the following, the approach taken towards modeling the packet-processing blocks of the *Neutron-Symnet* implementation is described. Thus, the most important packet processing blocks identified are: *Netfilter* hooks and *OpenFlow tables*. The first are mostly employed for verifying correctness of the *Neutron L3 agent*, whilst the latter is employed for verifying correctness of the L2 agent and security groups implementation.

Netfilter hooks, and especially *iptables* chains are used for routing and filtering traffic within the L3 agent - i.e. the component which implements Neutron Routers. Each iptables table is composed of a series of *rules*. An iptables *rule* is composed of a series of *matches* and a *target*. As an example of *SEFL modeling* of an *iptables* match, the following listing is presented to describe source IP matching:

```
def matchSourceIp(startIp : IPAddress, endIp : IPAddress) {
      meta.matched = false;
      if (packet.EtherType == EtherType.IPv4) {
            if (packet.SourceIp <= endIp && packet.SourceIp >= startIp) {
                  meta.matched = true;
            }
      }
 }
```

Listing 1: Source IP match in iptables

In order to depict the modeling undertaken for an iptables target, a more challenging example is presented. Thus, the SNAT action is presented. Note that this particular action can only be fired for the first packet of a connection in the NAT POSTROUTING chain. As such, SNAT can be described as follows:

```
def fireSNATTarget(ip : IPAddress, portStart : Int, portEnd : Int) {
      meta.SNAT.DstOriginalAddress = packet.IPDst;
      meta.SNAT.SrcOriginalAddress = packet.IUSrc;
      if (packet.IPProtocol == IPProtocol.TCP) {
            meta.SNAT.SrcOriginalPort = packet.TCPSrc;
            meta.SNAT.DstOriginalPort = packet.TCPDst;
      } else if (packet.IPProtocol == IPProtocol.UDP) {
            meta.SNAT.SrcOriginalPort = packet.UDPSrc;
            meta.SNAT.DstOriginalPort = packet.UDPDst;
      } else if (packet.IPProtocol == IPProtocol.ICMP) {
            meta.SNAT.SrcOriginalPort = packet.ICMPId;
            meta.SNAT.DstOriginalPort = packet.ICMPId;
      }
      meta.SNAT.IsSNAT = true;
      packet.IPSrc = ip;
      if (portStart.isSpecified || portEnd.isSpecified) {
            if (packet.IPProtocol == IPProtocol.TCP) {
                  packet.TCPSrc = Symbol();
                  constrain packet.TCPSrc <= portEnd && packet.TCPSrc >=
portStart;
            } else if (packet.IPProtocol == IPProtocol.UDP) {
                  packet.UDPSrc = Symbol();
                  constrain packet.UDPSrc <= portEnd && packet.UDPSrc >=
portStart;
            }
      }
}
```

Listing 2. SNAT Target in SEFL

Connection Tracking

One of the most challenging components to model from a behavioral point of view is the connection tracking engine implemented within the Linux Kernel as part of the Netlter framework.
Same as iptables, the connection tracking engine is using the concept of hooks to perform a set
of actions on an input packet.
Conceptually, the connection tracking engine (or *conntrack*) denes a connection as a 5-tuple composed of source IP address, destination IP address, IP Protocol and two distinct L4 addresses meant to distinguish between multiple connections. At the entry of a packet within conntrack, it is assigned to a new entry inside a locally managed table. A packet may be found as already being part of an existing connection or may be a new packet, which will in

turn be assigned a new entry inside the table. If a connection is found, then the packet may be in the forward direction or in the backward direction. If a packet is not part of a known connection, conntrack assigns two new entries inside its
connection tracking table. The first sets forward expectations - i.e. how are forward packets mapped to this connection, while the second refers to backward packets - mapping return packets to the a connection in the table. The first packet from a connection dictates from the moment of its arrival, the expectations for forward and backward packets.
Conntrack also provides to the user-space a set of states for a given connection. The states of a connection are either: NEW, ESTABLISHED, RELATED, INVALID. Apart from those, two virtual states called SNAT and DNAT are available, which represent the fact that the packet has undergone SNAT, DNAT or the reverse operations priorly in the netfilter chain. At kernel-level, the state machine is more complex, as it takes into account protocol specific information, such as TCP flags for packets and their tracing. The high-level logic for a connection is as follows:

- On packet arrived if no connection corresponds ⇒ Create new connection with forward expectation congruent to the packet's fields and backward expectations corresponding to the packet's reversed fields; Set connection state to NEW
- On packet arrived if connection corresponds and connection state is NEW and packet matches backward expectation ⇒ Set connection to state ESTABLISHED
- On packet arrived if connection corresponds and connection state is ESTABLISHED ⇒ Connection state remains the same

For simplicity, the modeling approach in the current paper does not take into account the RELATED state. In conntrack, a connection state of RELATED is used to represent a packet not part of a regular protocol ow, but rather an auxiliary part of it (e.g. ICMP packets for signaling errors in FTP connections).

In what timing is concerned, conntrack has two of points of activation: in the PREROUTING chain and in the POSTROUTING chain. Thus, in the PREROUTING chain, conntrack is called to identify the connection which the current packet belongs to. It is at this point where conntrack writes new connections to the connection table and sets internal variables to point to the corresponding entry in the table. In the POSTROUTING chain, the packet is committed to conntrack, establishing the backward expectations for the connection. Packet filtering is accomplished in hooks between the two steps.

NAT
In relation to the connection tracking module depicted above, NAT features (destination and source NAT) must be integrated with the connection tracking logic. Internally, a structure similar to that of the connection table is kept for all packets which undergo SNAT or DNAT.

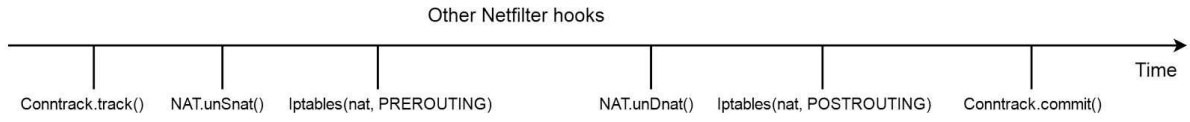
The point of insertion for NAT-related functions are the PREROUTING and POSTROUTING chains. In these chains, the following logic is applied: a packet passes through the iptables nat tables (either PREROUTING or POSTROUTING) if it is the first packet belonging to a connection.

For SNAT (source NAT), upon first packet of a connection arriving at POSTROUTING chain, the iptables nat table is traversed. Should any NAT action be executed, the packet is modified (i.e. its source network and transport layer source addresses are changed) and the original state of the packet is stored in the NAT table. Upon committing the packet to conntrack, the backward expectations for the packet will be those of the current packet (with all fields modified).

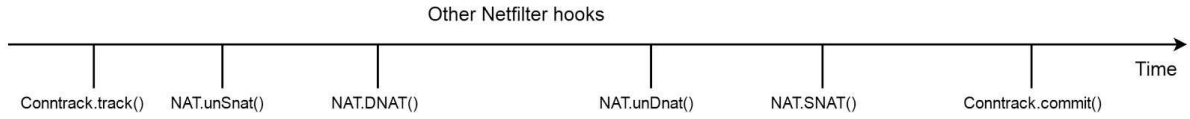
The reverse operation is applied in the PREROUTING chain after the conntrack track operation is executed and before any iptables PREROUTING traversals are performed. At this stage, the current packet is looked up in the NAT table and if a backward expectation is found, then it is transformed accordingly.

For a packet matching a forward expectation, the traversal of the iptables nat POSTROUTING chain is bypassed and the existing NAT mappings from the NAT table are applied.

