Technical Report

Cornell University
Computer Science

NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson
Swarthmore College *

Nate Foster
Cornell University

Arjun Guha
Cornell University

Jean-Baptiste Jeannin
Cornell University

Dexter Kozen
Cornell University

Cole Schlesinger
Princeton University

David Walker
Princeton University

Cornell University
Computing and Information Science
October 5, 2013

Abstract. Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation has left language designers with little guidance in determining how to incorporate new features, and programmers without a means to reason precisely about their code.

This paper presents NetKAT, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe the design of NetKAT, including primitives for filtering, modifying, and transmitting packets; operators for combining programs in parallel and in sequence; and a Kleene star operator for iteration. We show that NetKAT is an instance of a canonical and well-studied mathematical structure called a Kleene algebra with tests (KAT) and prove that its equational theory is sound and complete with respect to its denotational semantics. Finally, we present practical applications of the equational theory including syntactic techniques for checking reachability properties, proving the correctness of compilation and optimization algorithms, and establishing a non-interference property that ensures isolation between programs.

* Work perfomed while employed at Cornell University.

1. Introduction

Traditional network devices have been called "the last bastion of mainframe computing" [9]. Unlike modern computers, which are implemented with commodity hardware and programmed using standard interfaces, networks are built the same way as in the 1970s: out of special-purpose devices such as routers, switches, firewalls, load balancers, and middle-boxes, each implemented with custom hardware and programmed using proprietary interfaces. This design makes it difficult to extend networks with new functionality and effectively impossible to reason precisely about their behavior.

However, in recent years, a revolution has taken place with the rise of *software-defined networking* (SDN). In SDN, a general-purpose *controller machine* manages a collection of *programmable switches*. The controller responds to network events such as newly connected hosts, topology changes, and shifts in traffic load by re-programming the switches accordingly. This *logically centralized*, global view of the network makes it possible to implement a wide variety of standard applications such as shortest-path routing, traffic monitoring, and access control, as well as more sophisticated applications such as load balancing, intrusion detection, and fault-tolerance using commodity hardware.

A major appeal of SDN is that it defines open standards that any vendor can implement. For example, the OpenFlow API defines a low-level configuration interface that clearly specifies the capabilities and behavior of switch hardware. However, programs written directly for SDN platforms such as OpenFlow are akin to assembly: easy for hardware to implement, but difficult for humans to write.

Network programming languages. Several research groups have proposed domain-specific SDN programming languages [5–7, 22–24, 28, 29]. These *network programming languages* allow programmers to specify the behavior of each switch using high-level abstractions that a compiler translates to low-level instructions for the underlying hardware. Unfortunately, the design of these languages has been largely ad hoc, driven more by the needs of individual applications and the capabilities of present-day hardware than by any foundational principles. Indeed, the lack of guiding principles has left language designers unsure about which features to incorporate into their languages and programmers without a means to reason directly about their programs.

As an example, the NetCore language [7, 22, 23] provides a rich collection of programming primitives including predicates that filter packets, actions that modify and forward packets, and composition operators that build larger policies out of smaller ones. NetCore has even been formalized in Coq. But like other network programming languages, the design of NetCore is ad hoc. As the language has evolved, its designers have added, deleted, and changed the meaning of primitives as needed. Without principles or metatheory to guide its development, the evolution of NetCore has lacked clear direction and foresight. It is not clear which constructs are essential and which can be derived. When new primitives are added, it is not clear how they should interact with existing constructs.

An even more pressing issue is that these languages specify the behavior of the switches in the network, but nothing more. Of course, when network programs are actually executed, the end-to-end functionality of the system is determined both by the behavior of switches and by the structure of the network topology. Hence, to answer almost any interesting question such as *"Can X connect to Y?"*, *"Is traffic from A to B routed through Z?"*, or *"Is there a loop involving S?"*, the programmer must step outside the confines of the linguistic model and the abstractions it provides.

To summarize, we believe that a foundational model for network programming languages is essential. Such a model should (1) identify the essential constructs for programming networks, (2) provide guidelines for incorporating new features, and (3) unify reasoning about switches, topology and end-to-end behavior. No existing network programming language meets these criteria.

Semantic foundations. We begin our development by focusing on the global behavior of the whole network. This is in contrast to previous languages, which have focused on the local behavior of the individual switches. Abstractly, a network can be seen as an automaton that shuttles packets from node to node along the links in its topology. Hence, from a linguistic perspective, it is natural to begin with regular expressions, the language of

finite automata. Regular expressions are a natural way to specify the packet-processing behavior of a network: a path is represented as a concatenation of processing steps ($p; q; \cdots$), a set of paths is represented using union ($p + q + \cdots$), and iterated processing is represented using Kleene star. Moreover, by modeling networks in this way, we get a ready-made theory for reasoning about formal properties: *Kleene algebra* (KA), a decades-old sound and complete equational theory of regular expressions.

With Kleene algebra as the choice for representing global network structure, we can turn our attention to specifying local switch-processing functionality. Fundamentally, a switch implements *predicates* to match packets and *actions* that transform and forward matching packets. Existing languages build various abstractions atop the predicates and actions supplied by the hardware, but predicates and actions are essential. As a consequence, a foundational model for SDN must incorporate both *Kleene algebra* for reasoning about network structure and *Boolean algebra* for reasoning about the predicates that define switch behavior. Fortunately, these classic mathematical structures have already been unified in previous work on *Kleene algebra with tests* (KAT) [14].

By now KAT has a well-developed metatheory, including an extensive model theory and results on expressiveness, deductive completeness, and complexity. The axioms of KAT are sound and complete over a variety of popular semantic models, including language, relational, and trace models, and KAT has been applied successfully in a number of application areas, including the verification of compiler optimizations, device drivers and communication protocols. Moreover, equivalence in KAT has a PSPACE decision procedure. This paper applies this theory to a new domain: networks.

NetKAT. NetKAT is a new framework based on Kleene algebra with tests for specifying, programming, and reasoning about networks. As a programming language, NetKAT has a simple denotational semantics inspired by that of NetCore [22], but modified and extended in key ways to make it sound for KAT (which NetCore is not). In this respect, the semantic foundation provided by KAT has delivered true guidance: the axioms of KAT dictate the interactions between primitive program actions, predicates, and other operators. Moreover, any future proposed primitive that violates a KAT axiom can be summarily rejected for breaking the equations that allow us to reason effectively about the network. NetKAT thus provides a foundational structure and consistent reasoning principles that other network programming languages lack.

For specification and reasoning, NetKAT also has an axiomatic semantics, characterized by a finite set of equations that capture equivalences between NetKAT programs. The equational theory includes the axioms of KAT, as well as domain-specific axioms that capture manipulations of packets. These axioms enable reasoning about local switch processing functionality (needed in compilation and optimization) as well as global network behavior (needed to check reachability and traffic isolation properties). We prove that the equational theory is sound and complete with respect to the denotational semantics. While soundness follows mostly standard lines, our proof of completeness is novel: we construct an alternate language model for NetKAT and leverage the completeness of KA.

To evaluate the practical utility of our theory and the expressive power of NetKAT, we demonstrate how to use it to reason about a diverse collection of applications. First, we show that NetKAT can pose a variety of interesting reachability queries that network operators need to answer. Next, we state and prove a non-interference property for networks that provides a strong form of isolation between NetKAT programs. Finally, we prove that NetKAT can be correctly compiled to a low-level form analogous to switch flow tables.

