A Firewall Algebra for OpenStack

Simon N. Foley and Ultan Neville,
Department of Computer Science,
University College Cork, Ireland

***Abstract*—An algebra is proposed for constructing and reasoning about anomaly-free firewall policies. Based on the notion of refinement as safe replacement, the algebra provides operators for sequential composition, union and intersection of policies. The algebra is used to provide a uniform way to specify and reason about OpenStack host-based and network access controls, in particular, security group and perimeter firewall policies.**

I. Introduction

The cloud computing paradigm has become widely adopted, with applications ranging from research to enterprise. However, for end-users, administrators and providers alike, there are security challenges. Managing the host-based and network access controls within and across cloud deployments is complex and error-prone. Cross-tenant accesses are often necessary for service-to-service communication, and the environment is highly dynamic due to platform and service migration. Multiple access control policies of varying types are required for a deployment, and a misconfigured policy may permit accesses that were intended to be denied or vice-versa. We regard the specification of an access control policy as a process that evolves. Threats to, and access requirements for, resources within a cloud do not usually remain static, and over time, a policy or distributed policy configuration may be updated on an ad-hoc basis possibly by multiple specifiers/administrators. This can be problematic and may introduce conflicts, whereby the intended semantics of the specified access controls become ambiguous.

In this paper, we present a firewall policy algebra for constructing and reasoning over anomaly-free policies. The algebra allows a policy specifier to compose policies in such a way that the result of the composition upholds the access requirements of each policy involved, and allows one to reason as to whether some policy is a safe (secure) replacement for another policy in the sense of [7], [8], [12].

The proposed algebra is used to reason about OpenStack cloud deployments. OpenStack [1] is an open-source cloud operating system. It provides Infrastructure-as-a-Service (IaaS) through a collection of interrelated projects/services. These services are used to manage pools of compute, storage, and networking resources throughout a datacenter [1]. This paper focusses on the OpenStack *Neutron* networking service.

The primary contribution of this paper is an algebra that can be used to explore different kinds of firewall policy in a distributed configuration. The paper is organised as follows. Section II presents an algebra $\mathcal{FW}_0$ for OpenStack host-based and network access control policy configuration. In Section III, OpenStack perimeter firewalls and security groups are encoded in the algebra. Section IV presents a case study OpenStack deployment that illustrates the practical use of the algebra. Section V outlines an extension to the algebra. Related work is outlined in Section VI and Section VII concludes the paper.

The Z notation [15] is used to provide a consistent syntax for structuring and presenting the definitions and examples in this paper. We use only those parts of Z that can be intuitively understood and Appendix A gives a brief overview of the notation used. Mathematical definitions have been syntax- and type-checked using the *f*UZZ tool [14].

II. $\mathcal{FW}_0$ The basic firewall algebra

In this section we present an algebra for constructing and reasoning about anomaly-free firewall policies. For the purposes of this paper we focus on stateless firewall policies that are defined in terms of constraints on individual IP addresses, ports, and protocols. Support for other features such as IP/port ranges, TCP flags and stateful filtering is provided by an extended algebra $\mathcal{FW}_1$ [9] which we do not consider in this paper for reasons of space, but discuss its application in Section V.

Let *IP*, *PORT* and *PROTO* define the sets of IP addresses, ports and protocols, respectively.

$$[IP, PORT, PROTO]$$

For simplicity, we do not consider how the values of these types may be constructed, other than to assume that the usual human-readable notation can be used, such as 1.2.3.4 for IP addresses, natural numbers for ports, and literals tcp and udp for protocols TCP and UDP, respectively. For ICMP, literals include valid ICMP Type/Code combinations, such as icmp8/0 $\in$ *PROTO* and icmp17/0 $\in$ *PROTO*, with icmpAll $\in$ *PROTO* representing an instance of ICMP for all valid ICMP Type/Code combinations.

A network packet header is a five-tuple (s, *sprt*, d, *dprt*, p), representing network traffic originating from source IP address s, with source port number *sprt*, destined for destination IP address d, with destination port number *dprt*, using protocol p. Let *Packet* define the set of all packet headers:

$$Packet == IP \times PORT \times IP \times PORT \times PROTO$$

A firewall policy defines the packets that may be allowed or denied by a firewall. Let *Policy* define the set of all firewall policies, whereby

$$Policy == \{A, D : \mathbb{P}\, Packet \mid A \cap D = \emptyset\}$$

A firewall policy $(A, D) \in Policy$ defines that a packet $p \in A$ should be allowed by the firewall, while a packet $p \in D$ should be denied by the firewall. Given $(A, D) \in Policy$ then A and D are disjoint: this avoids any contradiction in deciding whether a packet should be allowed or dropped. Thus, *Policy* defines the set of *anomaly-free* policies in the sense that they contain no redundancy, shadowing, or other anomalies [5].

Consider the following network security policy which we will refer to throughout this section. Client IP address 192.168.2.5 is permitted access to a HTTP server at 172.16.1.6 and to a Git server at 172.16.1.5, and all Telnet traffic to the HTTP server is to be denied. The packets that are used to define this policy include http_1, git_1 and tel_1, where

$$\mathsf{http}_1 == (192.168.2.5, 1025, 172.16.1.6, 80, \mathsf{tcp})$$
$$\mathsf{git}_1 == (192.168.2.5, 1025, 172.16.1.5, 9418, \mathsf{tcp})$$
$$\mathsf{tel}_1 == (192.168.2.5, 1025, 172.16.1.6, 21, \mathsf{tcp})$$

A firewall policy $\mathsf{Admin}_1 \in Policy$ allows the HTTP and Git packets while denying the Telnet traffic:

$$\mathsf{Admin}_1 == (\{\mathsf{http}_1, \mathsf{git}_1\}, \{\mathsf{tel}_1\})$$

Note that $(A, D) \in Policy$ need not partition *Packet*: the allow and deny sets define the packets to which the policy *explicitly* applies and an *implicit* default decision is applied for those packets in $Packet \setminus (A \cup D)$. For the purposes of modeling OpenStack firewalls it is sufficient to assume *default deny*, though we observe that the $\mathcal{FW}_0$ can also be used to reason about default allow firewall policies.

The policy accessor functions *allows* and *denies* are analogous to functions *first* and *second* for ordered pairs:

$$\begin{array}{l} allows, \\ denies : Policy \to \mathbb{P}\, Packet \\ \hline \forall\, a, d : \mathbb{P}\, Packet \bullet \\ \quad allows\,(a, d) = a \land \\ \quad denies\,(a, d) = d \end{array}$$

Thus, we have for all $P : Policy$ then $P = (allows(P), denies(P))$.

a) Policy Refinement: An ordering can be defined over firewall policies, whereby given $P, Q \in Policy$ then $P \sqsubseteq Q$ means that P is no less restrictive than Q, that is, any packet that is denied by Q is denied by P. Intuitively, policy P is considered to be a *safe replacement* for policy Q, in the sense of [7], [8], [12] and any firewall that enforces policy Q can be reconfigured to enforce policy P without any loss of security. The set *Policy* forms a lattice under the safe replacement ordering and is defined as follows.

$$\begin{array}{l} \mathcal{FW}_0 \\ \hline \bot, \top : Policy \\ \_ \sqsubseteq \_ : Policy \leftrightarrow Policy \\ \_ \sqcap \_, \_ \sqcup \_ : Policy \times Policy \to Policy \\ \hline \bot = (\emptyset, Packet) \land \top = (Packet, \emptyset) \\ \forall\, P, Q : Policy \bullet \\ \quad P \sqsubseteq Q \Leftrightarrow (allows\,(P) \subseteq allows\,(Q)) \land \\ \qquad\qquad (denies\,(P) \supseteq denies\,(Q)) \land \\ \quad P \sqcap Q = (allows\,(P) \cap allows\,(Q), \\ \qquad\qquad denies\,(P) \cup denies\,(Q)) \land \\ \quad P \sqcup Q = (allows\,(P) \cup allows\,(Q), \\ \qquad\qquad denies\,(P) \cap denies\,(Q)) \end{array}$$

Formally, $P \sqsubseteq Q$ iff every packet allowed by P is allowed by Q and that any packet explicitly denied by Q is also explicitly denied by P. Note that in this definition we distinguish between packets *explicitly* denied in the policy versus packets *implicitly* denied by default. This means that, everything else being equal, a policy that explicitly denies a packet is considered more restrictive than a policy that relies on the implicit default-deny for the same packet; we shall see that this distinction is important when safely extending policies with new rules.