For DNAT (destination NAT), a similar operation occurs, with the difference that the packet modification step occurs in the iptables PREROUTING chain, while the reverse step occurs in the POSTROUTING chain.



In the above figure, a set of functions which are applied to a newly arrived packet are depicted in the calling order defined within the netfilter framework. The representation does not take into account other iptables chains and the interactions they may exert upon the conntrack and NAT modules.



In the above figure, the same diagram is depicting the actions undertaken for a packet which already went through the connection tracking module and is not the first in the given connection. Notice that the calls to iptables nat table in chains PREROUTING and POSTROUTING are replaced with calls to the NAT modules corresponding to the insertion place.

All netfilter hooks (i.e. all iptables calls) performed between the PREROUTING and the POSTROUTING chains experience the packet with its internal source and destination addresses. If SNAT is regarded as a translation between an internal address to an external addresses, then all translation in a forward packet happens in the POSTROUTING chain. Meanwhile, all translations for a backward packet (i.e. un-SNAT) happen in the PREROUTING chain. The same goes for DNAT, with the difference that in the case of DNAT, the translation (DNAT) happens in the PREROUTING chain, since its semantics is

usually that of changing an external address to an internal one, while the reverse operation (i.e. un-DNAT) happens in the POSTROUTING chain.

OpenvSwitch Blocks

OVS (OpenvSwitch) bridges are important components of a Neutron deployment. The ML2 plug-in used in most installations relies on the OpenvSwitch driver to perform layer 2 bridging, tenant separation and filtering.

From a structural perspective, the OpenvSwitch bridge (OVS) can be seen as an object with a number of components:

- An *OVS bridge* is a *layer 2 software switch*.
- An *OVS bridge* has multiple *port*s.
- A *port* has multiple *interface*s.
- A *port* can be an *access or trunking port*.
- A *trunking port* may specify a list of *allowed VLANs*.
- A *trunking port* with no *allowed VLANs* performs trunking for all VLANs.
- An *access port* must specify the *VLAN tag* it belongs to.
- A *port* can be directly connected with a *port* in another *bridge*.
- All traffic that enters a *directly connected port* is directly passed to the *port* it connects to.
- A *port* can be directly connected with a *remote port* using some encapsulation technology (e.g. vxlan, gre, geneve, ipsec).
- All traffic that enters a *port directly connected with a remote one* is directly passed to the *remote port*.

All OVS behavior is specified using OpenFlow flows. A *flow* belongs to an *flow tables*. For more details, see OpenFlow specification and *ovs-ofctl* command's *manpage*.

- An *OpenFlow switch* has multiple *flow tables*.
- An *OpenFlow switch* has multiple *ports*.
- Each *flow table* is composed of multiple *flows*.
- Each *flow* has a *priority* in the *flow table* it belongs to.
- Each *flow* has multiple *matches* and multiple *actions*.
- A *match* is a *key-value* pair which specifies a *field* and a *value* to match.
- Whenever a *packet* matches the conditions imposed by the *set of matches*, the *action*s are marked for execution.
- An *action* is a modification on either *packet fields*, *internal switch variables* or *processing flow*.
- *Actions* are cumulative; thus, actions are marked for execution as flows are being matched, but are executed whenever a flow traversal ends or an *apply-actions* action is encountered.
- All packets not matching any *flow* are applied the *default flow* (in the case under study, the packet is dropped).
- When a packet enters a *port*, the *input port* internal variable is set to that port and the packet is sent for processing into *table* 0.

In Neutron, the actions that need to be taken by the OVS layer are of *Apply-Actions* type (see OpenFlow Specification for details). This choice eases the process of modeling the OpenFlow pipeline and makes it, to some extent, similar to that of iptables.

A modelling sample for OpenFlow input port match is depicted below:

```
def matchInPort(inPort : Int) {
      meta.matched = false;
      if (meta.in_port == inPort) {
            meta.matched = true;
      }
}
```

Listing 3. OpenFlow Input Port Match

The OpenFlow specification defines special port numbers for given applications (e.g. LOCAL, NORMAL, FLOOD etc.). For the scope of the current project the only special port encountered was the NORMAL port; in essence, it sends the packet out to OVS in order to perform normal layer 2 switching. At the moment of writing this paper, the current approach to modeling the Layer 2 OVS switch behavior is sub-optimal. The CAM table is not used to distinguish between destination ports and thus, the packet is simply forwarded on all ports except for the entry port. VLAN processing is taken into account for correct forwarding. Thus, the assumption that through an access port, only untagged packets can pass, while through a trunk port only 802.1Q-tagged packets can transit is asserted by the following code generation approach:

```
def l2Switching(bridge : OVSBridge, inputPort : Port) {
      if (inputPort.isAccess) {
            vlanEncapsulate(inputPort.vlanId);
      }
      for (vlan : bridge.knownVlans) {
            if (packet.VLANId == vlan) {
                  for (port : bridge.ports) {
                        if (port.isAccess) {
                              if (port.vlanId == vlan) {
                                    vlanDecapsulate();
                                    sendOut(bridge, port);
                              }
                        } else if (port.isTrunk) {
                              if (port.accepts(vlan)) {
                                    sendOut(bridge, port);
                              }
                        }
                  }
            }
      }
}
```

Listing 4. OVS Normal Switching

Integration

One of the most challenging aspects regarding handling complex software systems resides in making different pieces of technology work together. Throughout the course of the preceding subsections, a number of distinct blocks have been documented with focus on their packet-processing behavior. In the following paragraphs, a number of considerations regarding the way they interact are presented, as well as a general packet forwarding flow.

In Neutron, each tenant-level *port* maps directly to a network interface of type *tap*. The *tap interface* resides on a *compute node* and is connected to the OVS integration bridge. At this level, the security groups are enforced and switching behavior (such as VLAN tagging/un-tagging) is defined. Assume that a packet leaves out on some *tap interface* heading outbound. Then, the following actions apply:

- The packet enters the integration bridge which is an *OVS bridge*
- The packet enters the *OpenFlow* integration bridge through the *OpenFlow port* corresponding to the *tap interface*
- The *OpenFlow bridge* sends packet to table 0 for processing.
- The packet gets modified through the *OpenFlow pipeline*.
- If the packet goes through the *NORMAL* switching behavior, classical L2 switch behavior is implemented at *OVS* level.
- In the previous case, the switch will push a VLAN Id tag to the packet and flood it on all ports that belong to the given switch, except for the input port (i.e. the *tap interface*).
- If the packet exits through a *tap interface* with a VM attached, then the execution **ends** (at **VM**).
- If the packet exits through the tunneling bridge, then, based on the VLAN ID, it will be assigned a tunnel id, 802.1Q encapsulated and sent out to the *Network node*.
- At *Network node* level, the packet is de-capsulated and sent to processing out on the tunneling bridge.
- At tunneling switch level in the network node, the ID of the tunnel the packet came from is translated into an internal VLAN ID.
- The packet is tagged with the internal VLAN ID and is forwarded to the integration bridge of the network node.
- At integration bridge level, the packet is broadcast to the OVS ports tagged with the VLAN tag equal to that of the packet.
- Upon receiving the packet on an access port at integration bridge level, the packet enters the Linux ip stack.
- Iptables filtering is performed as per *netfilter* hooks in the kernel
- Connection tracking is performed for the packet and, depending on its layer 3 destination address, it may be forwarded or received internally.
- If the packet is destined for an interface in the current namespace, the packet is delivered locally and the execution **ends** (at **namespace**).
- Else, it is forwarded to either a bridged interface or an external interface.
- If the packet is forwarded to an *external interface*, then the execution **ends** (at **external interface**).