In summary, the main contributions of this paper are as follows:

- We develop a new semantic foundation for network programming languages based on Kleene algebra with tests (KAT).
- We formalize the NetKAT language in terms of a denotational semantics and an axiomatic semantics based on KAT; we prove the equational axioms sound and complete with respect to the denotational semantics.
- We apply the equational theory in several diverse domains including reasoning about reachability, traffic isolation, and compiler correctness.

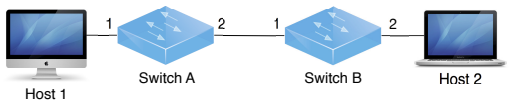

Figure 1. Example network.

The next section presents a simple example to motivate NetKAT and introduces the key elements of its design. The subsequent sections define the language formally, develop its main theoretical properties, and discuss applications.

2. Overview

This section introduces the syntax and informal semantics of NetKAT with a simple example. Consider the network shown in Figure 1. It consists of two switches, A and B, each with two ports labeled 1 and 2, and two hosts. The switches and hosts are connected together in series. Suppose we want to configure the network to provide the following services:

- *Forwarding:* transfer packets between the hosts, but
- *Access control:* block SSH packets.

The forwarding component is straightforward—program each switch to forward packets destined for host 1 out port 1, and similarly for host 2—but there are many ways to implement the access control component. We will describe several implementations and show that they are equivalent using NetKAT's equational theory.

Forwarding. To warm up, let us define a simple NetKAT policy that implements forwarding. To a first approximation, a NetKAT policy can be thought of as a function from packets to sets of packets. (In the next section we will generalize this type to functions from lists of packets to sets of lists of packets, where the lists encode packet-processing histories, to support reasoning about network-wide properties.) We represent packets as records with fields for standard headers such as source address (src), destination address (dst), and protocol type (typ), as well as two fields, switch (sw) and port (pt), that identify the location of the packet in the network.

The most basic NetKAT policies are filters and modification. A filter $(f = n)$ takes an input packet pk and yields the singleton set $\{pk\}$ if field f of pk equals n, and $\{\}$ otherwise. A modification $(f \leftarrow n)$ takes an input packet pk and yields the singleton set $\{pk'\}$, where pk' is the packet obtained from pk by setting f to n.

To help programmers build richer policies, NetKAT also has policy combinators that build bigger policies out of smaller ones. Parallel composition $(p + q)$ produces the union of the sets produced by applying each of p and q to the input packet, while sequential composition $(p; q)$ first applies p to the input packet, then applies q to each packet in the resulting set, and finally takes the union of the resulting sets. Using these operators, we can implement the forwarding policy for the switches:

$$p \triangleq (\mathsf{dst} = H_1; \mathsf{pt} \leftarrow 1) + (\mathsf{dst} = H_2; \mathsf{pt} \leftarrow 2)$$

At the top level, this policy is the parallel composition of two sub-policies. The first updates the pt field of all packets destined for H_1 to 1 and drops all other packets, while the second updates the pt field of all packets destined for H_2 to 2. The parallel composition of the two generates the union of their behaviors—in other words, the policy forwards packets across the switches in both directions.

Access control. Next, we extend the policy with access control. The simplest way to do this is to compose a filter that blocks SSH traffic with the forwarding policy:

$$p_{ac} \triangleq \neg(\mathsf{typ} = \mathrm{SSH}); p$$

This policy drops the input packet if its typ field is SSH and otherwise forwards it using p. Of course, a quick inspection of the network topology shows that it is unnecessary to test *all* packets at *all* places in the network

in order to block SSH traffic: packets travelling between host 1 and host 2 must traverse both switches, so it is sufficient to filter only at switch A:

$$p_A \triangleq (\mathsf{sw} = A; \neg(\mathsf{typ} = \text{SSH}); p) + (\mathsf{sw} = B; p)$$

or only at switch B:

$$p_B \triangleq (\mathsf{sw} = A; p) + (\mathsf{sw} = B; \neg(\mathsf{typ} = \text{SSH}); p)$$

These policies are slightly more complicated than the original policy, but are more efficient because they avoid having to store and enforce the access control policy at both switches.

Naturally, one would prefer one of the two optimized policies. Still, given these programs, there are several questions we would like to be able to answer:

- *"Are non-*SSH *packets forwarded?"*
- *"Are* SSH *packets dropped?"*
- *"Are p_{ac}, p_A, and p_B equivalent?"*

Network administrators ask these sorts of questions whenever they modify the policy or the network itself. Note that we cannot answer them just by looking at the policies—the answers depend on the network topology. We will see how to incorporate the topology as a part of the NetKAT program next.

Topology. Given a program that describes the behavior of the switches, it is not hard to define a semantics that accounts for the topology. For example, Guha et al. [7] define a network-wide operational relation that maps sets of packets to sets of packets by modeling the topology as a partial function from locations to locations. At each step, the semantics extracts a packet from the set of in-flight packets, applies the policy and the topology in sequence, and adds the results back in to the set of in-flight packets. However, to reason syntactically about network-wide behavior we need a *uniform representation* of policy and topology, not a separate auxiliary relation.

A network topology is a directed graph with switches as nodes and links as edges. We can model the behavior of the topology as the parallel composition of policies, one for each link in the network. Each link policy is the sequential composition of a filter that retains packets located at one end of the link and a modification that updates the sw and pt fields to the location at the other end of the link, thereby capturing the effect of sending a packet across the link. We assume that links are uni-directional, and encode bi-directional links as pairs of uni-directional links. For example, we can model the links between switches A and B with the following policy:

$$\begin{aligned} t = (\mathsf{sw} = A; \mathsf{pt} = 2; \mathsf{sw} \leftarrow B; \mathsf{pt} \leftarrow 1) + \\ (\mathsf{sw} = B; \mathsf{pt} = 1; \mathsf{sw} \leftarrow A; \mathsf{pt} \leftarrow 2) \end{aligned}$$

We call such a policy a *topology assertion*, since it is expected to be consistent with the physical topology.

Switches meet topology. A packet traverses the network in interleaved steps, processed first by a switch, then sent along the topology, and so on. In our example, if host 1 sends a non-SSH packet to host 2, it is first processed by switch A, then the link between A and B, and finally by switch B. This can be encoded by the NetKAT term $p_{ac}; t; p_{ac}$. More generally, a packet may require an arbitrary number of steps—in particular, if the network topology has a cycle. Using the Kleene star operator, which iterates a policy zero or more times, we can encode the end-to-end behavior of the network:

$$(p_{ac}; t)^*; p_{ac}$$

Note however that this policy processes packets that enter and exit the network at arbitrary locations, including internal locations such as on the link between switches A and B. In proofs, it is often useful to restrict attention to packets that enter and exit the network at specified external locations e:

$$e \triangleq (\mathsf{sw} = A; \mathsf{pt} = 1) + (\mathsf{sw} = B; \mathsf{pt} = 2)$$

Using this predicate, we can restrict the policy to packets sent or received by one of the hosts:

$$p_{net} \triangleq e; (p_{ac}; t)^*; p_{ac}; e$$

More generally, the input and output predicates may be distinct:

$$in; (p; t)^*; p; out$$

We call a network modeled in this way a *logical crossbar* [20], since it encodes the end-to-end processing behavior of the network (and elides processing steps on internal hops). This encoding is inspired by the model used in Header Space Analysis [10]. Section 3 discusses a more refined encoding that models hop-by-hop processing.