Safe replacement is defined as the cartesian product of subset orderings over accept and deny sets and it therefore follows that *Policy* is a partially ordered set under $\sqsubseteq$. $\bot$ and $\top$ define the most restrictive and least restrictive policies, that is, for any $P \in Policy$ we have $\bot \sqsubseteq P \sqsubseteq \top$. Thus, for example, any firewall enforcing a policy P can be safely reconfigured to enforce the (not very useful) firewall policy $\bot$.

Consider an update to the network security policy in our running example, where SSH traffic is to be denied to the HTTP and Git servers from malicious IP $BADIP. Some packets to be considered as part of this access restriction are mal_1 and mal_2, where

$$\mathsf{mal}_1 == (\$\mathsf{BADIP}, 1025, 172.16.1.6, 22, \mathsf{tcp})$$
$$\mathsf{mal}_2 == (\$\mathsf{BADIP}, 1025, 172.16.1.5, 22, \mathsf{tcp})$$

A firewall policy $\mathsf{Admin}_2 \in Policy$ extends the previous policy Admin_1 to deny these packets from this malicious host:

$$\mathsf{Admin}_2 == (\{\mathsf{http}_1, \mathsf{git}_1\}, \{\mathsf{tel}_1, \mathsf{mal}_1, \mathsf{mal}_2\})$$

We note that Admin_2 safely replaces Admin_1: $\mathsf{Admin}_2 \sqsubseteq \mathsf{Admin}_1$, but Admin_1 is not a safe replacement for Admin_2.

b) Policy intersection: Under this ordering, the meet, or intersection $P \sqcap Q$, of two firewall policies P and Q is defined as the policy that denies any packet that is explicitly denied by *either* P or Q, but allows packets that are allowed by both P and Q. Intuitively, this means that if a firewall is required to enforce both policies P and Q then it can be configured to enforce the policy $P \sqcap Q$, since $P \sqcap Q$ is a safe replacement for both P and Q, that is $(P \sqcap Q) \sqsubseteq P$ and $(P \sqcap Q) \sqsubseteq Q$. Given the definition of safe replacement as a product of two powerset lattices it follows that the policy meet provides the greatest lower bound operator. Thus, $P \sqcap Q$ provides the 'best'/least restrictive (under $\sqsubseteq$) safe replacement for both P and Q.

Consider a third firewall policy Admin_3, that permits the client IP address 192.168.2.5 SSH access to the HTTP and Git servers in our example, and denies all network traffic from the client destined for IP $BADIP. The policy $\mathsf{Admin}_3 == (\{\mathsf{ssh}_1, \mathsf{ssh}_2\}, \{\mathsf{mal}_3\})$ enforces these access restrictions according to packets ssh_1, ssh_2 and mal_3, where

$$\mathsf{ssh}_1 == (192.168.2.5, 1025, 172.16.1.6, 22, \mathsf{tcp})$$
$$\mathsf{ssh}_2 == (192.168.2.5, 1025, 172.16.1.5, 22, \mathsf{tcp})$$
$$\mathsf{mal}_3 == (192.168.2.5, 1025, \$BADIP, 6697, \mathsf{tcp})$$

We have $\mathsf{Admin}_2 \sqcap \mathsf{Admin}_3 = (\emptyset, \{\mathsf{tel}_1, \mathsf{mal}_1, \mathsf{mal}_2, \mathsf{mal}_3\})$, a safe replacement for both policies Admin_2 and Admin_3.

c) Policy Union: The *join* of two firewall policies P and Q is defined as the policy that allows any packet allowed by *either* P or Q, but denies packets that are explicitly denied by both P and Q. For example, we have $\mathsf{Admin}_2 \sqcup \mathsf{Admin}_3 = (\{\{\mathsf{http}_1, \mathsf{git}_1, \mathsf{ssh}_1, \mathsf{ssh}_2\}\}, \emptyset)$. Intuitively, this means that a firewall that is required to enforce either policy P or Q can be safely configured to enforce the policy $P \sqcup Q$ since $\sqcup$ provides a lowest upper bound operator and we have $P \sqsubseteq (P \sqcup Q)$ and $Q \sqsubseteq (P \sqcup Q)$.

d) Proposition: The set of all policies *Policy* forms a lattice under safe replacement. This follows from the definition of $\sqsubseteq$ as a cartesian product of two powerset lattice orderings.

A. Constructing firewall policies

The lattice of policies $\mathcal{FW}_0$ provides us with an algebra for constructing and interpreting firewall polices. The following constructor functions are used to build primitive policies.

Given a set of packets A then $(\mathsf{Allow}\, A)$ is a policy that allows packets in A, and $(\mathsf{Deny}\, D)$ is a policy that explicitly denies packets in D.

$$\mathsf{Allow}, \mathsf{Deny} : \mathbb{P}\, Packet \to Policy$$
$$\forall X : \mathbb{P}\, Packet \bullet \mathsf{Allow}\, X = (X, \emptyset) \land \mathsf{Deny}\, X = (\emptyset, X)$$

This provides what we refer to as a *weak* interpretation of allow and deny: packets that are not explicitly mentioned in parameter X are default-deny and therefore not specified in the deny set of the policy. The following provides us with a *strong* interpretation for these constructors:

$$\mathsf{Allow}^+, \mathsf{Deny}^+ : \mathbb{P}\, Packet \to Policy$$
$$\forall X : \mathbb{P}\, Packet \bullet \mathsf{Allow}^+\, X = (X, Packet \setminus X) \land \mathsf{Deny}^+\, X = (Packet \setminus X, X)$$

In this case $(\mathsf{Allow}^+\, A)$ allows packets specified in A while explicitly denying all other packets, and $(\mathsf{Deny}^+\, D)$ denies packets specified in D while allowing all other packets.

e) Proposition: A firewall policy P : *Policy* can be decomposed into their corresponding allow and deny policies and re-constructed using the algebra; for any $(A, D) \in Policy$, since A and D are disjoint then

$$\begin{aligned}(\mathsf{Allow}^+\, A) \sqcup (\mathsf{Deny}\, D) &= (A, Packet \setminus A) \sqcup (\emptyset, D) \\ &= (A, D) \\ &= (\mathsf{Allow}\, A) \sqcap (\mathsf{Deny}^+\, D)\end{aligned}$$

Thus, an alternative specification for policy Admin_3 is

$$\mathsf{Admin}_3 == (\mathsf{Allow}^+\, (\{\mathsf{ssh}_1, \mathsf{ssh}_2\}) \sqcup \mathsf{Deny}\, (\{\mathsf{mal}_3\})$$

f) Sequential Composition: Firewall policies are conventionally constructed as a sequence of rules, whereby for a given network packet, the decision to allow or deny a packet is checked against each policy rule, starting from the first, in sequence, and the first rule that matches gives the result that is returned. The algebra $\mathcal{FW}_0$ can be extended to include a similar form of sequential composition of policies. The policy constructions above can be regarded as representing the individual rules of a conventional firewall policy.