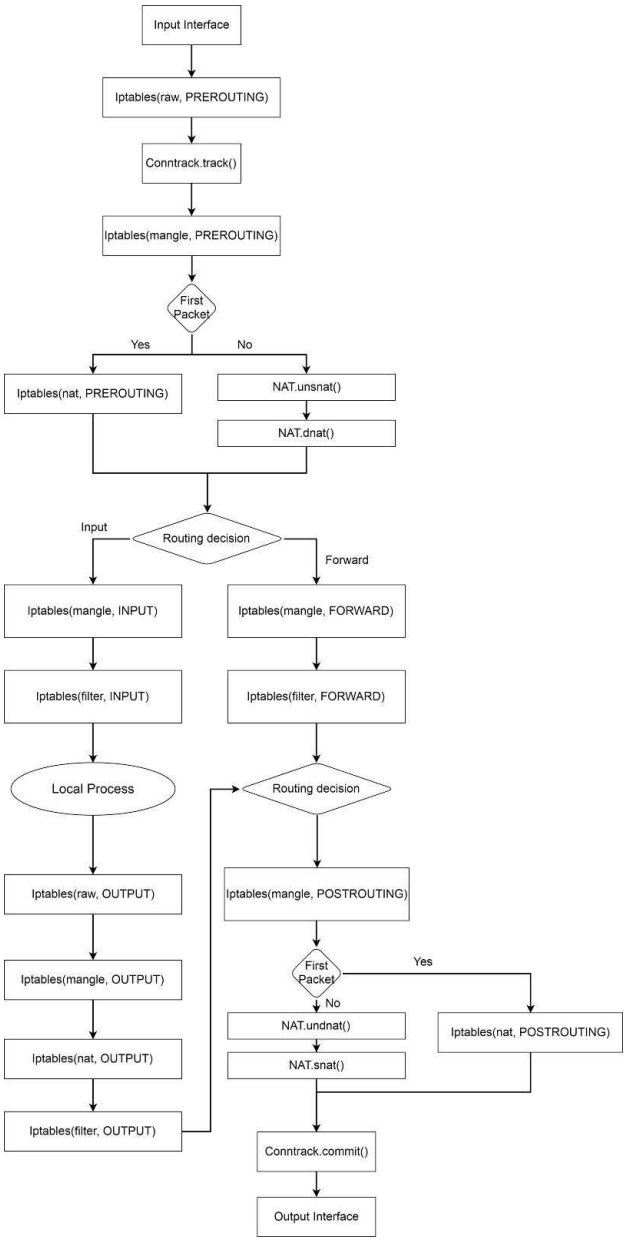
The execution **ends** whenever the packet has reached either an **external interface**, a **namespace** or a **VM**. The enumerated outcomes are called *final ports*. In general, for describing two-way traffic between different components in the system, the *symbolic packets* arriving at any of the ports are mirrored (i.e. L2, L3, L4 destinations and sources are reversed) and resent in the system from that point.

The steps depicted above present the normal forwarding (switching/routing) actions undertaken in a Linux-based Neutron deployment for a packet originated at VM level and heading outbound. Similar actions are required for other implementations.

In order to structure the packet processing flow, one can derive two basic modules which underly a Neutron deployment: a switching and a routing module. In the experiments under consideration, the switching component is comprised of multiple OVS switches, while the routing component is comprised of multiple Linux namespaces, handling IP routes and iptables rules augmented by connection tracking logic and NAT components.

The modules interact at: the interfaces at which packets enter, leave or transit the system and the connection tracking component (conntrack). Conntrack is used in because it integrates easily for creating stateful firewall processing. In Neutron, firewalling (i.e. security groups) may be obtained using either iptables or OVS drivers. For the environment under test, the OVS driver was chosen.

In principle, a packet-processing component (switching or routing) is composed a set of steps, by which a packet from an input interface is transformed, a set of component-specific variables are set and the packet is either sent out on another interface or dropped, as can be easily observed in the diagram below:
An interesting aspect regarding the interaction between components inside a Linux machine was the integration of the conntrack module specifications as per netfilter (in iptables) and those of the OVS conntrack module. To avoid limitations, the modeling process was tailored to handle both specifications.



Testing

In the following section, the results achieved testing the implementation depicted in Chapter 3 are presented. The tested scenarios, correctness of the outcomes and performance metrics are analyzed.

The experimental setup deployed in order to test the implemented features consists of two machines connected via 2 switches. The first machine, the *controller*, handles all OpenStack service layer, as well as all Neutron L3 functions (the L3 agent, DHCP agent and metadata agent). The L3 agent is of most interest for the current investigation since it is responsible for performing L3 routing between subnets as well as for providing internet access and floating IP functionality to the machines.

The second machine, *compute1* is a compute node as per OpenStack terminology and runs the virtual machines required at tenant level. From a networking point of view, *compute1* runs 3 OVS switches which enable machine interconnection and layer 2 forwarding between *compute1* and *controller*. Furthermore, the integration bridge on *compute1* enforces security group rules.

A snapshot of the experimental setup can be observed below. The topology has three distinct networks. They offer the following functionalities: the *Provider Network* is an administrator-defined network which ensures Internet access and floating IPs to the machines, the *Service Network* is responsible for driving the control plane communication between components that make up the OpenStack deployment, while the *Spanning Network* provides switching between machines using some segmentation protocol (e.g. VXLAN, GRE). For simplicity, the topology in Figure 3.4, was modified such that the *Spanning* and *Service* Networks are one and the same. For the OpenStack software components, it makes no difference if the two networks are not physically separated. In Figure 3.4, the full picture of the deployment is given.

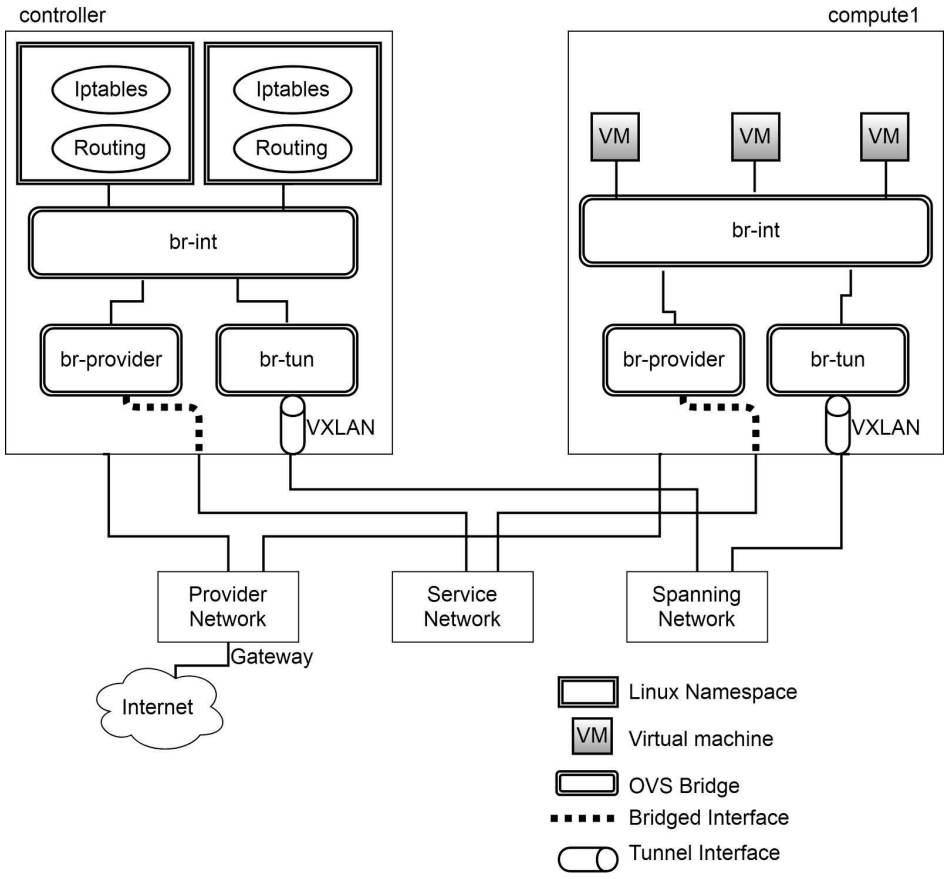


Figure: Physical experimental setup

Several tenant topologies were deployed. Two tenants were created, with two respective administrator users. For all the test cases, the two tenants will deploy a number of machines and administer networking components.