Formal reasoning. We now turn to formal reasoning problems and investigate whether the logical crossbar p_{net} correctly implements the specified forwarding and access control policies. It turns out that these questions, and many others, can be reduced to policy equivalence. We write $p \equiv q$ when p and q return the same set of packets on all inputs, and $p \leq q$ when p returns a subset of the packets returned by q on all inputs. Note that $p \leq q$ can be treated as an abbreviation for $p + q \equiv q$. To establish that p_{net} correctly filters all SSH packets, we check the following equivalence:

$$(\mathsf{typ} = \text{SSH}; p_{net}) \equiv \mathsf{drop}$$

To establish that p_{net} correctly forwards non-SSH packets from H_1 to H_2, we check the following inclusion:

$$\begin{aligned}&(\neg(\mathsf{typ} = \text{SSH}); \mathsf{sw} = A; \mathsf{pt} = 1; \mathsf{sw} \leftarrow B; \mathsf{pt} \leftarrow 2)\\ \leq\ &(\neg(\mathsf{typ} = \text{SSH}); \mathsf{sw} = A; \mathsf{pt} = 1; p_{net}; \mathsf{sw} = B; \mathsf{pt} = 2)\end{aligned}$$

and similarly for non-SSH packets H_2 to H_1. Lastly, to establish that *some* traffic can get from port 1 on switch A to port 2 on switch B, we check the following:

$$(\mathsf{sw} = A; \mathsf{pt} = 1; p_{net}; \mathsf{sw} = B; \mathsf{pt} = 2) \not\equiv \mathsf{drop}$$

Of course, to actually check these equivalences, we need a proof system. NetKAT is designed to not only be an expressive programming language, but also one that satisfies the axioms of a Kleene algebra with tests (KAT). Moreover, by extending KAT with additional axioms that capture the domain-specific features of networks, the equational theory is complete—*i.e.*, it can answer all the questions posed in this section, and many more. The following sections present the syntax, semantics, and equational theory of NetKAT formally (Section 3), prove that the equational theory is sound and complete with respect to the semantics (Section 4), and illustrate its effectiveness on a broad range of questions including additional reachability properties (Section 5), program isolation (Section 6) and compiler correctness (Section 7).

3. Netkat

This section defines the NetKAT syntax, semantics, and equational theory formally.

Preliminaries. A packet pk is a record with fields $f_1, \ldots, f_k$ mapping to fixed-width integers n. We assume a finite set of *packet headers*, including Ethernet and IP source and destination addresses, VLAN tag, TCP and UDP ports, along with special fields for the switch (sw), port (pt), and payload. For simplicity, we assume that every packet contains the same fields. We write $pk.f$ for the value in field f of pk, and $pk[f := n]$ for the packet obtained by updating field f of pk with n.

To facilitate reasoning about the paths a packet takes through the network, we maintain a *packet history* that records the state of each packet as it travels from switch to switch. Formally, a packet history h is a non-empty sequence of packets. We write $pk::\langle\rangle$ to denote a history with one element, $pk::h$ to denote the history constructed by prepending pk on to h, and $\langle pk_1, \ldots, pk_n\rangle$ for the history with elements pk_1 to pk_n. By convention, the first element of a history is the current packet; other elements represent older packets. We write H for the set of all histories, and $\mathcal{P}(\mathsf{H})$ for the powerset of H.

Syntax

Fields	f	$::=$	$f_1 \mid \cdots \mid f_k$
Packets	pk	$::=$	$\{f_1 = v_1, \cdots, f_k = v_k\}$
Histories	h	$::=$	$pk{::}\langle\rangle \mid pk{::}h$
Predicates	a, b, c	$::=$	id *Identity*
		$\mid$	drop *Drop*
		$\mid$	$f = n$ *Match*
		$\mid$	$a + b$ *Disjunction*
		$\mid$	$a; b$ *Conjunction*
		$\mid$	$\neg a$ *Negation*
Policies	p, q, r	$::=$	a *Filter*
		$\mid$	$f \leftarrow n$ *Modification*
		$\mid$	$p + q$ *Parallel composition*
		$\mid$	$p; q$ *Sequential composition*
		$\mid$	p^* *Kleene star*
		$\mid$	dup *Duplication*

Semantics

$$[\![p]\!] \in \mathsf{H} \to \mathcal{P}(\mathsf{H})$$

$$[\![\mathsf{id}]\!]\ h \triangleq \{h\}$$

$$[\![\mathsf{drop}]\!]\ h \triangleq \{\}$$

$$[\![f = n]\!]\ (pk{::}h) \triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$

$$[\![\neg a]\!]\ h \triangleq \{h\} \setminus ([\![a]\!]\ h)$$

$$[\![f \leftarrow n]\!]\ (pk{::}h) \triangleq \{pk[f := n]{::}h\}$$

$$[\![p + q]\!]\ h \triangleq [\![p]\!]\ h \cup [\![q]\!]\ h$$

$$[\![p; q]\!]\ h \triangleq ([\![p]\!] \bullet [\![q]\!])\ h$$

$$[\![p^*]\!]\ h \triangleq \bigcup_{i \in \mathbb{N}} F^i\ h$$

where $F^0\ h \triangleq \{h\}$ and $F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h$

$$[\![\mathsf{dup}]\!]\ (pk{::}h) \triangleq \{pk{::}(pk{::}h)\}$$

Kleene Algebra Axioms

$p + (q + r) \equiv (p + q) + r$	KA-Plus-Assoc
$p + q \equiv q + p$	KA-Plus-Comm
$p + \mathsf{drop} \equiv p$	KA-Plus-Zero
$p + p \equiv p$	KA-Plus-Idem
$p; (q; r) \equiv (p; q); r$	KA-Seq-Assoc
$\mathsf{id}; p \equiv p$	KA-One-Seq
$p; \mathsf{id} \equiv p$	KA-Seq-One
$p; (q + r) \equiv p; q + p; r$	KA-Seq-Dist-L
$(p + q); r \equiv p; r + q; r$	KA-Seq-Dist-R
$\mathsf{drop}; p \equiv \mathsf{drop}$	KA-Zero-Seq
$p; \mathsf{drop} \equiv \mathsf{drop}$	KA-Seq-Zero
$\mathsf{id} + p; p^* \equiv p^*$	KA-Unroll-L
$q + p; r \leq r \Rightarrow p^*; q \leq r$	KA-Lfp-L
$\mathsf{id} + p^*; p \equiv p^*$	KA-Unroll-R
$p + q; r \leq q \Rightarrow p; r^* \leq q$	KA-Lfp-R

Additional Boolean Algebra Axioms

$a + (b; c) \equiv (a + b); (a + c)$	BA-Plus-Dist
$a + \mathsf{id} \equiv \mathsf{id}$	BA-Plus-One
$a + \neg a \equiv \mathsf{id}$	BA-Excl-Mid
$a; b \equiv b; a$	BA-Seq-Comm
$a; \neg a \equiv \mathsf{drop}$	BA-Contra
$a; a \equiv a$	BA-Seq-Idem

Packet Algebra Axioms

$f \leftarrow n; f' \leftarrow n' \equiv f' \leftarrow n'; f \leftarrow n,$ if $f \neq f'$	PA-Mod-Mod-Comm
$f \leftarrow n; f' = n' \equiv f' = n'; f \leftarrow n,$ if $f \neq f'$	PA-Mod-Filter-Comm
$\mathsf{dup}; f = n \equiv f = n; \mathsf{dup}$	PA-Dup-Filter-Comm
$f \leftarrow n; f = n \equiv f \leftarrow n$	PA-Mod-Filter
$f = n; f \leftarrow n \equiv f = n$	PA-Filter-Mod
$f \leftarrow n; f \leftarrow n' \equiv f \leftarrow n'$	PA-Mod-Mod
$f = n; f = n' \equiv \mathsf{drop},$ if $n \neq n'$	PA-Contra
$\sum_i f = i \equiv \mathsf{id}$	PA-Match-All

Figure 2. NetKAT: syntax, semantics, and equational axioms.