Let $(\mathsf{Allow}\, A) \fatsemi Q$ denote a sequential composition of an allow rule $(\mathsf{Allow}\, A)$ with policy Q with the interpretation that a packet that is matched by A is allowed; if it does not match A then policy Q is enforced. The resulting policy either: allows packets in A (and denies all other packets), or allows/denies packets according to policy Q. This is defined as:

$$\begin{aligned}(\mathsf{Allow}\, A) \fatsemi Q &= (\mathsf{Allow}^+\, A) \sqcup Q \\ &= ((A \cup allows(Q)), \\ &\qquad ((Packet \setminus A) \cap denies(Q))) \\ &= ((A \cup allows(Q)), (denies(Q) \setminus A))\end{aligned}$$

which is as expected. A similar definition can be provided for the sequential composition $(\mathsf{Deny}\, D) \fatsemi Q$ whereby a packet that is matched by D is denied; if it does not match D then policy Q is enforced. This is defined as:

$$\begin{aligned}(\mathsf{Deny}\, D) \fatsemi Q &= (\mathsf{Deny}^+\, D) \sqcap Q \\ &= (allows(Q) \setminus D, denies(Q) \cup D)\end{aligned}$$

While in practice its usual to write a firewall policy in terms of many constructions of allow and deny rules, in principle, any firewall policy P : *Policy* can be defined in terms of one allow policy $(\mathsf{Allow}\, allows(P))$ and one deny policy $(\mathsf{Deny}\, denies(P))$ and since the allow and deny sets of P are disjoint we have $P \fatsemi Q = (\mathsf{Deny}\, denies(P)) \fatsemi (\mathsf{Allow}\, allows(P)) \fatsemi Q$. We have

$$\_ \fatsemi \_ : Policy \times Policy \to Policy$$
$$\forall \mathcal{FW}_0;\ P, Q : Policy \bullet P \fatsemi Q = (Q \sqcup (\mathsf{Allow}^+\, (allows(P)))) \sqcap (\mathsf{Deny}^+\, (denies(P)))$$

Continuing the running example, we give the policy Admin_4 as the sequential composition of Admin_2 and Admin_3.

$\mathsf{Admin}_4 == \mathsf{Admin}_2 \fatsemi \mathsf{Admin}_3$, where:

$$
\begin{aligned}
&(\{\mathsf{http}_1, \mathsf{git}_1\}, \{\mathsf{tel}_1, \mathsf{mal}_1, \mathsf{mal}_2\}) \fatsemi (\{\mathsf{ssh}_1, \mathsf{ssh}_2\}, \{\mathsf{mal}_3\}) \\
&\quad = ((\{\mathsf{ssh}_1, \mathsf{ssh}_2\}, \{\mathsf{mal}_3\}) \\
&\qquad \sqcup (\{\mathsf{http}_1, \mathsf{git}_1\}, \mathit{Packet} \setminus \{\mathsf{http}_1, \mathsf{git}_1\})) \\
&\qquad \sqcap (\mathit{Packet} \setminus \{\mathsf{mal}_1, \mathsf{mal}_2, \mathsf{tel}_1\}), \{\mathsf{mal}_1, \mathsf{mal}_2, \mathsf{tel}_1\})) \\
&\quad = (\{\mathsf{ssh}_1, \mathsf{ssh}_2, \mathsf{http}_1, \mathsf{git}_1\}, \{\mathsf{tel}_1, \mathsf{mal}_1, \mathsf{mal}_2, \mathsf{mal}_3\})
\end{aligned}
$$

The policy negation of $P : \mathit{Policy}$ allows packets explicitly denied by P and explicitly denies packets allowed by P. We define:

$$
\begin{array}{|l}
\textbf{not} : \mathit{Policy} \to \mathit{Policy} \\
\hline
\forall\, \mathcal{FW}_0;\ P : \mathit{Policy} \bullet \\
\quad \textbf{not}\ P = (\mathsf{Allow}^+\ (\mathit{denies}\,(P)))\sqcup \\
\qquad\qquad (\mathsf{Deny}\,(\mathit{allows}\,(P)))
\end{array}
$$

Thus, the negation of policy Admin_4 is

$$
\begin{aligned}
\textbf{not}\ \mathsf{Admin}_4 = (&\mathsf{Allow}^+\ (\{\mathsf{tel}_1, \mathsf{mal}_1, \mathsf{mal}_2, \mathsf{mal}_3\})\sqcup \\
&(\mathsf{Deny}\,(\{\mathsf{ssh}_1, \mathsf{ssh}_2, \mathsf{http}_1, \mathsf{git}_1\}))
\end{aligned}
$$

From this definition it follows that $(\textbf{not}\ P)$ is simply $(\mathit{denies}\,(P), \mathit{allows}\,(P))$ and thus $\textbf{not}\,(\mathsf{Deny}\,D) = (\mathsf{Allow}\,D)$ and $\textbf{not}\,(\mathsf{Allow}\,A) = (\mathsf{Deny}\,A)$. Note however, that in general policy negation does not define a complement operator in the algebra $\mathcal{FW}_0$, that is, it not necessarily the case that $(P \sqcup \textbf{not}\ P) = \top$ and $(P \sqcap \textbf{not}\ P) = \bot$.

g) Policy projection: The projection operators $@^u$ and $@^d$ filter a policy by a set of IP addresses. Firstly, let $\mathit{alSrc}(S)$ give the set of all packets that have $s \in S$ as source IP, and similarly $\mathit{alDst}(D)$ gives all packets with a destination IP address $d \in D$.

$$
\begin{array}{|l}
\mathit{alSrc}, \mathit{alDst} : \mathbb{P}\, \mathit{IP} \to \mathbb{P}\, \mathit{Packet} \\
\hline
\forall\, S, D : \mathbb{P}\, \mathit{IP} \bullet \\
\quad \mathit{alSrc}(S) = S \times \mathit{PORT} \times \mathit{IP} \times \mathit{PORT} \times \mathit{PROTO} \land \\
\quad \mathit{alDst}(D) = \mathit{IP} \times \mathit{PORT} \times D \times \mathit{PORT} \times \mathit{PROTO}
\end{array}
$$

For a policy P and a set of IP addresses S, $P@^uS$ is the upstream projection of P, and consists of the allow and deny packets from P where each packet has as source IP some member of S. Similarly, $P@^dS$ is the downstream projection of P, it consists of the allow and deny packets from P whereby each packet has as destination IP some member of S.

$$
\begin{array}{|l}
\_@^u\_, \\
\_@^d\_ : \mathit{Policy} \times \mathbb{P}\ \mathit{IP} \to \mathit{Policy} \\
\hline
\forall\, P : \mathit{Policy};\ S : \mathbb{P}\ \mathit{IP} \bullet \\
\quad P\ @^u\ S = (\mathit{allows}(P) \cap \mathit{alSrc}(S), \mathit{denies}(P) \cap \mathit{alSrc}(S)) \\
\quad \land \\
\quad P\ @^d\ S = (\mathit{allows}(P) \cap \mathit{alDst}(S), \mathit{denies}(P) \cap \mathit{alDst}(S))
\end{array}
$$