Test cases

In order to prove correctness of the modeling performed on the components and of their integration, three testing traffic patterns were considered:

1. North-south traffic via floating IPs. For this pattern, an external host attempts to reach an internal machine via floating IP. Packets flow in through the outbound interface at *controller* and are destined for a floating IP.
2. South-north traffic via SNAT. For this pattern, an internal host attempts to reach the Internet. Packets flow in through a VM interface at *br-int* in *compute1* and are destined for an external host at L3 level.
3. East-west traffic. For this pattern, an internal host attempts to reach another host on the same network. Packets flow in through a VM interface at the integration bridge in *compute1* and are destined for a host in its internally defined IP range at L3.

The traffic patterns considered are sufficient to demonstrate correctness of all modeled components in the topology. Furthermore, in subsection 3.4.2, a number of considerations are provided with respect to policy compliance of this deployment.

North-south traffic

The scenario under test uses a floating IP as packet destination and an external source IP address. In the following schema, the tenant settings **only** allow port 22 (SSH) TCP traffic inbound to the machine. The results obtained executing the *tenant* and *provider perspectives* were similar in that:

- From the *tenant perspective*, only one packet exited the port belonging to the floating IP machine, which was constrained to be an IP, TCP packet with destination port equal to 22 port.
- From the *provider perspective*, the only forward packets which arrived at the tap interface corresponding to the VM which was targeted were: IP, TCP, SSH packets and UDP packets with source port 67 and destination port 68. The latter belongs to the DHCP Discover protocol which is neglected at tenant-level.
- From the *tenant perspective*, return traffic was forwarded back to the initial destination going through the floating IP block and yielding a packet with destination TCP address constrained to 22.
- From the *provider perspective*, return traffic was also forwarded back to the initial destination and source TCP port was constrained to port 22.

Conclusion: In this experiment, end-to-end connectivity between an external host and an internal machine was proven with respect to the security policy defined to the machine (i.e. traffic filtering by TCP port).

South-north traffic

The testing scenario depicted in this paragraph shows compliance of the current system with Internet bound traffic from a local machine. The input packet was originated at a machine in the default security group with an external destination IP address. In the default security group, all outbound traffic from machines is allowed. The following outcomes were observed:

- All outbound traffic from a machine reached the external world from both *tenant* and *provider* perspectives.
- SNAT works correctly from both *tenant* and *provider* perspectives.
- All inbound traffic in reply to the initial packet was allowed back to the machine ⇒ reverse SNAT works from both *tenant* and *provider* perspectives

Conclusion: In this experiment, end-to-end connectivity between a machine and an outbound host was proven for all IP packets.

East-west traffic

The scenario presented throughout this paragraph shows that hosts on the same network have end-to-end connectivity, while at the same time shows tenant separation. The following outcomes were observed:

- Traffic issuing at some VM heading West reaches the DHCP Server and Router interfaces from both *tenant* and *provider* perspectives.
- Traffic between two machines in the same default security group is allowed.
- Traffic between two machines with the first in one security group and the second in another is not allowed (i.e. ingress packet policy is respected) from both *tenant* and *provider* perspectives.
- Traffic from a VM in one network is not reaching another tenant's networks in both *tenant* and *provider* perspectives.
- Reverse traffic is correctly sent back to the respective tenant from both *tenant* and *provider* perspectives.

Conclusion 1: In this experiment, end-to-end connectivity between machines running on the same network was proven for all IP packets.
Conclusion 2: In this experiment, in-tenant connectivity at Layer 3 was proven between different subnets - i.e. the L3 agent offers inter-subnet connectivity.
Conclusion 3: In this experiment, tenant separation was proven.

Results

In this section, results from multiple symbolic executions are analyzed with respect to their runtime performance. Since proving correctness of a given deployment must occur early in the installation phase and changes to the topology may occur frequently, the importance of quickly proving or disproving the properties of a given system is decisive. Thus, the results presented below take into account time as a metric for the performance of a symbolic analysis.
As per Symnet's semantics, all fork instructions yield 2 distinct and independent execution paths. With that in mind, it is obvious that *the state explosion* will occur for chained fork instructions. It is interesting to note that theoretically, the amount of paths which a symbolic packet transits is exponential in the amount of Fork instructions. However, some states will be quickly trimmed due to non-compliance to some constraints. The amount of concurrent states active at some moment is proportional to the amount of unconstrained symbolic values of the input packet. If the packet is made concrete from the beginning of the execution, the amount of concurrently existing states will drop dramatically and so will the time to execute the topology.

The experiments were performed on two distinct datasets containing information related to the state of the deployment. The *small* dataset is composed of a series of 121 configuration files, while the *large* dataset consists of a series of 134 files covering a larger number of user scenarios.
The scenario under test for the *small* dataset contains 2 virtual machines in different security groups attached to a self-service network and one router. The *deployment perspective* configuration set is composed: 151 *iptables rules* in all routing namespaces at the network node and 157 flows in all OVS bridges in the deployment.
For the *large* dataset, four more machines, a new network and a router interface were added. The *deployment perspective* configuration set is composed of 153 *iptables rules* in all routing namespaces at the network node and 275 flows in all OVS bridges in the deployment.
The most time-consuming component, as far as symbolic execution is concerned is the OpenFlow table of the integration bridge at node *compute1*. This conclusion was drawn by analyzing a set of profiling information regarding the execution time of multiple components in the system.

Tenant perspective

From the tenant perspective, all measurements which were taken indicate small processing times, even for the largest configuration sets. The greatest execution time observed in the system amounts to 5 seconds. The dependency of the execution time on the number and semantics of unconstrained packet variables is not as influential as for the *provider perspective*. In fact, the processing model for the *tenant perspective* implies explicitly cutting down illegal Ethernet and IP source and destination addresses (as per Security Policy and Address Spoofing specifications) immediately after packet insertion in the system. The number of issuing states is smaller in comparison to those in the *provider perspective*.

Provider perspective

The measurements acquired for each experiment reside in: time to parse the configuration files, forward execution time (i.e. the time for a packet to reach destination one-way) and backward execution time (i.e. the time for a reply packet to reach the initial source). To simplify the tables below, the time to parse the configuration files using ANTLR4 and a series of regular expression matchers takes on average 4.4s on the *large* configuration set and 0.5s on the *small* one.
The following tests are aimed to show the execution times of a East-west traffic pattern issuing from one VM and heading West to a known host. In this example, the known host represents a known Ethernet Destination. The statistics acquired for the current experiment are:

OpenFlow Flows at br-int compute1	215
Forward runtime	21.3s
Forward success states	6
Forward failed states	2

Backward runtime	28.2s
Backward success states	9
Backward failed states	37

Table 1: *Large* dataset: East-west traffic with Ethernet Destination bound

In the following example, a similar packet is issued at the same interface, with the only difference being that the Ethernet Destination is symbolic at input and the destination IP address is assumed constant at input. This change modifies significantly the time of execution, since in the first OVS OpenFlow table, there is no reference to the destination IP, but rather to the Ethernet Destination.