KAT-Invariant	If $a; p \equiv p; a$ then $a; p^* \equiv a; (p; a)^*$	Lemma 2.3.2 in [14]
KAT-Sliding	$p; (q; p)^* \equiv (p; q)^*; p$	Identity 19 in [14]
KAT-Denesting	$p^*; (q; p^*)^* \equiv (p + q)^*$	Identity 20 in [14]
KAT-Commute	If for all atomic x in q, $x; p \equiv p; x$ then $q; p \equiv p; q$	Corollary of Lemma 4.4 in [2]

Figure 3. KAT theorems.

Syntax. Syntactically, NetKAT expressions are divided into two categories: predicates (a, b, c) and policies (p, q, r). Predicates include constants true (id) and false (drop), matches ($f = n$), and negation ($\neg a$), disjunction ($a + b$), and conjunction ($a; b$) operators. Policies include predicates, modifications ($f \leftarrow n$), parallel ($p + q$) and sequential ($p; q$) composition, iteration (p^*), and a special policy that records the current packet in the history (dup). The complete syntax of NetKAT is given in Figure 2. By convention, $(^*)$ binds tighter than $(;)$, which binds tighter than $(+)$. Hence, $a; b + c; d^*$ is the same as $(a; b) + (c; (d^*))$.

Semantics. Semantically, every NetKAT predicate and policy denotes a function that takes a history h and produces a (possibly empty) set of histories $\{h_1, \ldots, h_n\}$. Producing the empty set models dropping the packet (and its history); producing a singleton set models modifying or forwarding the packet to a single location; and producing a set with multiple histories models modifying the packet in several ways or forwarding the packet to multiple locations. Note that policies only ever inspect or modify the first (current) packet in the history. This means implementations need not actually record histories—they are only needed for reasoning.

Figure 2 defines the denotational semantics of NetKAT. There is no separate definition for predicates—every predicate is a policy, and the semantics of $(;)$ and $(+)$ are the same whether they are composing policies or predicates. The syntactic distinction between policies and predicates arises solely to ensure that negation is only applied to a predicate, and not, for example, to a policy such as p^*. Formally, a predicate denotes a function that returns either the singleton $\{h\}$ or the empty set $\{\}$ when applied to a history h. Hence, predicates behave like filters. A modification $(f \leftarrow n)$ denotes a function that returns a singleton history in which the field f of the current packet has been updated to n. Parallel composition $(p + q)$ denotes a function that produces the union of the sets generated by p and q, and sequential composition $(p; q)$ denotes the Kleisli composition $(\bullet)$ of the functions p and q, where the Kleisli composition of functions of type $\mathsf{H} \to \mathcal{P}(\mathsf{H})$ is defined as:

$$(f \bullet g)\, x \triangleq \bigcup \{g\, y \mid y \in f\, x\}\,.$$

Policy iteration p^* is interpreted as a union of semantic functions F_i of h, where each F_i is the Kleisli composition of function denoted by p i times. Finally, dup denotes a function that duplicates the current packet and adds it to the history. Since modification updates the packet at the head of the history, dup "freezes" the current state of the packet and makes it observable.

Equational theory. NetKAT, as its name suggests, is an extension of Kleene algebra with tests.

Formally, a *Kleene algebra* (KA) is an algebraic structure $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$, where K is an idempotent semiring under $(+, \cdot, 0, 1)$, and $p^* \cdot q$ (respectively $q \cdot p^*$) is the least solution of the affine linear inequality $p \cdot r + q \leq r$ (respectively $r \cdot p + q \leq r$), where $p \leq q$ is an abbreviation for $p + q = q$. The axioms of KA are listed in Figure 2. The axioms are shown in the syntax NetKAT, which uses the more suggestive names ;, drop, and id for $\cdot$, 0, and 1, respectively.

A *Kleene algebra with tests* (KAT) is a two-sorted algebra

$$(K,\ B,\ +,\ \cdot,\ ^*,\ 0,\ 1,\ \neg)$$

with $\neg$ a unary operator defined only on B, such that

- $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$ is a Kleene algebra,
- $(B,\ +,\ \cdot,\ \neg\ ,\ 0,\ 1)$ is a Boolean algebra, and

- $(B, +, \cdot, 0, 1)$ is a subalgebra of $(K, +, \cdot, 0, 1)$.

Elements of B and K are usually called *tests* and *actions* respectively; we will identify them with NetKAT predicates and policies. The axioms of Boolean algebra consist of the axioms of idempotent semirings (already listed as KA axioms) and the additional axioms listed in Figure 2.

It is easy to see that NetKAT has the required syntactic structure to be a KAT. However, the KAT axioms are not complete for the underlying NetKAT packet model. To establish completeness, we also need the packet algebra axioms listed in Figure 2. The first three axioms specify commutativity conditions. For example, PA-MOD-MOD-COMM states that assignments $\mathsf{src} \leftarrow X$ and $\mathsf{dst} \leftarrow Y$ can be applied in either order, as src and dst are different:

$$\mathsf{src} \leftarrow X; \mathsf{dst} \leftarrow Y \equiv \mathsf{dst} \leftarrow Y; \mathsf{src} \leftarrow X$$

Similarly, axiom PA-MOD-FILTER-COMM states that the assignment $\mathsf{src} \leftarrow X$ and predicate $\mathsf{sw} = A$ can be applied in either order. The axiom PA-DUP-FILTER-COMM states that every predicate commutes with dup. Interestingly, only this single axiom is needed to characterize dup in the equational theory. The next few axioms characterize modifications. The PA-MOD-FILTER axiom states that modifying a field f to n and then filtering on packets with f equal to n is equivalent to the modification alone. Similarly, the axiom PA-FILTER-MOD states that filtering on packets with field f equal to n and then modifying that field to n is equivalent to just the filter. PA-MOD-MOD states that only the last assignment in a sequence of assignments to the same f has any effect. The last two axioms characterize filters. The axiom PA-CONTRA states that a field cannot be equal to two different values at the same time. Finally, the axiom PA-MATCH-ALL states that the sum of filters on every possible value is equivalent to the identity. This implies packet values are drawn from a finite domain, such as fixed-width integers.

A simple example: access control. To illustrate the NetKAT equational theory, we prove a simple equivalence using the policies from Section 2. Recall that the policy p_A filtered SSH packets on switch A while p_B filtered SSH packets on switch B. We prove that these programs are equivalent on SSH traffic going from left to right across the network topology shown in Figure 1. This can be seen as a simple form of code motion—moving the filter from switch A to switch B. We use the logical crossbar encoding with input and output predicates defined as follows:

$$in \triangleq (\mathsf{sw} = A; \mathsf{typ} = \mathrm{SSH}) \qquad \text{and} \qquad out \triangleq (\mathsf{sw} = B)$$

The proof, given in Figure 4, is a straightforward calculation using the equational axioms and some standard KAT theorems, listed in Figure 3. The shaded term on each line indicates a term that will be changed in the next proof step. To lighten the notation we write s_A for $(\mathsf{sw} = A)$ and similarly for s_B, and SSH for $(\mathsf{typ} = \mathrm{SSH})$.

4. Soundness, Completeness, and Decidability

This section proves the soundness and completeness of the NetKAT axioms with respect to the denotational semantics defined in Section 3. That is, every equation provable from the axioms of NetKAT also holds in the denotational model (Theorem 1) and every equality that holds in the denotational model is provable using the axioms (Theorem 2). In addition, we establish the decidability of NetKAT equivalence, and show that the algorithm is PSPACE-complete.