B. Firewall policy anomalies

A firewall policy is conventionally constructed as a sequence of order-dependent rules. When a network packet matches with two or more policy rules, the policy is anomalous [4]–[6]. By definition, the allow set and deny set of some $P : \mathit{Policy}$ are disjoint, and therefore P is anomaly-free by construction. We can, define anomalies using the algebra, by considering how a policy changes when composed with other policies.

h) Redundancy: A policy P is redundant given policy Q if their composition results in no difference between the resulting policy and Q, in particular, $P \fatsemi Q = Q$. Considering policies Admin_1 and Admin_2, we see that Admin_1 is redundant to Admin_2 since $\mathsf{Admin}_1 \fatsemi \mathsf{Admin}_2 = \mathsf{Admin}_2$. We note that even though both policies explicitly accept the same packets, and that Admin_1 denies packets mal_1 and mal_2 by default, it is not the case that Admin_2 is redundant to Admin_1.

i) Shadowing: Some part of policy Q is shadowed by the entire policy P in the composition $P \fatsemi Q$ if the packet constraints that are specified by P contradict the constraints that are specified by Q, in particular, if $(\textbf{not}\ P) \fatsemi Q = Q$. This is a very general definition for shadowing. Perhaps a more familiar interpretation of this definition is one where the policy P is a specific allow/deny rule that shadows a part or all of the policy with which it is composed. Recall that $(\textbf{not}(\mathsf{Allow}\,A)) = (\mathsf{Deny}\,A)$ and, for example, in $(\mathsf{Allow}\,A) \fatsemi Q$ all or part of policy Q is shadowed by the rule/primitive policy $(\mathsf{Allow}\,A)$ if Q denies the packets specified in A, that is, $(\mathsf{Deny}\,A) \fatsemi Q = Q$. Similarly, in $(\mathsf{Deny}\,D) \fatsemi Q$ part or all of policy Q is shadowed by the rule/primitive policy $(\mathsf{Deny}\,D)$ if $(\textbf{not}\,(\mathsf{Deny}\,D)) \fatsemi Q = Q$.

Further definitions for shadowing may be constructed using the algebra. For example, an interpretation of the generalisation anomaly [4] in the composition $P \fatsemi Q$, where Q is a generalised by P if all of P shadows (specifically) part of Q. We are currently investigating how other anomalies can be reasoned about within the algebra.

j) Inter- policy anomalies: Anomalies can also occur between the different policies of distributed firewalls [5]. In the following, assume that P is a policy on an upstream firewall and Q is a policy on a downstream firewall. An inter-redundancy anomaly exists between policies P and Q if some part of Q is redundant to some part of P, whereby the target action of the redundant packets is deny. Given some set of packets A denied by P, and some set of packets B denied by Q, if $(\mathsf{Deny}\,A) \fatsemi (\mathsf{Deny}\,B) = (\mathsf{Deny}\,A)$ then there exists an inter-redundancy between P and Q.

An inter-shadowing anomaly exists between policies P and Q if some part of Q's allows are shadowed by some part of P's denies. Given some set of packets A denied by P, and some set of packets B allowed by Q, if $(\mathsf{Deny}\,A) \fatsemi (\mathsf{Allow}\,B) = (\mathsf{Deny}\,A)$, then there is an inter-shadowing anomaly between P and Q. An inter-spuriousness anomaly exists between policies P and Q if some part of Q's denies are shadowed by some part of P's allows. Again, given some set of packets A allowed by P, and some set of packets B denied by Q, if $(\mathsf{Allow}\,A) \fatsemi (\mathsf{Deny}\,B) = (\mathsf{Allow}\,A)$, then there exists an inter-spuriousness anomaly between P and Q.

III. OpenStack firewall policies

In this section the algebra is used to encode the network and host-based access controls available in OpenStack. The

OpenStack Networking service, called *Neutron*, is a standalone API-centric networking service. In general, the OpenStack networking configuration for a deployment will be segmented into four physical data center networks, as part of three distinct security domains. The *Management* network is used for inter-communication between OpenStack services, and is considered the *Management Security Domain*. The *API* network is used by tenants to access OpenStack API's, and is considered the *Public Security Domain*. The *External* network, also in the *Public Security Domain*, is used by virtual machines (VMs) for Internet access. The *Guest* network is used for instance-to-instance communication between VMs, and is considered the *Guest Security Domain*. In this paper, we focus on host-based and network access controls within the Guest Security Domain. These controls consist of perimeter firewall policies and Neutron security groups.

A. Perimeter firewall policies

Firewall-as-a-Service (FWaaS) adds perimeter firewall management to an OpenStack project by filtering traffic at the Neutron router. It is implemented as a sequence of iptables rules, where a default-deny policy is enforced. One firewall policy is supported per project, whereby the policy is applied to all networking routers within the project [2]. FWaaS is currently considered an experimental feature of OpenStack Networking [3]. A FWaaS rule can be constructed using the OpenStack Neutron command-line client:

```
neutron firewall-rule-create
  --source-ip-address $s
  --source-port $sprt
  --destination-ip-address $d
  --destination-port $dprt
  --protocol $p
  --action $act
```

The source ($\$s$) and destination ($\$d$) IP fields may be given as a single IP address/an IP address block (CIDR), the source ($\$sprt$) and destination ($\$dprt$) ports may be specified as single port values or ranges. The protocol ($\$p$) field may be given as TCP/UDP/ICMP/Any, and the action field ($\$act$) specifying the access decision, may be given as allow/deny.

A FWaaS policy is a sequence of allow and/or deny rules defined over packet filter conditions. Let FC define the set of all FWaaS filter conditions

$$FC == \mathbb{P}\, IP \times \mathbb{P}\, PORT \times \mathbb{P}\, IP \times \mathbb{P}\, PORT \times \mathbb{P}\, PROTO$$

whereby a filter condition $(s, sprt, d, dprt, p) \in FC$ matches a network packet that has a source IP address in s, originating from a source port in $sprt$, destined for destination IP in d, with destination port from $dprt$, using a protocol from p. A FWaaS rule defines an action (allow or deny) for a given filter condition. Let $Rule$ define the set of all FWaaS rules where

$$\begin{aligned} Rule ::= {} & \mathsf{allow}\ \langle\langle FC \rangle\rangle \\ | {} & \mathsf{deny}\ \langle\langle FC \rangle\rangle \end{aligned}$$

Let $flatten(f)$ define the flattening of a FWaaS filter condition $f \in FC$ into an equivalent set of individual packets, where

$$\begin{array}{|l} flatten : FC \to \mathbb{P}\, Packet \\ \hline \forall\, s, d : \mathbb{P}\, IP;\ sprt, dprt : \mathbb{P}\, PORT;\ p : \mathbb{P}\, PROTO \bullet \\ \quad flatten(s, sprt, d, dprt, p) = s \times sprt \times d \times dprt \times p \end{array}$$

k) Proposition: It follows from its definition that the flattening function defines an isomorphism between firewall filter conditions FC and $\mathbb{P}\, Packet$. This proposition means that any FWaaS allow rule $(\mathsf{allow} f) \in Rule$ has a corresponding unique representation $(\mathsf{Allow}\ flatten(f))$ in the policy algebra and vice-versa. With a similar result for FWaaS deny rules, we can define a FWaaS interpretation function as

$$\begin{array}{|l} \mathcal{I} : Rule \to Policy \\ \hline \forall f : FC \bullet \\ \quad \mathcal{I}(\mathsf{allow} f) = (\mathsf{Allow}\ (flatten\ (f))) \land \\ \quad \mathcal{I}(\mathsf{deny} f) = (\mathsf{Deny}\ (flatten\ (f))) \end{array}$$

A FWaaS firewall policy is defined as a sequence of rules $\langle r_1, r_2, \ldots, r_n \rangle$ for $r_i \in Rule$, and is encoded in the policy algebra as $\mathcal{I}(r_1) \fatsemi \mathcal{I}(r_2) \fatsemi \ldots \fatsemi \mathcal{I}(r_n)$.