OpenFlow Flows at br-int compute1	215
Forward runtime	153.4s
Forward success states	27
Forward failed states	104
Backward runtime	222.6s
Backward success states	27
Backward failed states	68

Table 2: *Large* dataset: East-west traffic with Ethernet Destination unconstrained and known IP destination

As can be easily observed from comparing the results in tables 1 and 2, in the second case, the experiment runs roughly 10 times slower than in the former due to the exhaustive search it must perform inside the OpenFlow table at node *compute1*. Furthermore, to underline the weight of the number of flows in the first integration bridge, the following experiment was considered. Let a packet issuing some node be destined outbound to an external address. Two outcomes are discussed based on their respective dimension in terms of OpenFlow entries in the br-int table, as per table 3.3.

	Number of OpenFlow tables	
Metric	215	107
# OK States	27	15

# Failed States	113	97
Execution time (s)	67.5	39.3

Table 3: Time vs. number of OpenFlow flows for outbound traffic

From table 3, it can be easily observed that the dependency between time of execution and the number of OpenFlow Flows in the integration bridge at *compute1* is close to linear. However, in order to compare the penalties incurred by the increasing amount of configuration units in the system, a completely unconstrained input packet from some *port* in the system was sent. The results are compared for the two datasets:

	Small dataset	*Large* dataset
Forward time (s)	335.8	666.3
Backward time (s)	125	539

Table 4: Unbound packet analysis

The preliminary observation regarding the results in table 4 is that, even though execution time is large, its increase does not appear to be exponential in the number of configuration units, but more rather linear in this respect.

Deployment correctness

While testing and deploying the Symnet-Neutron integration, two important results were shown, stressing the importance of early verification within a cloud infrastructure.
Because of a mis-configuration at *compute1* level, several OpenFlow flows were not successfully installed on the node. Thus, even though execution from the *tenant perspective* suggested that no packets were allowed from a machine to another, the execution from the *provider perspective* showed that traffic was allowed between the two. In reality, network diagnostics showed that the installed data plane configurations were not compliant to the tenant's intention.
Also, verification came to the rescue of the *tenant network administrator* at the moment where a machine was accidentally inserted in a different security group, while keeping the second in the default group. No connectivity between the two machines was available. A *tenant perspective* execution showed that the specified packet was being trimmed precisely at ingress security group level for the destination machine.

Notes on memory usage

The solution presented in this thesis aims to symbolically execute a complex network topology containing a lot of atomic instructions. As such, the expected memory consumption is large and growing exponentially with the number of *Fork* instructions encountered.
The observed behavior is that, indeed, the memory footprint of a Symnet execution is large and may incur hidden penalties, such as important garbage collection overheads (which

may, in turn, incur time penalties upon the execution engine). For the most unfavorable case, a memory footprint of roughly 8 GB was observed.

4. Optimizing match-action table verification

Due to recent advancements in switch design and architecture [0] that enable Match-Action Tables to be implemented in hardware, they are one of the driving forces for deploying Software Defined Networking(SDN) technologies. This paradigm of packet processing can be found in devices ranging from the traditional switches and routers - their forwarding information base (FIB) being represented in this way - to the more novel devices that allow custom packet forwarding(SDN - enabled) such as OpenFlow[*], P4[*], Open Packet Processor[*].

A MAT-instance consists of a number of table entries, each composed of a number of conditions and a number of actions. If a table contains a single entry, the set of conditions stated by this entry is checked against every packet that is received for processing. If the conditions hold, it is said that the packet matched the conditions and is next going to be processed according to the actions corresponding to this table entry. Otherwise, if the conditions did not hold, the packet is going to be processed according to a default set of actions, specific to every MAT-instance. In the case tables consisting of multiple entries, the packet is matched against every condition set(one per table entry). If the packet matched no entry, then the default action-set is applied. If only one entry matched, then processing happens in the same manner as with the single-entry table. If more than one entry were matched, then a tie-breaking strategy is required for ensuring processing by only one table entry. Some of the more common tie-breaking strategies are: setting a matching priority and then choosing the one with the highest priority or, in the case of routing FIBs, choosing the entry with the longest prefix(longest-prefix matching).

Modelling MATs for verification purposes

From a verification perspective, the task of deciding *what set* - or to put it slightly differently, *if a set of packets* gets processed by a given table entry is a fundamental primitive of any verification algorithm. The complexity of this task lies not in checking if a set of packets matches a set of conditions corresponding to a single table entry(which is bound in size, usually by a constant factor due to hardware limitations), but in determining what subset did not match any set of match conditions stated by table entry of higher priority. In other words, for a packet to be processed according to a certain table entry two conditions must hold: the packet must match the conditions of this table entry and it must not match any condition set stated by an entry of higher priority. As an example, for a packet to be forwarded by a router according to the default route (‘0\0’), no other forwarding rule should be applicable.

In technical terms, for a table entry TE, consisting of a condition set S and an action set, to be applied S must hold for the packet P and, for every entry of higher priority TE_1, TE_2, ... , TE_n defining the sets of conditions S_1, S_2, … , S_n, the negated sets of conditions $\overline{S_1}$, $\overline{S_2}$, … , $\overline{S_n}$ are constructed and their conjunction must also hold. Taking a look back at our

previous example of packet routing, applying the previous observation in the case of the default route would result in a number of negated conditions that is linear in the number of table entries. Furthermore, from a table-wide perspective, the total number of conditions that must be verified is quadratic in the number of entries (table size). This looks very discouraging given the commonality of FIBs with sizes in the hundreds of thousands.

To make matters worse, current MAT implementations(OpenFlow, P4, etc) support matching on multiple fields at once (in the same table entry). Even the way matching can be specified evolved, ranging from exact matching to more complex matching schemes, such as: bit masks, longest-prefix matching, ranges etc.

Another constraint is that any algorithm we might came up with must work well in the dynamic context in which data planes operate in production. Whenever rules are added or deleted, the computation of the model that considers this update should be performed at the same timescale.

Problem formulation

In this section we will give a concrete formulation of the problem.

As previously stated, a table entry *TE* is defined by two components a set of matching constraints *S* a packet must satisfy in order to be processed by the actions defined for that particular TE - which represent its second component. We will focus for the rest of this writing on just the first part of a *TE*.

The size of a constraint set *S* is dictated by the number of match conditions defined by the structure of the MAT. For instance if the MAT instance is a routing FIB, then the constraint set has only just one member - a longest-prefix match condition imposed on the IP destination field; another example would be a table that performs filtering based on the value of the destination port and transport protocol number - in which case the constraint set would be of size two, denoting two equality conditions imposed on destination port and protocol number. We will denote this by $S_n = \{C_{n1}, C_{n2}, \ldots, C_{nm}\}$; in which C_{xy} denotes the *y*th condition belonging to the *x*th *TE* in the MAT; having the property that for each $j \in \{1, 2, ..., m\}$ and any *x*, conditions C_{xj} affect the same field. This last property states that each *TE* should respect the MAT structure - the field being matched are consistent across *TE*s.

We define an overlap as the case in which for two *TE*s: TE_x and TE_y there is at least one pair of conditions $C_{xj}\ and\ C_{yj}$ that can hold for the same packet.