To accomplish this, we prove results that are much stronger and more enlightening from a theoretical point of view. For soundness, we show that the packet-history model used in the denotational semantics is isomorphic to another model based on binary relations, and appeal to the soundness of KAT over binary relation models. That elements of our packet-history model can be represented as binary relations is not particularly surprising. What is more surprising is that one can also formulate language models that play the same role in NetKAT as the regular sets of strings in KA or the regular sets of guarded strings in KAT. These models are the free KA and KAT, respectively, on their generators, and the same holds for our NetKAT language model. Relating packet-history models and language models plays a crucial role in the proof given here—it allows us to leverage the completeness of KA.

$in; (p_A; t)^*; p_A; out$

$\equiv$ { definition in, out, and p_A }

$s_A; \text{ssh}; ((s_A; \neg\text{ssh}; p + s_B; p); t)^*; p_A; s_B$

$\equiv$ { KAT-Invariant }

$s_A; \text{ssh}; ((s_A; \neg\text{ssh}; p + s_B; p); t; \text{ssh})^*; p_A; s_B$

$\equiv$ { KA-Seq-Dist-R }

$s_A; \text{ssh}; (s_A; \neg\text{ssh}; p; t; \text{ssh} + s_B; p; t; \text{ssh})^*; p_A; s_B$

$\equiv$ { KAT-Commute }

$s_A; \text{ssh}; (s_A; \neg\text{ssh}; \text{ssh}; p; t + s_B; p; t; \text{ssh})^*; p_A; s_B$

$\equiv$ { BA-Contra }

$s_A; \text{ssh}; (s_A; \text{drop}; p; t + s_B; p; t; \text{ssh})^*; p_A; s_B$

$\equiv$ { KA-Seq-Zero, KA-Zero-Seq,KA-Plus-Comm, KA-Plus-Zero }

$s_A; \text{ssh}; (s_B; p; t; \text{ssh})^*; p_A; s_B$

$\equiv$ { KA-Unroll-L }

$s_A; \text{ssh}; (\text{id} + (s_B; p; t; \text{ssh}); (s_B; p; t; \text{ssh})^*); p_A; s_B$

$\equiv$ { KA-Seq-Dist-L and KA-Seq-Dist-R }

$(s_A; \text{ssh}; p_A; s_B)+$

$(s_A; \text{ssh}; s_B; p; t; \text{ssh}; (s_B; p; t; \text{ssh})^*; p_A; s_B)$

$\equiv$ { KAT-Commute }

$(s_A; s_B; \text{ssh}; p_A)+$

$(s_A; s_B; \text{ssh}; p; t; \text{ssh}; (s_B; p; t; \text{ssh})^*; p_A; s_B)$

$\equiv$ { PA-Contra }

$(\text{drop}; \text{ssh}; p_A)+$

$(\text{drop}; \text{ssh}; p; t; \text{ssh}; (s_B; p; t; \text{ssh})^*; p_A; s_B)$

$\equiv$ { KA-Zero-Seq, KA-Plus-Idem }

drop

$\equiv$ { KA-Seq-Zero, KA-Zero-Seq, KA-Plus-Idem }

$s_A; (p_B; t)^*; (\text{ssh}; \text{drop}; p + s_B; \text{drop}; p; s_B)$

$\equiv$ { PA-Contra and BA-Contra }

$s_A; (p_B; t)^*; (\text{ssh}; s_A; s_B; p + s_B; \text{ssh}; \neg\text{ssh}; p; s_B)$

$\equiv$ { KAT-Commute }

$s_A; (p_B; t)^*; (\text{ssh}; s_A; p; s_B + \text{ssh}; s_B; \neg\text{ssh}; p; s_B)$

$\equiv$ { KA-Seq-Dist-L and KA-Seq-Dist-R }

$s_A; (p_B; t)^*; \text{ssh}; (s_A; p + s_B; \neg\text{ssh}; p); s_B$

$\equiv$ { KAT-Commute }

$s_A; \text{ssh}; (p_B; t)^*; (s_A; p + s_B; \neg\text{ssh}; p); s_B$

$\equiv$ { definition in, p_B, and out }

$in; (p_B; t)^*; p_B; out$

Figure 4. Access control code motion proof.

4.1 Soundness

To prove soundness, it is helpful to reformulate the standard packet-history semantics introduced in Section 3 in terms of binary relations. In the standard semantics, policies and predicates are modeled as functions $[\![p]\!] : \mathsf{H} \to \mathcal{P}(\mathsf{H})$. This semantics is isomorphic to a relational semantics $[\cdot]$ in which each policy and predicate

is interpreted as a binary relation $[p] \subseteq \mathsf{H} \times \mathsf{H}$:

$$(h_1, h_2) \in [p] \;\Leftrightarrow\; h_2 \in [\![p]\!]\,(h_1).$$

Intuitively $[p]$ is the set of input-output pairs of the policy p.

Formally, the maps $[\![p]\!] : \mathsf{H} \to \mathcal{P}(\mathsf{H})$ are morphisms of type $\mathsf{H} \to \mathsf{H}$ in $\mathsf{Kl}\,\mathcal{P}$, the Kleisli category of the powerset monad. It is well known that the Kleisli category $\mathsf{Kl}\,\mathcal{P}$ is isomorphic to the category Rel of sets and binary relations, as witnessed by currying:

$$X \to \mathcal{P}(Y) \cong X \to Y \to \mathbf{2} \cong X \times Y \to \mathbf{2} \cong \mathcal{P}(X \times Y).$$

In the relational model $[\cdot]$, product is interpreted as ordinary relational composition, and the remaining KAT operations translate under the isomorphism to the usual KAT operations on binary relations. Since the relational model with these distinguished operations satisfies the axioms of KAT (see *e.g.* [14, 17]), so do NetKAT models with the packet-history semantics of Section 3.

Let $\vdash$ denote provability in NetKAT. We are now ready to prove the soundness of the KAT axioms.

Theorem 1 (Soundness)**.** *The* KAT *axioms with the equational premises of Figure 2 are sound with respect to the semantics of Section 3. That is, if* $\vdash p \equiv q$*, then* $[\![p]\!] = [\![q]\!]$.

Proof sketch. We have already argued that any packet history model is isomorphic to a relational KAT, therefore satisfies all the axioms of KAT listed in Figure 2. It remains to show that the specialized NetKAT axioms on the right-hand side of Figure 2 are satisfied. These can all be verified by elementary arguments in relational algebra (see *e.g.* [26]). Some are special cases of [2, Equations (6)–(11)], whose soundness is proved in [2, Theorem 4.3]. See Appendix A for the full proof. □

4.2 Completeness

The proof of completeness proceeds in four steps:

1. We define *reduced Net*KAT, a subset of NetKAT where policies are a regular expression over *atoms* (a normal form for sequences of tests), *complete assignments* (a normal form for sequences of modifications), and dup. We show that every NetKAT policy is provably equivalent to a reduced NetKAT policy and that reduced terms have a simplified set of axioms pertaining to assignments and tests.
2. Inspired by past proofs of completeness for KA and KAT, we develop a *language model* for reduced NetKAT. This language model gives semantics to policies via regular sets of I, a set of guarded join-irreducible strings. We prove the language model and the standard model of NetKAT are isomorphic.
3. We define normal forms for NetKAT and show that every reduced NetKAT policy is provably equivalent to its normal form.
4. We prove completeness by relating the language model to the NetKAT normal forms.

The rest of this section outlines the major steps of the proof.