B. Security group policies

A security group policy is a container for IP filter rules. Traditionally, security group capabilities were managed as part of the OpenStack Compute service, called *Nova*, and were instance-based. In Neutron, security groups are virtual interface port based. When utilizing Neutron as part of an OpenStack deployment, best practice [3] stipulates that security group capabilities be disabled in Nova, due to both possible conflicting policies, and also the more powerful capabilities of Neutron security groups. Security group rules allow administrators/tenants the ability to specify the type and direction of traffic that is allowed to pass through a virtual interface port. When a port is created in Neutron it is associated with a security group. If no security group is specified, a 'default' security group is assigned. This default group will drop all ingress traffic except that traffic originating from the default group, and allow all egress [3]. Rules may be added/removed to/from any security group by a tenant/administrator to change the default behaviour. A security group rule can be constructed using the OpenStack Neutron command-line client:

```
neutron security-group-rule-create
  --direction $dir
  --port-range-min $min
  --port-range-max $max
  --remote-ip-prefix $rsrc
  --remote-group-id $rsrc
  --protocol $p
  SECURITY_GROUP
```

The direction field ($\$dir$) is specified as ingress/egress. The remote source ($\$rsrc$) may be given as an IP address/an IP address block (CIDR) using the remote IP prefix, or as a

Neutron security group using the remote group id. Selecting a security group as the remote source will allow access to/from any instance in that security group, depending on the value specified in the direction field. The destination port ($\$min, \$max$) may be given as a single port/port range. The protocol field ($\$p$) may be specified as TCP/UDP/ICMP/Any. Additionally, when specifying a rule with an ICMP protocol, given that ICMP does not support ports, the specific ICMP Type/Code may be given in place of the destination port. The `SECURITY_GROUP` attribute specifies as to which security group to which this rule applies. Note that there are additional parameters that may be provided as command-line arguments, however the above are sufficient for our purposes.

A security group policy is a sequence of filter conditions that define the packets to be accepted relative to the members of the security group. Network traffic may flow to and from a security group, as defined by direction:

$$Dir ::= \mathsf{ingress} \mid \mathsf{egress}$$

Let FC_{SG} define the set of all security group filter conditions:

$$FC_{SG} == \mathbb{P}\, IP \times Dir \times \mathbb{P}\, IP \times \mathbb{P}\, PORT \times \mathbb{P}\, PROTO$$

A security group filter condition $(sgm, dir, rsrc, dprt, proto) \in FC_{SG}$ specifies that for all members sgm of the security group to which the rule belongs, network traffic is permitted in direction dir to/from remote-source $rsrc$ (depending on direction dir), to destination ports $dprt$, using protocols p.

A security group filter condition f can be mapped to the set of packets $flatten_{sg}(f)$ that it matches, whereby

$$\begin{array}{|l} flatten_{sg} : FC_{SG} \to \mathbb{P}\, Packet \\ \hline \forall\, sgm, rsrc : \mathbb{P}\, IP;\ prt : \mathbb{P}\, PORT;\ p : \mathbb{P}\, PROTO \bullet \\ \quad flatten_{sg}(sgm, \mathsf{egress}, rsrc, prt, p) \\ \qquad = sgm \times PORT \times rsrc \times prt \times p \wedge \\ \quad flatten_{sg}(sgm, \mathsf{ingress}, rsrc, prt, p) \\ \qquad = rsrc \times PORT \times sgm \times prt \times p \end{array}$$

If the direction attribute is ingress then the filter condition constrains packets coming from the remote source and destined to the members of the security group; if direction attribute is egress then the filter condition constrains packets coming from members of the security group (source) and destined to the remote resource. A security group rule is simply an allow action on its filter condition:

$$\begin{array}{|l} \mathcal{I}^s : FC_{SG} \to Policy \\ \hline \forall f : FC_{SG} \bullet \\ \quad \mathcal{I}^s(f) = (\mathsf{Allow}\,(flatten_{sg}(f))) \end{array}$$

A security group policy is written as a sequence of security group rules $\langle r_1, r_2, \ldots, r_n \rangle$ where each $r_i \in FC_{SG}$ and is encoded in the policy algebra as $\mathcal{I}^s(r_1) \fatsemi \mathcal{I}^s(r_2) \fatsemi \ldots \fatsemi \mathcal{I}^s(r_n)$. Note that in this encoding it is assumed that each rule in the original policy has the same membership, that is, $group(r_i) = group(r_j)$ for all rules r_i and r_j in the policy where $group(r)$ gives the group in the rule/filter condition r.

IV. Reasoning about OpenStack host controls

Continuing the running example, consider a company that migrated platforms and services to an on-premises OpenStack deployment. The deployment hosts both a development and a production cloud for the Web-service provided by the company and the code revision control systems for the Web-service. These two private clouds are defined as independent OpenStack projects/tenants as depicted in Figure 1. System Administrator Bob manages the network access controls (NAC) for both the development and production clouds. In the following subsections, a partial extract of the security group and FWaaS rules enforced in cloud deployment are examined.

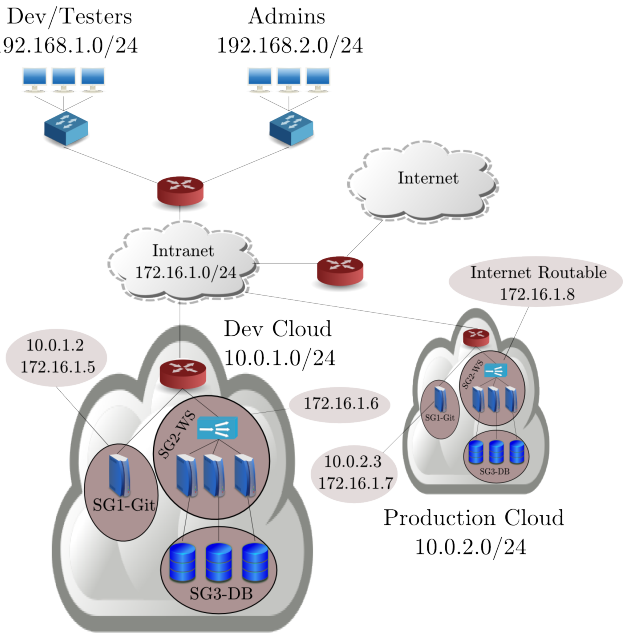

Fig. 1. Guest and External Network Architecture

A. Reasoning about security groups

Bob creates a security group policy $\mathsf{Git}_{\mathsf{SG1}}$ within the development cloud to manage the type of traffic permitted to/from the code revision control server. sgm_1 denotes the set of IP addresses for the members of this security group. Bob begins to add rules $\mathsf{git}_1, \mathsf{git}_2$, for ICMP ping for each member of the dev/tester subnet to allow developers and testers to ping the Git server in the development cloud.