For any such pair of *TE*s, the less prioritary one has to be augmented such that there will be no overlap. The trivial solution for this, building a new condition set $S_{x'} = \{C_{x1} \wedge \neg C_{y1}, C_{x2} \wedge \neg C_{y2}, \ldots, C_{xm} \wedge \neg C_{ym}\}$, would clearly yield a valid $S_{x'}$ in the sense that there are no more overlapping conditions, clearly. The problem is one mentioned before that it would scale poorly in the number of extra conditions that are to be added - yielding a heavy toll on

the constraint solving procedure. Thus the goal is to build an algorithm that would yield a constraint set $S_{x\,min}$ that is minimal in the number of extra conditions added to mitigate every overlap, but would work at time scales comparable to those at which MAT updates appear in practice.

We have built on the body of research already existing [1,2] on this subject by taking a step back and targeting a broader solution that would be applicable in a wide range of possible matching methodologies (lpm, range matching, wildcards etc) - that c
an easily incorporate specific optimizations (given a specific matching method) without requiring extensive modifications.

Our solution

We have started with the ADT represented by a forest of multiple-node trees. For every condition type, there will be one condition C_{xj} per table entry TE_x. For this set of conditions that share the same type (the same matching procedure applied to the same packet header) $\{C_{1j,}\ C_{2j}\ \ldots\ C_{nj}\}$; n representing the size of the table (number of table entries) we will build one such forest. In this forest, every node will correspond to one condition C_{xj}.

In this data structure there can be four types of relations between the nodes:

- The nodes are completely independent if their corresponding conditions C_{xj} and C_{yj} do not overlap
- Parent-of - this representing a down-link in a tree between a parent node and a child node if the condition corresponding to the child node denotes a field space entirely contained by that denote by the condition of the parent (the parent's condition is more general and includes the child's condition)
- Ancestor-of - this representing a relation between two nodes that are connected in the same tree by several 'parent-of' links
- Neighbor-of - connecting two nodes that correspond to overlapping conditions, but neither condition is fully contained by the other; and no node already has an ancestor node linked to the other node via a 'neighbor-of' link. In other words, if there is a 'Neighbor-of' relation between two nodes, then there will be no other such link between descendant nodes of these two.

Having presented the definition of the ADT, for every condition C_{xj} in order determine the simplest condition that would match condition C_{xj} but no other overlapping condition, one builds the set of conditions C as the union between the set of conditions corresponding to 'child-of' and 'neighbor-of' nodes and then negate every condition in this set. Note that, given the structure of the tree this operation has Θ(|C|) complexity.

The optimality of this algorithm in terms of the number of conditions added to mitigate the overlap comes from the fact that if a node is selected ('child-of' or 'neighbor-of') then there is no need to consider any of its descendants, since their conditions are fully covered by the one of the selected ancestor, thus rendering them redundant.

We implemented the forest construction algorithm in its most general form, that can handle range-matching, longest-prefix matching, wildcard matching etc. with no modification - given there is an implementation to test the relation between two nodes. Even without specific optimizations that can be made given the type of matching, we saw an improvement in the runtime of the model-construction algorithm as compared to the default implementation in Symnet, and also a massive reduction (2x) in the number of constraints being generated. We evaluated the implementation using FIBs taken from Stanford backbone router - having more than 180 000 entries.

Table size	Forest height	Average time to build model for one TE	Number of constraints
183 369	6	5ms	298 112
62 396	6	1.45ms	96 189
1 815	4	0.1ms	2 791

We can conclude that even for very large scale tables, the cost to construct the model corresponding to a new insert is still under 10ms, which makes it feasible to update the model at the pace at which MAT updates can occur.

[0]: Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. SIGCOMM Comput. Commun. Rev. 43, 4 (August 2013), 99-110. DOI: http://dx.doi.org/10.1145/2534169.2486011

[1] A NICE Way to Test OpenFlow Applications - Marco Canini, Daniele Venzano, Peter Perešíni, and Dejan Kostić, EPFL; Jennifer Rexford, Princeton University; NSDI ‘12

[2]:VeriFlow: Verifying Network-Wide Invariants in Real Time - Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey, University of Illinois at Urbana-Champaign, NSDI ‘13

5. Anomaly detection

In a previous report, we proposed a novel universal anomaly detection algorithm, which was able to learn the normal behavior of systems and alert for abnormalities, without any prior knowledge on the system model, nor any knowledge on the characteristics of the attack. The suggested method utilized the Lempel-Ziv universal compression algorithm in order to

optimally give probability assignments for normal behavior (during learning), then estimate the likelihood of new data (during operation) and classify it accordingly.

In this report, we evaluate the algorithm on real-world data and network traces, showing how a universal, low complexity identification system can be built, with high detection rates and low false-alarm probabilities. We first apply the detection algorithms to the problems of malicious tools detection via system calls monitoring and data leakage identification. We then give some results for detecting anomalous HTTP traffic, e.g., Command and Control channels of Botnets.

Detection of Malicious Tools and Data Leakage

We apply the anomaly detection system suggested to system calls in order to detect malicious tools on a Windows machine, and to TCP traffic of a server in order to detect unwanted data leakage. In both experiments, the capability of the tool to detect abnormal behavior without prior knowledge is demonstrated.

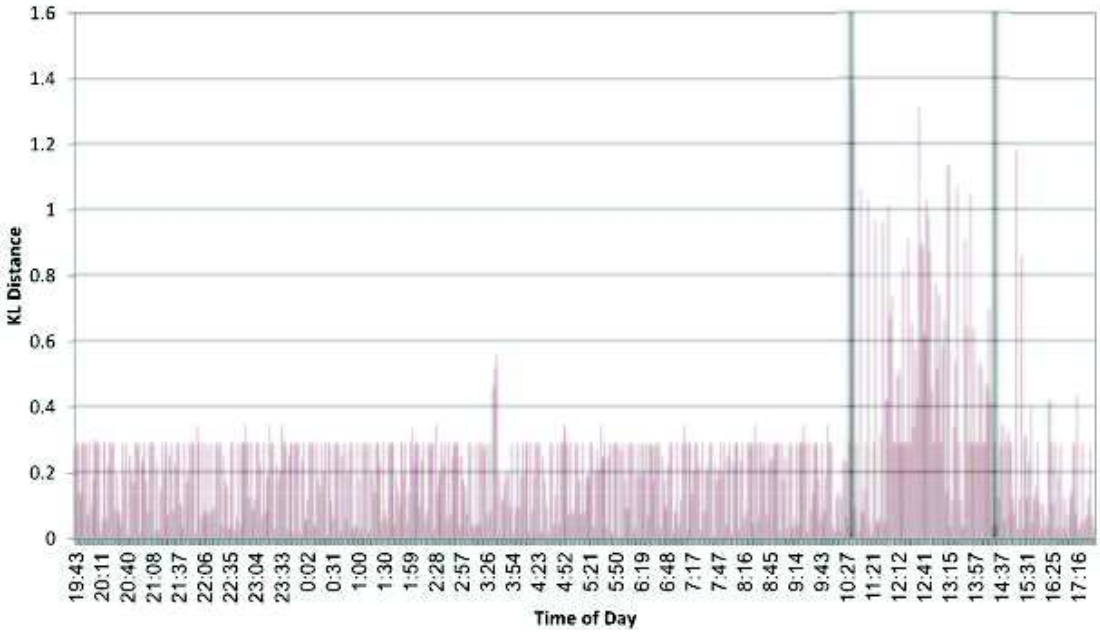
Monitoring the Context of System Calls for Anomalous Behavior

The sequence of systems calls used by a process can serve as an identifier for the process behavior and use of resources. Moreover, when a program is exploited or malicious tools are running, the sequence of system calls may differ significantly compared to normal behavior, incriminating the program or entire machine. The universal anomaly detection tool was used to learn the *context of normal system calls*, and alert for anomalous behavior. Specifically, the sequences of system calls created by a process (e.g., firefox.exe) were recorder, processed, and learned. Then, when viewing new data from the same process, the anomaly detection algorithm compared the processed new data to the learned model in order to decide whether the process is still benign, or was it maliciously exploited by some tool.