Step 1: reduced NetKAT. Let $f_1, \ldots, f_k$ be a list of all fields of a packet in some (fixed) order. For each tuple $\bar{n} = n_1, \ldots, n_k$ of values, let $\bar{f} = \bar{n}$ and $\bar{f} \leftarrow \bar{n}$ denote the expressions

$$f_1 = n_1; \cdots; f_k = n_k \qquad\qquad f_1 \leftarrow n_1; \cdots; f_k \leftarrow n_k,$$

respectively. The former expression is called an *atom* and the latter a *complete assignment*. The terminology *atom* comes from the fact that it is an atomic (minimal nonzero) element of the Boolean algebra generated by the primitive tests $f = n$. Note that the atoms and complete assignments are in one-to-one correspondence according to the values $\bar{n}$. Hence, if α is an atom, we denote the corresponding complete assignment by π_α,

Syntax

$$\begin{aligned}
\text{Atoms} \quad & \alpha, \beta \triangleq f_1 = n_1; \cdots; f_k = n_k \\
\text{Assignments} \quad & \pi \triangleq f_1 \leftarrow n_1; \cdots; f_k \leftarrow n_k \\
\text{Join-irreducibles} \quad & x, y \in At; (P; \mathsf{dup})^*; P
\end{aligned}$$

Simplified axioms for At and P

$$\pi \equiv \pi; \alpha_\pi \quad \alpha; \mathsf{dup} \equiv \mathsf{dup}; \alpha \quad \sum_\alpha \alpha \equiv \mathsf{id},$$

$$\alpha \equiv \alpha; \pi_\alpha \qquad \pi; \pi' \equiv \pi' \qquad \alpha; \beta \equiv \mathsf{drop},\ \alpha \neq \beta$$

Join-irreducible concatenation

$$\alpha; p; \pi \diamond \beta; q; \pi' = \begin{cases} \alpha; p; q; \pi' & \text{if } \beta = \alpha_\pi \\ \text{undefined} & \text{if } \beta \neq \alpha_\pi \end{cases}$$

$$A \diamond B = \{p \diamond q \mid p \in A,\ q \in B\} \subseteq I$$

Regular Interpretation: $R(p) \subseteq (P + At + \mathsf{dup})^*$

$$\begin{aligned}
R(\pi) &= \{\pi\} \\
R(p+q) &= R(p) \cup R(q) \\
R(\alpha) &= \{\alpha\} \\
R(p;q) &= \{xy \mid x \in R(p), y \in R(q)\} \\
R(\mathsf{dup}) &= \{\mathsf{dup}\} \\
R(p^*) &= \bigcup_{n \geq 0} R(p^n)
\end{aligned}$$

with $p^0 = 1$ and $p^{n+1} = p; p^n$

Language Model: $G(p) \subseteq I = At; (P; \mathsf{dup})^*; P$

$$\begin{aligned}
G(\pi) &= \{\alpha; \pi \mid \alpha \in At\} \\
G(p+q) &= G(p) \cup G(q) \\
G(\alpha) &= \{\alpha; \pi_\alpha\} \\
G(p;q) &= G(p) \diamond G(q) \\
G(\mathsf{dup}) &= \{\alpha; \pi_\alpha; \mathsf{dup}; \pi_\alpha \mid \alpha \in At\} \\
G(p^*) &= \bigcup_{n \geq 0} G(p^n)
\end{aligned}$$

Figure 5. Completeness definitions.

and if π is a complete assignment, we denote the corresponding atom by α_π. We let At denote the set of atoms and P the set of complete assignments.

Now that we have defined atoms and complete assignments, we investigate their properties. The left-hand side of Figure 5 does so via a series of simple axioms. Each axiom is easily provable using the full NetKAT axioms. A useful consequence of the axioms is that the sum of corresponding atoms and complete assignments is equivalent to the identity:

$$\textstyle\sum_{\alpha \in At} \alpha; \pi_\alpha \equiv \mathsf{id}$$

Now, the proof that any policy is equivalent to a policy in which all atomic assignments $f \leftarrow n$ appear in the context of a complete assignment is straight-forward.

$$\begin{aligned}
f \leftarrow n &\equiv \mathsf{id}; f \leftarrow n \\
&\equiv (\textstyle\sum_{\alpha \in At} \alpha; \pi_\alpha); (f \leftarrow n) \\
&\equiv \textstyle\sum_{\alpha \in At} \alpha; \pi'_\alpha
\end{aligned}$$

where π'_α is π_α with the assignment to f replaced by $f \leftarrow n$. Similarly, all tests are equivalent to sums of atoms:

$$b \equiv \textstyle\sum_{\alpha \leq b} \alpha$$

Since all modifications can be replaced by complete assignments and all tests by atoms, any NetKAT policy p may be viewed as a regular expression over the alphabet $P \cup At \cup \{\mathsf{dup}\}$. The top right of Figure 5 shows this by defining a mapping R from reduced NetKAT to regular sets.

Step 2: language model. Both KA and KAT have a language model in which each expression is interpreted as a regular set of join-irreducible (minimal non-zero) terms. For KA, the language model is the KA of regular sets

of strings, and for KAT, it is the KAT of regular sets of guarded strings [17]. NetKAT also has a language model. It consists of regular subsets of a restricted class of guarded strings $I = At; (P; \mathsf{dup})^*; P$. Each string drawn from this set has the form

$$\alpha; \pi_0; \mathsf{dup}; \pi_1; \mathsf{dup}; \cdots ; \mathsf{dup}; \pi_n \tag{4.1}$$

for some $n \geq 0$. These strings represent the join-irreducible elements of the standard model of NetKAT.

The bottom right of Figure 5 defines the language model as a mapping G from reduced NetKAT expressions to regular subsets of I. The case for sequential composition makes use of the concatenation operator ($\diamond$) over strings from I, which we lift to concatenation of sets of strings from I. Both definitions appear on the bottom left of Figure 5.

Note that $\diamond$ is partial on strings but total on sets. Using the simplified axioms of Fig. 5, it is easily shown that $\diamond$ on strings is associative and on sets is associative, distributes over union, and has two-sided identity $\{\alpha; \pi_\alpha \mid \alpha \in At\}$. Also note that if $\alpha; p; \pi \diamond \beta; q; \pi'$ exists, then $\vdash \alpha; p; \pi; \beta; q; \pi' \equiv \alpha; p; \pi \diamond \beta; q; \pi'$ and $\alpha; p; \pi \diamond \beta; q; \pi' \in I$, and otherwise $\vdash \alpha; p; \pi; \beta; q; \pi' \equiv \mathsf{drop}$.

Given our newly defined language model, the next step is to show that it is isomorphic to the standard packet model presented in Section 3. We first show that every expression is the join of its join-irreducibles,[1] using a straightforward proof by induction on p.

Lemma 1. *For all reduced policies p, we have $[\![p]\!] = \bigcup_{x \in G(p)} [\![x]\!]$.*

Next we show that every x in I is completely determined by $[\![x]\!]$.

Lemma 2. *If $x, y \in I$, then $[\![x]\!] = [\![y]\!]$ if and only if $x = y$.*

Finally, using Lemmas 1 and 2, we conclude that the language model is isomorphic to the denotational model presented earlier.

Lemma 3. *For all reduced NetKAT policies p and q, we have $[\![p]\!] = [\![q]\!]$ if and only if $G(p) = G(q)$.*

Step 3: NetKAT normal forms. A policy is *guarded* if it is of the form $\alpha; \pi; x; \pi'$ or $\alpha; \pi$. An expression p is in *normal form* if it is a sum of zero or more guarded terms and $R(p) \subseteq I$. A policy is *normalizable* if it is equivalent to a policy in normal form.