$$\begin{aligned} \mathsf{git}_1 &= (\,\{\mathsf{192.168.1.3}\}, \mathsf{ingress}, \mathsf{sgm}_1, \\ &\qquad PORT, \{\mathsf{icmp8/0}\}) \\ \mathsf{git}_2 &= (\,\{\mathsf{192.168.1.4}\}, \mathsf{ingress}, \mathsf{sgm}_1, \\ &\qquad PORT, \{\mathsf{icmp8/0}\}\,) \\ \mathsf{Git}_{\mathsf{SG1}} &= \mathcal{I}^{\mathsf{s}}(\mathsf{git}_1) \fatsemi \mathcal{I}^{\mathsf{s}}(\mathsf{git}_2) \end{aligned}$$

Bob finds this tedious and decides to simply add a rule git_3 that allows all inbound ICMP traffic from the dev/tester subnet.

$$\begin{aligned} \mathsf{git}_3 &= (\,\{\mathsf{192.168.1.0/24}\}, \mathsf{ingress}, \mathsf{sgm}_1, \\ &\qquad PORT, \mathsf{icmpAll}\,) \\ \mathsf{Git}_{\mathsf{SG2}} &= \mathsf{Git}_{\mathsf{SG1}} \fatsemi \mathcal{I}^{\mathsf{s}}(\mathsf{git}_3) \end{aligned}$$

In doing so, however, git_1 and git_2 are now redundant to git_3, $(\mathcal{I}^s(\mathsf{git}_1) \fatsemi \mathcal{I}^s(\mathsf{git}_2)) \fatsemi \mathcal{I}^s(\mathsf{git}_3) = \mathcal{I}^s(\mathsf{git}_3)$.

Rule git_4 is introduced to allow all developers and testers access to the code revision control system (Git) in the development cloud, where:

$$\mathsf{git}_4 = (\{192.168.1.0/24\}, \mathsf{ingress}, \mathsf{sgm}_1, \{9418\}, \{\mathsf{tcp}\})$$
$$\mathsf{Git}_{\mathsf{SG3}} = \mathsf{Git}_{\mathsf{SG2}} \fatsemi \mathcal{I}^s(\mathsf{git}_4)$$

Cross-tenant access is required for source code replication, therefore Bob must ensure that rsync via SSH is permitted from the Git server in the development cloud to the Git server in the production cloud. To do so, he introduces rule git_5, where:

$$\mathsf{git}_5 = (\mathsf{sgm}_1, \mathsf{egress}, \{172.16.1.7\}, \{22\}, \{\mathsf{tcp}\})$$
$$\mathsf{Git}_{\mathsf{SG4}} = \mathsf{Git}_{\mathsf{SG3}} \fatsemi \mathcal{I}^s(\mathsf{git}_5)$$

Bob creates the security group policy $\mathsf{Web}_{\mathsf{SG}}$ to manage the accesses to/from the Web-service load balancer in the development cloud. Let sgm_2 denote the set of IP addresses for the members of this security group. Bob adds rules to allow HTTP traffic from the developers and testers (web_1), and from the administrators (web_2), where:

$$\mathsf{web}_1 = (\{192.168.1.0/24\}, \mathsf{ingress}, \mathsf{sgm}_2, \{80\}, \{\mathsf{tcp}\})$$
$$\mathsf{web}_2 = (\{192.168.2.0/24\}, \mathsf{ingress}, \mathsf{sgm}_2, \{80\}, \{\mathsf{tcp}\})$$
$$\mathsf{Web}_{\mathsf{SG}} = \mathcal{I}^s(\mathsf{web}_1) \fatsemi \mathcal{I}^s(\mathsf{web}_2)$$

The security group policy $\mathsf{DB}_{\mathsf{SG}}$ is created by Bob to manage accesses to/from the Web-service data tier in the development cloud. The literal sgm_3 denotes the set of IP addresses for the members of this security group. He adds the rule db_1 to allow all inbound traffic from members of the $\mathsf{Web}_{\mathsf{SG}}$ group to MySQL port 3306.

$$\mathsf{db}_1 = (\mathsf{sgm}_2, \mathsf{ingress}, \mathsf{sgm}_3, \{3306\}, \{\mathsf{tcp}\})$$
$$\mathsf{DB}_{\mathsf{SG}} = \mathcal{I}^s(\mathsf{web}_1)$$

The development cloud enforces the security group policies $\mathsf{Git}_{\mathsf{SG4}}, \mathsf{Web}_{\mathsf{SG}}$ and $\mathsf{DB}_{\mathsf{SG}}$. Recall that a security group policy is a container for allow rules managing the access to/from the security group, and that each security group policy in a cloud deployment is enforced independent of the other security group policies. Thus, the overall security group policy is the union of the individual policies:

$$\mathsf{Dev}_{\mathsf{SG}} = \mathsf{Git}_{\mathsf{SG4}} \sqcup \mathsf{Web}_{\mathsf{SG}} \sqcup \mathsf{DB}_{\mathsf{SG}}$$

B. Reasoning about FWaaS firewalls

As part of the configuration, Bob must also ensure the appropriate traffic traverses the perimiter firewall at the edge router of the development cloud. He therefore enforces FWaaS policy $\mathsf{Dev}_{\mathsf{FW1}}$ and begins to add some rules.

$$\mathsf{dev}_1 = \mathsf{deny}(\{172.16.1.5\}, \mathit{PORT}, \mathit{IP}, \{22\}, \{\mathsf{tcp}\})$$
$$\mathsf{dev}_2 = \mathsf{allow}(\{172.16.1.5\}, \mathit{PORT}, \mathit{IP}, \{22\}, \{\mathsf{tcp}\})$$
$$\mathsf{Dev}_{\mathsf{FW1}} = \mathcal{I}(\mathsf{dev}_1) \fatsemi \mathcal{I}(\mathsf{dev}_2)$$

Bob mistakenly introduces rule dev_1, creating a shadowing anomaly of rule dev_2, that is, $\mathbf{not}(\mathcal{I}(\mathsf{dev}_1)) \fatsemi \mathcal{I}(\mathsf{dev}_2) = \mathcal{I}(\mathsf{dev}_2)$, whereby the logical traffic flow is broken between the code revision control systems in the development and production clouds.

Rule dev_3 ensures the developers and testers are permitted to ping the Git server in the development cloud.

$$\mathsf{dev}_3 = \mathsf{allow}(\{192.168.1.0/24\}, \mathit{PORT}, \{172.16.1.5\}, \mathit{PORT}, \mathsf{icmpAll})$$
$$\mathsf{Dev}_{\mathsf{FW2}} = \mathsf{Dev}_{\mathsf{FW1}} \fatsemi \mathcal{I}(\mathsf{dev}_3)$$

The rule dev_4 permits unwanted Telnet traffic to the Git server, thereby allowing spurious traffic into the development cloud, where:

$$\mathsf{dev}_4 = \mathsf{allow}(\{192.168.1.0/24\}, \mathit{PORT}, \{172.16.1.5\}, \{21\}, \{\mathsf{tcp}\})$$
$$\mathsf{Dev}_{\mathsf{FW3}} = \mathsf{Dev}_{\mathsf{FW2}} \fatsemi \mathcal{I}(\mathsf{dev}_4)$$

Recall that all traffic entering the development cloud must traverse the development cloud perimeter firewall, and that the policy defining the complete set of internal accesses for the cloud is given as $\mathsf{Dev}_{\mathsf{SG}}$. Thus, the policy constraining accesses for traffic from upstream firewall $\mathsf{Dev}_{\mathsf{FW3}}$ to downstream composite security groups $\mathsf{Dev}_{\mathsf{SG}}$ is calculated as:

$$\mathsf{Pol}^{\mathsf{IN}}_{\mathsf{DEV}} = \mathsf{Dev}_{\mathsf{SG}} @^d (\mathsf{sgm}_1 \cup \mathsf{sgm}_2 \cup \mathsf{sgm}_3 \cup \mathsf{floatIP}) \fatsemi \mathsf{Dev}_{\mathsf{FW3}} @^d (\mathsf{sgm}_1 \cup \mathsf{sgm}_2 \cup \mathsf{sgm}_3 \cup \mathsf{floatIP})$$

where floatIP is the set of floating IP addresses used by the tenant. Note that all security group members are included since (at the time of writing) FWaaS applies to all routers within a tenant, not just to the perimeter.