Due to the large amount of possible system calls, calls were grouped into 7 types, based on the nature of the call: *Device, Files, Memory, Process, Registry, Security and Synchronization*. That is, the quantization process did not include any minimization of distances or a requirement for uniform probabilities, but, rather, labeled the calls based on their known functionality. Recording and classification used NtTrace.

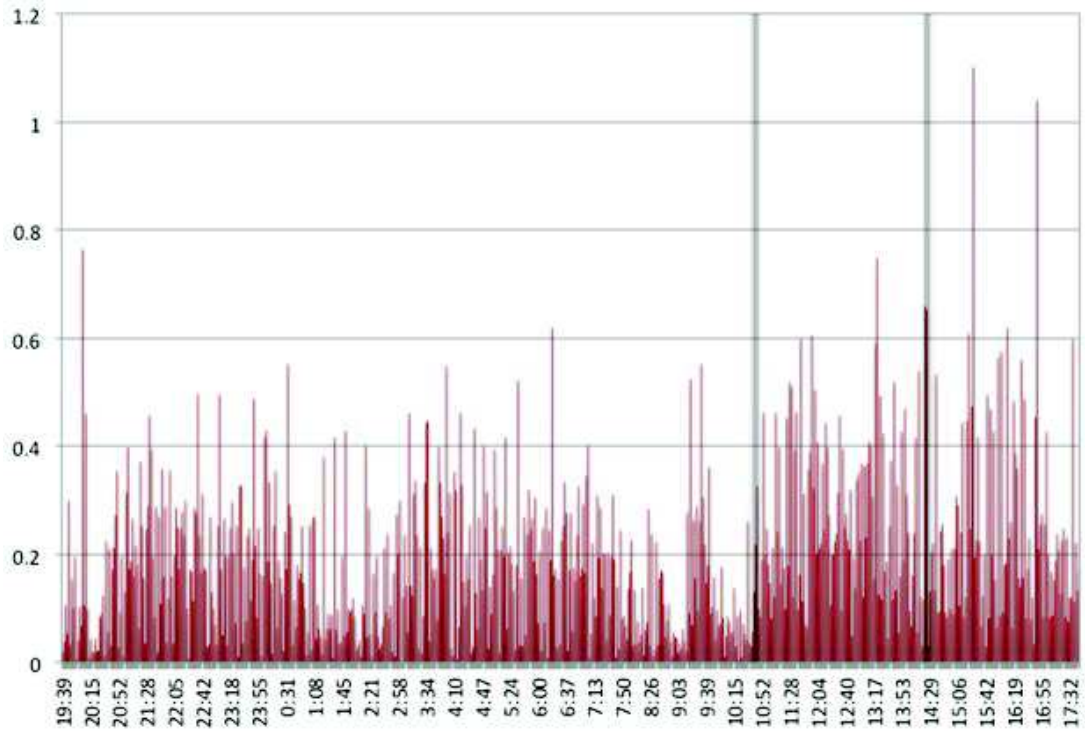
In the learning phase, system calls were recorder, quantized according to the types above and then a discrete sequence over the alphabet of size 7 was created. The sequence was used to build the (normal behavior) LZ tree, as described in the previous report, from which a histogram for the probabilities of tuples of length 20 was calculated. This histogram was *the only data saved from the learning phase*. The learning phase included 4 hours of data.

For testing, segments of 2 minutes were recorded. For each segment, a histogram was calculated, similar to the learning phase (calculating probabilities for tuples of length 20 over an alphabet of size 7). Decisions were made based on the Kullback-Leibler divergence between the normal histogram and the tested one.



The figure above plots the KL distances between the histogram during the learning phase, and the histograms extracted during the testing phase. *The process tested was firefox.exe, and the two vertical thick lines mark the time when the tool "Zeus" was active*. It is very clear that the context of the system calls changes dramatically when the tool is active, and that simple monitoring of the KL distances every few minutes is sufficient to detect a change in the system behavior. Note, however, that the specific tool used, Zeus, installs instances in several processes simultaneously, hence might not be as active within all processes.

The next figure below depicts the same setting, yet with winword.exe. It is clear Zeus is harder to identify within that process.



It is clear Zeus is harder to identify within that process. Nevertheless, from both figures, it is clear the anomaly detection algorithm suggested, when it monitors a few processes on the machine, can easily identify malicious behavior *with minimal delay and complexity*.

Identifying Data Leakage

In this part of the work, the universal anomaly detection algorithm was used in order to identify data leakage from a web server. Specifically, the setting was as follows. In the learning phase (a period of a few days), benign traffic on a web server was recorded using Wireshark. Similar to the previous examples, timing-based sequences where extracted, quantized and used in order to build an LZ tree. This LZ tree served as a model for normal data.

Then, using Ncat, a script was installed on the server. This script initiated downloads of large chunks of data from the server. Several periods, each 30 minutes long, of traffic which includes Ncat were recorded. For comparison, similar length periods of traffic without Ncat were recorded as well. An LZ tree was built for *each of the 30 minutes datasets*.

To identify data leakage, unlike the Botnets setting considered in Section V, in this case, we compared the *joint distributions of* k-*tuples* resulting from the LZ trees. That is, we used the distribution of k-tuples resulting from the LZ tree as an identifier for the data set, and calculated the distances between the distributions.

	Ncat1	Ncat2	Normal1	Normal2	Normal3	Normal4	Normal5	Normal6
MSE	0.962	1.262	0.044	0.153	0.143	0.43	0.142	0.017
KL	2.05	17.163	1.353	1.228	2.026	4.12	2.121	1.396

The table depicts the distances between the learned, normal data, and 8 testing periods, two which include data leakage using Ncat and 6 without. Two distance measures were used in this part of the work: Mean Square Error (MSE) and KL distance. Under MSE, the leakage sessions clearly stand out compared to normal data. Results under the KL distance are less clear, especially in the first Ncast session, which included more normal data than the second.

Finally, to further challenge the algorithm, and see whether data leakage will also stand out when the normal communication includes (peaceful) massive downloads, the normal communication was augmented with benign downloads of various sizes. The table below depicts the results (under the KL distance). It is clear that while Ncat stands out compared to normal traffic on the web server, it is almost indistinguishable when the normal *traffic learned includes downloads of large files*. This is expected, as Ncat uses a similar protocol, and the key differences in the timing are caused by file sizes. Hence, data leakage is clearly detected compared to normal surfing, yet, it is indistinguishable when the server, in peaceful times, serves large downloads.

	Normal	Normal + 1.3MB	Normal + 10MB	Normal + 200MB
Normal	0.906	0.843	0.583	0.72
Ncat	19.05	0.787	0.733	0.353

Identifying Anomalous Traffic

In this part of the experiment, we used the algorithm to identify anomalous traffic.
Specifically, traffic between sets of hosts and clients was tested, and the goal was to identify anomalous traffic, e.g., HTTP traffic used as command and control of Botnets, etc.
Each client, denoted by 'cid', may connect with several hosts (web-servers), denoted by 'hid'. On each transaction, data is sent both by the client and the host. This defines communication pairs CID HID. Each transaction is labeled as either legal, denoted by 'good', for normal data traffic generated by the client, or illegal, denoted by 'hostile', that is, Bot traffic. Labeling was done by the security company's experts based on well-known black-lists. Note that these labels are not used during the classification process. They are used only in the validation phase.