Lemma 4. *Every policy is normalizable.*

Proof. The proof proceeds by induction on the structure of policies. To begin, the primitive symbols are normalizable:

$$\begin{array}{lcl} h \leftarrow n & \equiv & \sum_{\alpha \in At} \alpha; \pi'_\alpha \\ \mathsf{dup} & \equiv & \sum_{\alpha \in At} \alpha; \pi_\alpha; \mathsf{dup}; \pi_\alpha \\ b & \equiv & \sum_{\alpha \leq b} \alpha; \pi_\alpha \end{array}$$

Next, we show that sums and products of normalizable expressions are normalizable. The case for sums is trivial, and the case for products follows by a simple argument:

$$\left(\sum_i s_i\right); \left(\sum_j t_j\right) \equiv \sum_i \sum_j s_i; t_j \equiv \sum_i \sum_j s_i \diamond t_j.$$

The most interesting case is for Kleene star. Consider an expression p^*, where p is in normal form. We first prove the uniform case: when all guarded terms in the sum p have the same initial atom α, that is, $p = \alpha; t$ where t is a sum of terms each with a leading and trailing π, and $R(t) \subseteq P; (\mathsf{dup}; P)^*$. Let u be t with all terms

[1] We abuse notation slightly here by applying the union operator $\bigcup$ to functions $\mathsf{H} \to \mathcal{P}(\mathsf{H})$. This is interpreted pointwise: $\bigcup [\![p]\!] = \lambda s . \bigcup [\![p]\!] (s)$.

whose trailing π is not π_α deleted and with the trailing π_α deleted from all remaining terms. By the simplified axioms of Figure 5, we have $t;\alpha;t \equiv u;t$, therefore $t;\alpha;t;\alpha \equiv u;t;\alpha$. Using [2, Lemma 4.4],

$$\begin{aligned}(t;\alpha)^*;t &\equiv t + t;\alpha;(t;\alpha)^*;t \\ &\equiv t + u^*;t;\alpha;t \\ &\equiv t + u^*;u;t \\ &\equiv u^*;t,\end{aligned}$$

and hence

$$\begin{aligned}p^* &\equiv \mathsf{id} + p^*;p \\ &\equiv \mathsf{id} + (\alpha;t)^*;\alpha;t \\ &\equiv \mathsf{id} + \alpha;(t;\alpha)^*;t \\ &\equiv \mathsf{id} + \alpha;u^*;t \\ &\equiv \mathsf{id} + \alpha;t + \alpha;u;u^*;t,\end{aligned}$$

which after normalizing the id is in normal form.

Finally, for the case p^* where the initial tests in the sum p are not uniform, the argument is by induction on the number of terms in the sum. If $p = \alpha;x + q$, then by induction hypothesis, q^* has an equivalent normal form $\hat{q}^*$. Using KAT-Denesting (Figure 3), we obtain

$$p^* \equiv (\alpha;x+q)^* \equiv q^*;(\alpha;x;q^*)^* \equiv \hat{q}^*;(\alpha;x;\hat{q}^*)^*,$$

and then proceed as in the previous case. □

Step 4: Completeness. We need just two further lemmas before delivering the completeness result that relates R and G. Lemma 5 states that $G(p)$ is equal to the join of the elements of $R(p)$. The proof is by induction on the structure of p

Lemma 5.

$$G(p) = \bigcup_{x \in R(p)} G(x).$$

Next, Lemma 6 shows that $R(p) = G(p)$ if $R(p)$ is a subset of the join-irreducibles. The proof is straightforward using Lemma 5.

Lemma 6. *If $R(p) \subseteq I$, then $R(p) = G(p)$.*

Proof. Suppose $R(p) \subseteq I$. Since $G(x) = \{x\}$ for $x \in I$, by Lemma 5 we have

$$G(p) = \bigcup_{x \in R(p)} G(x) = \bigcup_{x \in R(p)} \{x\} = R(p).$$ □

The proof of completeness for NetKAT now follows from our lemmas and completeness of KA [12].

Theorem 2 (Completeness)**.** *Every semantically equivalent pair of expressions is provably equivalent in NetKAT. That is, if $[\![p]\!] = [\![q]\!]$, then $\vdash p \equiv q$.*

Proof. Let $\hat{p}$ and $\hat{q}$ be the normal forms of p and q. By Lemma 4, we can prove that each is equivalent to its normal form: $\vdash p \equiv \hat{p}$ and $\vdash q \equiv \hat{q}$. By soundness we have $[\![p]\!] = [\![\hat{p}]\!]$ and $[\![q]\!] = [\![\hat{q}]\!]$. Hence $[\![\hat{p}]\!] = [\![\hat{q}]\!]$ by transitivity. By Lemma 3, we have $G(\hat{p}) = G(\hat{q})$. Moreover, by Lemma 6, we have $G(\hat{p}) = R(\hat{p})$ and $G(\hat{q}) = R(\hat{q})$, thus $R(\hat{p}) = R(\hat{q})$ by transitivity. Since $R(\hat{p})$ and $R(\hat{q})$ are regular sets, by the completeness of KA, we also have $\vdash \hat{p} \equiv \hat{q}$. Finally, as $\vdash p \equiv \hat{p}$ and $\vdash q \equiv \hat{q}$ and $\vdash \hat{p} \equiv \hat{q}$, by transitivity we conclude that $\vdash p \equiv q$, as required. □

4.3 Decidability

The following theorem shows that the deciding the equational theory of NetKAT is no more nor less difficult than KA or KAT.

Theorem 3. *The equational theory of Net*KAT *is* PSPACE*-complete.*

Proof sketch. To show PSPACE-hardness, reduce KA to NetKAT as follows. Let Σ be a finite alphabet. For a regular expression e over Σ, let $R(e)$ be the regular set of strings over Σ as defined in §4. Transform e to a NetKAT expression e' by replacing each occurrence in e of a symbol $p \in \Sigma$ with $(p; \mathsf{dup})$ and prepending with an arbitrary but fixed atom α. It follows from Lemmas 3 and 6 that $R(e_1) = R(e_2)$ if and only if $R(e'_1) = R(e'_2)$ if and only if $G(e'_1) = G(e'_2)$ if and only if $[\![e'_1]\!] = [\![e'_2]\!]$.

To show that the problem is in PSPACE, given two NetKAT expressions e_1 and e_2, we know that $[\![e_1]\!] \neq [\![e_2]\!]$ if and only if there is a packet pk and packet history h such that $h \in [\![e_1]\!](pk) \setminus [\![e_2]\!](pk)$ or $h \in [\![e_2]\!](pk) \setminus [\![e_1]\!](pk)$; let us say the former without loss of generality. We guess pk nondeterministically and follow a nondeterministically-guessed trajectory through e_1 that produces some $h \in [\![e_1]\!](pk)$. At the same time, we trace all possible trajectories through e_2 that could generate a prefix of h, ensuring that none of these produce $h \in [\![e_2]\!]$. It takes only polynomial space to represent the current values of the fields of the head packet and the possible positions in e_2 for the current prefix of h. The algorithm is nondeterministic, but can be made deterministic using Savitch's theorem. □

5. Reachability Properties

One of the most common networking problems occurs when a configuration is changed and two hosts can no longer communicate for some reason or another. To diagnose such problems, network operators often need to reason about *reachability properties*. This section shows how to encode two common reachability properties as succinct NetKAT equations. Moreover, we prove that our equations are equivalent to intuitive definitions of reachability queries defined in the packet-history semantics by appealing to the language model defined in the previous section.

Reachability. The simplest kind of reachability property is just *"Can host A send packets to host B?"*. If satisfiable, then the policy p and network topology t will not drop every packet that A sends to B. More generally, we can ask if a set of packets satisfying some predicate a—*e.g.*, all packets located at a specified group of hosts, or all packets of a certain type—can transition to a state where they satisfy some predicate b.