l) The production cloud firewall: Bob must also ensure the configuration is correct for the production cloud perimiter firewall $\mathsf{Prod}_{\mathsf{FW1}}$.

$$\mathsf{prod}_1 = \mathsf{deny}(\{\$\mathsf{BADIP}\}, \mathit{PORT}, \mathit{IP}, \{80\}, \{\mathsf{tcp}\})$$
$$\mathsf{Prod}_{\mathsf{FW1}} = \mathcal{I}(\mathsf{prod}_1)$$

Rule prod_1 denies HTTP traffic to the production Web servers from the known malicious IP range $BADIP.

$$\mathsf{prod}_2 = \mathsf{allow}(\{172.16.1.5\}, \mathit{PORT}, \{172.16.1.7\}, \{22\}, \{\mathsf{tcp}\})$$
$$\mathsf{Prod}_{\mathsf{FW2}} = \mathsf{Prod}_{\mathsf{FW1}} \fatsemi \mathcal{I}(\mathsf{prod}_2)$$

Rule prod_2 is introduced by Bob to ensure the rsync via SSH between the code revision control systems in the development and production clouds is permitted. However, the problematic rule dev_1 in the development cloud firewall, has also caused a shadowing anomaly between the two perimeter firewalls ($\mathbf{not}(\mathcal{I}(\mathsf{dev}_1)) \fatsemi \mathcal{I}(\mathsf{prod}_2) = \mathcal{I}(\mathsf{prod}_2)$), whereby the the development cloud firewall is denying traffic from the development Git server to the Git server in the production cloud, while the rule prod_2 of the production cloud firewall is permitting the traffic.

$$\mathsf{prod}_3 = \mathsf{allow}(\mathit{IP}, \mathit{PORT}, \mathit{IP}, \{80\}, \{\mathsf{tcp}\})$$
$$\mathsf{Prod}_{\mathsf{FW3}} = \mathsf{Prod}_{\mathsf{FW2}} \fatsemi \mathcal{I}(\mathsf{prod}_3)$$

Bob introduces prod_3 to allow all other HTTP traffic to the Web-service load balancer. The rule prod_3 is generalised by the rule at prod_1, as prod_1 partially shadows prod_3.

m) Production cloud Git security group: In the production cloud, Bob creates security group $\mathsf{Git2_{SG}}$ to manage the type of traffic permitted to/from the production code revision control server. The literal sgm_4 denotes the set of IP addresses for the members of this security group. Rule $\mathsf{git}_1^{\mathsf{p}}$ is introduced to allow all developers and testers access to the code revision control system (Git) in the production cloud, where:

$$\begin{array}{l}\mathsf{git}_1^{\mathsf{p}} = (\,\{\mathsf{192.168.1.0/24}\}, \mathsf{ingress}, \mathsf{sgm}_4,\\ \qquad \{\mathsf{9418}\}, \{\mathsf{tcp}\}\,)\\ \mathsf{Git2_{SG}} = \mathcal{I}^{\mathsf{s}}\,(\mathsf{git}_1^{\mathsf{p}})\end{array}$$

The rule $\mathsf{git}_2^{\mathsf{p}}$, intended to ensure rsync via SSH between the code revision control systems in the development and production clouds is permitted, is inter-shadowed by the upstream rule dev_1 of the development cloud perimiter firewall, where:

$$\begin{array}{l}\mathsf{git}_2^{\mathsf{p}} = (\,\{\mathsf{172.16.1.5}\}, \mathsf{ingress}, \mathsf{sgm}_4, \{\mathsf{22}\}, \{\mathsf{tcp}\}\,)\\ \mathsf{Git2_{SG2}} = \mathsf{Git2_{SG}} \fatsemi \mathcal{I}^{\mathsf{s}}\,(\mathsf{git}_2^{\mathsf{p}})\end{array}$$

Recall $\mathsf{Pol}_{\mathsf{DEV}}^{\mathsf{IN}}$ which provided a policy about traffic traversing the perimeter firewall for the development cloud to the composite security group. A similar policy can be given for the traffic traversing the perimeter firewall of the production cloud destined to the security group within the production cloud. Further definitions can be given about policies on traffic leaving the respective clouds. These in turn can be composed to give a policy that is effectively about the rsync via ssh for the code revision systems in the development cloud to the production cloud.

V. Policies over IP and port ranges

The proposed algebra provides a semantics for firewall policies. While useful for the purposes of reasoning, it is not efficient to naively implement the algebra since a policy is defined in terms of rules constraining *individual* IP addresses and ports. For example, a policy constraining access from a subnet range `192.168.*.*` involves more than 65K individual packet rules, whatever about the impact of combining these with further constraints on destination IPs and ports. In practice, firewall rules are defined in terms of *ranges* of IP addresses and ports. The policy algebra $\mathcal{FW}_I$ defines a firewall policy in terms of rules constraining *ranges* of IP addresses and ports. Extending the policy algebra to include ranges is a non-trivial revision to $\mathcal{FW}_0$, as is illustrated using the following example.

Suppose that a policy is defined in terms of (allow and deny) sets of IP address and port ranges, where we use natural numbers to represent individual IP addresses and ports. For example, the policy

$$P_1 = (\{([1\,..\,3],[1\,..\,3],[1\,..\,3],[1\,..\,3],\{\mathsf{tcp}\})\},\emptyset)$$

has no deny constraints ($\emptyset$) and has one accept rule that permits any packet matching $([1\,..\,3],[1\,..\,3],[1\,..\,3],[1\,..\,3],\{\mathsf{tcp}\})$ (ranges of source and destination IP addresses and ports, and protocols). A second policy is similarly defined:

$$P_2 = (\{([2\,..\,4],[2\,..\,4],[2\,..\,4],[2\,..\,4],\{\mathsf{tcp}\})\},\emptyset)$$

In composing these policies under a lowest-upper-bound style operation one cannot simply take a union of the sets of intervals as in some cases they may coalesce and in other cases they may partition into a number of disjoint intervals. The composition of the above policies, result in the following policy.

$$\begin{array}{rl}P_1 \sqcup P_2 = (\{&([1\,..\,4],[2\,..\,3],[2\,..\,3],[2\,..\,3],\{\mathsf{tcp}\}),\\ &([1\,..\,3],[1],[1\,..\,3],[1\,..\,3],\{\mathsf{tcp}\}),\\ &([2\,..\,4],[4],[2\,..\,4],[2\,..\,4],\{\mathsf{tcp}\}),\\ &([1\,..\,3],[2\,..\,3],[2\,..\,3],[1],\{\mathsf{tcp}\}),\\ &([1\,..\,3],[2\,..\,3],[1],[1\,..\,3],\{\mathsf{tcp}\}),\\ &([2\,..\,4],[2\,..\,3],[2\,..\,3],[4],\{\mathsf{tcp}\}),\\ &([2\,..\,4],[2\,..\,3],[4],[2\,..\,4],\{\mathsf{tcp}\})\},\emptyset)\end{array}$$

VI. Related Work

In this section, we examine a selection of related research from the perspective of firewall policy modelling, firewall configuration analysis and firewall/security policy composition/refinement.