Each transaction is represented by a single record in the data set, which consists of the following fields: 'time', referring to the time the transaction took place; 'time-taken', is the total time the transaction took; 'cs-bytes' and 'sc-bytes' fields represent the total bytes sent by the client/server(host) to the server(host)/client during the transaction, respectively; 'mime-type' denotes the Internet content type of the transaction, such as: plain text, image, html page, application, etc.; 'cat' is the category of the transaction - 'good' or 'hostile'; and the 'hid' and 'cid' fields refer to the host-index (Internet site, web- server) and client-index respectively. Indices were given arbitrary to protect the identity of the hosts. However, some malicious sites are identified by their domain name, e.g., 'hotsearchworld.com' or 'blitzkrieg88.bl.funpic.de'.

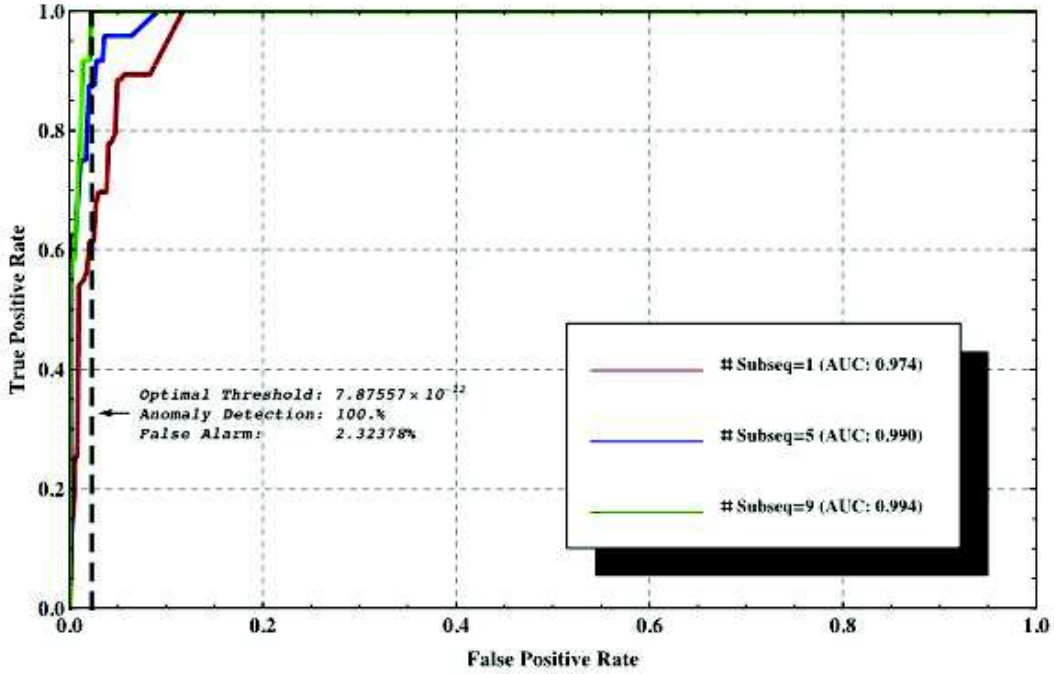
Processing of the data included serialization and feature extraction: first, the given data set is split into set of flows based on CID HID connections. A 'Flow' is a sequence of related transactions of the same communication pair CID HID sorted by time and with the same

label, either ‘good’ or ‘hostile’. In total, there are 19164 flows labeled as ‘good’ and only 65 ‘hostile’ flows (0.338%). This indicates the imbalance of the data set where most of the transactions are legal and only a small fraction is illegal. However, this is characteristic of real network traffic behaviour.

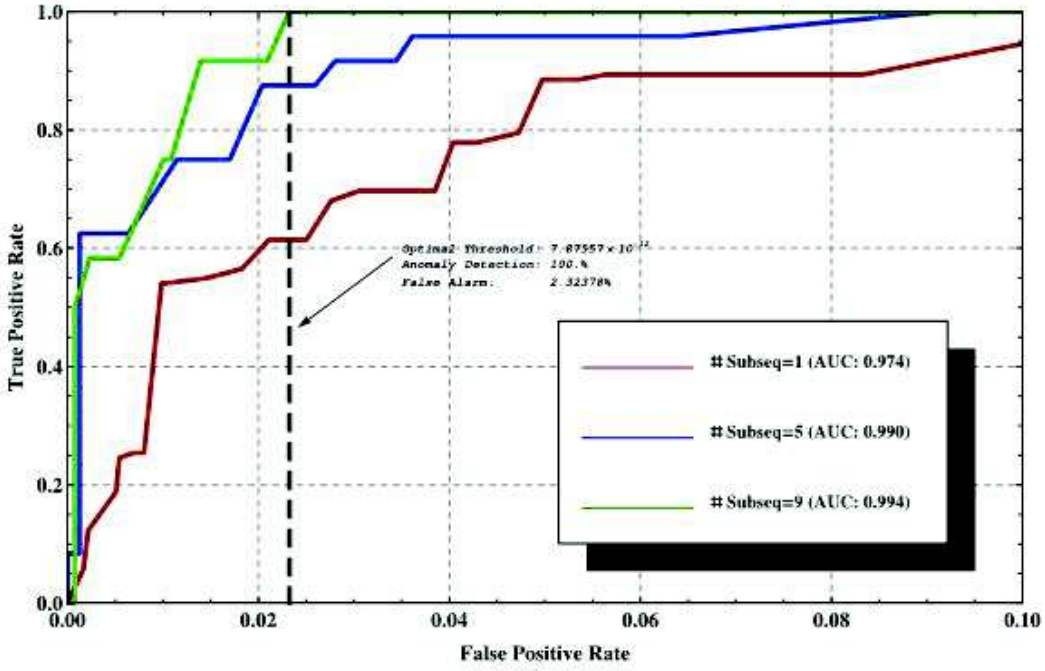
Next, selected features are extracted from each transaction , e.g., Time-Difference, Time-Taken, Server-Client-Bytes and Client-Server-Bytes. After quantization, the resulting sequences are the discrete time, finite alphabet sequence on which learning and testing was performed.

The best results, in terms of optimal threshold and ROC-AUC (Area Under Curve), were achieved using the Time-Difference (TD) representation of the data sequences along with the ‘Uniform’ quantization (several quantization algorithms were tested). To better understand why TD was superior, consider a *legitimate web surfer compared to a hostile connection using HTTP only as a C&C channel*. While the surfer must have a reasonable behaviour in the time domain, affected by the times required to read a page, the times required for the server to respond, etc., a C&C channel may behave differently, without, for example, a reasonable response time from the server as it only collects data from the bots, and the “GET” messages are used solely to *transmit* information. Due to space limitation, we do not include the results for the inferior features, and focus on the results under TD and uniform quantization.

Still, using TD, the optimal threshold for 100% detection results in 11.75% false alarms. However, *this is when only a single, short sequence is tested*. To further improve the above results, a majority vote for several sequences within the flow can be used. Each data segment is partitioned into several subsequences of length 10. The classification is done based on the majority of these subsequences’ estimations, as either positive or negative, resulting in better classification performance as the number of subsequences is higher. For example, an AUC of 0.994 and false alarms rate of 2.32378% are achieved using a threshold of 7.87557x10−12, as illustrated in the figure below.



A zoom on the relevant part of the figure is also available:



6. Conclusions

This internal deliverable documents progress Superfluidity has achieved in WP6. The common goal of the presented work is to increase the security and robustness of the network, either via pro-active policy-driven verification (with Symnet and applied to Openstack, P4/Openflow/iptables) or reactive anomaly detection.