Much work has been completed in area of firewall configuration analysis, and various types of firewall policy model have been proposed. In [4], a firewall policy is modelled as a single rooted tree. Relations between rules are defined on a pairwise basis, and definitions for firewall configuration anomalies are provided. In [5], the work is extended to distributed firewall policies. In [6], a firewall policy is modelled as a linked-list, and in [10] rule relations within a policy are modelled in a directed graph. In [11], [16] Binary Decision Diagrams (BDDs) are used to model firewall rulesets. Concerning configuration analysis, in [4]–[6], [10], [16] an algorithmic approach is taken to detect/resolve anomalies.

Our work differs from these, firstly, in that we model a firewall policy as an ordered pair of disjoint sets, where the set of policies *Policy* forms a lattice under $\sqsubseteq$, and each $P \in$ *Policy* is anomaly-free by construction. Secondly, we regard as a novel aspect of our work, the algebraic (as opposed to algorithmic) approach taken towards modelling anomalies in a single policy, and across a distributed policy configuration through policy composition.

In [17], a firewall policy algebra is proposed. However, the authors note that an anomaly-free composition is not guaranteed as a result of using the algebraic operators they define. Our work differs, in that policy composition under the $\sqcup, \sqcap$ and $\fatsemi$ operators defined in this paper all result in anomaly-free policies.

In [13], a process algebraic approach for the migration of VMs and related packet-filter firewall policies is presented. The algebra, called *cloud calculus*, is used to capture the topology of cloud computing systems and the global firewall policy for a given configuration. We would view our work presented in this paper as a possible extension of the work in [13], given

that the $\mathcal{FW}_0$ algebra may be used in conjuction with cloud calculus to guarantee anomaly-free dynamic firewall policy reconfiguration. We would also view the ordering relation $\sqsubseteq$ as a viable alternative for the given equivalence relation defined over 'cloud' terms for the formal verification of firewall policy preservation after a live migration.

VII. Conclusion

A policy algebra $\mathcal{FW}_0$ is defined in which firewall policies can be specified and reasoned about. At the heart of this algebra is the notion of safe replacement, that is, whether it is secure to replace one firewall policy by another. The set of policies form a lattice under safe replacement and this enables consistent operators for safe composition to be defined. Policies in this lattice are anomaly-free by construction, and thus, composition under greatest lower and lowest upper bound operators preserves anomaly-freedom. A policy sequential composition operator is also proposed that can be used to interpret firewall policies defined more conventionally as sequences of filter condition rules. The algebra can be used to characterize anomalies, such as shadowing and redundancy, that arise from sequential composition.

The algebra $\mathcal{FW}_0$ provides a formal interpretation of the host-based and network access controls in OpenStack. In particular, it gives a meaning for OpenStack security group policies and perimeter firewalls. This provides us with a uniform notation to define and reason about different kinds policies in OpenStack. For example, reasoning over combinations of perimeter firewall and security group policies to ensure that modifications are safe (replacements) and checking for heterogenous inter-policy anomalies.

The results in this paper are described in terms of the algebra $\mathcal{FW}_0$ for stateless firewall policies. As outlined in Section V, supporting stateful policies defined in terms of constraints on *ranges* of IPs and ports requires a more expressive algebra. While this algebra $\mathcal{FW}_I$ is not described in this paper for reasons of space, it has been been implemented in Python and can be used to model and reason about OpenStack policies in a manner similar to $\mathcal{FW}_0$.

Acknowledgements

This research has been supported in part by by Science Foundation Ireland grants SFI 10/CE/I1853 and SFI 13/RC/2077.

References

[1] Openstack - Open source software for creating private and public clouds. *https://www.openstack.org/, Website last accessed,* July 2015.

[2] *OpenStack Cloud Administrator Guide, Eight Revision*. OpenStack Foundation, February 2015.

[3] *OpenStack Security Guide, 6th Revision*. OpenStack Foundation, April 2015.

[4] E. Al-Shaer and H. Hamed. Firewall Policy Advisor for Anomaly Discovery and Rule Editing. In *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*, pages 17–30, 2003.

[5] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE J.Sel. A. Commun.*, 23(10):2069–2084, September 2006.

[6] F. Cuppens, N. Cuppens-Boulahia, and J. Garcia-Alfaro. Detection and removal of firewall misconfiguration. In *Proceedings of the 2005 IASTED International Conference on Communication, Network and Information Security*, volume 1, pages 154–162, 2005.

[7] S.N. Foley. A model for secure information flow. In *IEEE Symposium on Security and Privacy*, May 1989.

[8] S.N. Foley. The specification and implementation of commercial security requirements including dynamic segregation of duties. In *ACM Conference on Computer and Communications Security*, 1997.

[9] S.N. Foley and U. Neville. An algebra for iptables firewall policies. Technical report, in preparation, University College Cork, Ireland, 2015.

[10] A. Hari, S. Suri, and G. Parulkar. Detecting and resolving packet filter conflicts. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1203–1212. IEEE, 2000.

[11] S. Hazelhurst, A. Fatti, and A. Henwood. Binary decision diagram representations of firewall and router access lists. *Department of Computer Science, University of the Witwatersrand, Tech. Rep*, 1998.

[12] J.L. Jacob. The varieties of refinement. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, pages 441–455. Springer-Verlag, 1991.

[13] Y. Jarraya, A. Eghtesadi, M. Debbabi, Y. Zhang, and M. Pourzandi. Cloud calculus: Security verification in elastic cloud computing platform. In Waleed W. Smari and Geoffrey Charles Fox, editors, *CTS*, pages 447–454. IEEE, 2012.

[14] J. M. Spivey. *The fuzz manual*. Computing Science Consultancy, 1992.

[15] J. M. Spivey. *The Z Notation: A Reference Manual*. Series in Computer Science. Prentice Hall International, second edition, 1992.

[16] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 199–213, Washington, DC, USA, 2006. IEEE Computer Society.

[17] H. Zhao and S. M. Bellovin. Policy algebras for hybrid firewalls. Technical Report CUCS-017-07, Columbia University, 2007.

Appendix

A set may be defined in Z using set specification in comprehension. This is of the form $\{ D \mid P \bullet E \}$, where D represents declarations, P is a predicate and E an expression. The components of $\{ D \mid P \bullet E \}$ are the values taken by expression E when the variables introduced by D take all possible values that make the predicate P true. For example, the set of squares of all even natural numbers is defined as $\{ n : \mathbb{N} \mid (n \bmod 2) = 0 \bullet n^2 \}$. When there is only one variable in the declaration and the expression consists of just that variable, then the expression may be dropped if desired. For example, the set of all even numbers may be written as $\{ n : \mathbb{N} \mid (n \bmod 2) = 0 \}$. Sets may also be defined in display form such as $\{1, 2\}$.

In Z, relations and functions are represented as sets of pairs. A (binary) relation R, declared as having type $A \leftrightarrow B$, is a component of $\mathbb{P}(A \times B)$, where $\mathbb{P}X$ is the powerset of X. For $a \in A$ and $b \in B$, then the pair (a, b) is written as $a \mapsto b$, and $a \mapsto b \in R$ means that a is related to b under relation R. Functions are treated as special forms of relations. The schema notation is used to structure specifications. A schema such as $\mathcal{FW}_0$ defines a collection of variables (limited to the scope of the schema) and specifies how they are related. The variables can be introduced via schema inclusion, as done, for example, in the definition of sequential composition.