To reason about reachability, however, we need a slightly different model of our network. Specifically, we want to be able to observe the packet at each step that it takes through the network. To model hop-by-hop processing through a network, we use a policy of the form:

$$in; \mathsf{dup}; (p; t; \mathsf{dup})^*; out$$

Here the packet state is recorded on entry to the network and after every hop. In particular, this model distinguishes between switch policies that forward all packets to the same network egresses, but do so along different paths.

Now that we can observe the packet and build up a suitable hop-by-hop packet history, we define reachability as follows:

Definition 1 (Reachability)**.** *We say b is reachable from a if and only if there exists a trace $\langle pk_1, \cdots, pk_n \rangle \in \mathsf{rng}([\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$ such that $[\![a]\!] \langle pk_n \rangle = \{\langle pk_n \rangle\}$ and $[\![b]\!] \langle pk_1 \rangle = \{\langle pk_1 \rangle\}$.*

To decide if b is reachable from a we need only determine the validity of the following NetKAT equation:

$$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \not\equiv \mathsf{drop}$$

Intuitively, the expression $\mathsf{dup};(p;t;\mathsf{dup})^*$ denotes the set of all histories generated by the policy and topology. Prefixing this set with a and postfixing it with b filters the set to histories that begin with packets satisfying a and end with packets satisfying b. We would like to show that this equation corresponds to the semantic notion of reachability in Definition 1. The key to the proof is to translate both the denotational definition of reachability and the above equation into the language model, where they are easy to relate to one another.

Theorem 4 (Reachability Correctness)**.** *For predicates a and b, policy p, and topology t, $a;\mathsf{dup};(p;t;\mathsf{dup})^*;b \not\equiv \mathsf{drop}$, if and only if b is reachable from a.*

Proof. First we translate the NetKAT term into the language model:

$$\begin{array}{ll} & a;\mathsf{dup};(p;t;\mathsf{dup})^*;b \not\equiv \mathsf{drop} \\ \Rightarrow & \exists \alpha\, \pi_n \cdots \pi_1, \\ & \quad \alpha;\pi_n;\mathsf{dup};\cdots;\mathsf{dup};\pi_1 \in G(a;\mathsf{dup};(p;t;\mathsf{dup})^*;b) \end{array}$$

We also translate each term in the semantic definition of reachability into the language model:

$$\begin{array}{ll} & \exists pk_1 \cdots pk_n. \\ & \quad \langle pk_1, \cdots, pk_n\rangle \in \mathsf{rng}(\,[\![\mathsf{dup};(p;t;\mathsf{dup})^*]\!]), \\ & \quad [\![a]\!]\,\langle pk_n\rangle = \{\langle pk_n\rangle\} \text{ and} \\ & \quad [\![b]\!]\,\langle pk_1\rangle = \{\langle pk_1\rangle\} \\ \Rightarrow & \exists \pi'_1 \cdots \pi'_m. \\ & \quad \alpha_{\pi'_m};\pi'_m;\mathsf{dup};\cdots;\mathsf{dup};\pi'_1 \in G(\mathsf{dup};(p;t;\mathsf{dup})^*), \\ & \quad \alpha_{\pi'_m};\pi'_m \in G(a) \text{ and} \\ & \quad \alpha_{\pi'_1};\pi'_1 \in G(b) \end{array}$$

Next, we prove each direction by setting $\alpha = \alpha_{\pi_n}$ and $m = n$. For example, for soundness we need to show:

$$\begin{array}{rlcl} & \alpha;\pi_n;\mathsf{dup};\cdots;\mathsf{dup};\pi_1 & \in & G(a;\mathsf{dup};(p;t;\mathsf{dup})^*;b) \\ \Rightarrow & \alpha_{\pi'_m};\pi'_m;\mathsf{dup};\cdots;\mathsf{dup};\pi'_1 & \in & G(\mathsf{dup};(p;t;\mathsf{dup})^*) \end{array}$$

This holds by the definition of $\diamond$. Completeness is similar. □

Waypointing. Another important kind of reachability property is waypointing. Intuitively, waypointing properties ensure that every packet traverses a specified node such as a firewall or intrusion-detection middlebox. More formally, we say that W is a waypoint from A to B if all paths from A to B go through W. Phrased in terms of the denotational semantics, W is a waypoint if, for all histories where B occurs after A, W appears between A and B.

Definition 2 (Waypoint)**.** *For predicates a, b, and w, packets from a to b are* waypointed *through w if and only if for all packet histories $\langle pk_1 \cdots pk_n\rangle \in \mathsf{rng}(\,[\![\mathsf{dup};(p;t;\mathsf{dup})^*]\!])$ where $[\![a]\!]\,\langle pk_n\rangle = \{\langle pk_n\rangle\}$ and $[\![b]\!]\,\langle pk_1\rangle = \{\langle pk_1\rangle\}$, there exists $pk_x \in \langle pk_1 \cdots pk_n\rangle$ such that $[\![w]\!]\,\langle pk_x\rangle = \{\langle pk_x\rangle\}$ and for all i where $1 < i < x$, we have $[\![b]\!]\ pk_i = \{\}$, and for all i where $x < i < n$, we have $[\![a]\!]\ pk_i = \{\}$.*

We can decide if w is a waypoint between a and b by checking the following NetKAT equation:

$$a;\mathsf{dup};(p;t;\mathsf{dup})^*;b \leq a;(\neg b;p;t;\mathsf{dup})^*;w;(\neg a;p;t;\mathsf{dup})^*;b$$

The left-hand side of this relation is exactly reachability. The right-hand side splits the network expression into two pieces—the first ensures that packets do not prematurely visit b before the waypoint, w, and the second ensures that packets do not return to a after the waypoint, if they are going to evetually reach b.

To prove the test sound and complete, we again translate both the test and the denotational semantics into the language model, and prove the following theorem:

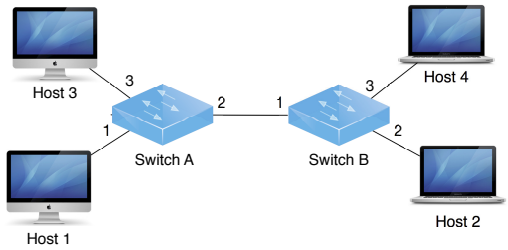

Figure 6. A simple network controlled by two parties.

Theorem 5 (Waypoint Correctness). *For predicates a, b, and w,*

$$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \leq a; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b$$

if and only if all packets from a to b are waypointed through w.

Proof. The proof is similar to the reachability proof above. Complete proofs of each direction are included in the long version of this paper. □

Using these encodings and theorems as building blocks, we can write reachability-checking equations for other properties as well. For example, we can check for self-loops, test a firewall, and even string together multiple waypoints into composite tests.

6. Traffic Isolation

NetKAT's policy combinators help programmers construct rich network policies out of simpler parts. The most obvious combinator is parallel composition, which combines two smaller policies into one that, intuitively, provides the "union" of both behaviors. But naive use of parallel composition can lead to undesirable results due to interference between the packets generated by each sub-policy.

Example. Consider the network in Figure 6. Now, suppose the task of routing traffic between hosts 1 and 2 has been assigned to one programmer, while the task of routing traffic between hosts 3 and 4 has been assigned to another programmer. The first programmer might produce the following policy for switch B,

$$p_{B1} \triangleq \mathsf{sw} = B; (\mathsf{pt} = 1; \mathsf{pt} \leftarrow 2 + \mathsf{pt} = 2; \mathsf{pt} \leftarrow 1)$$

and the other programmer might produce a similar switch policy for B. This second policy differs from the first only by sending traffic from port 1 out port 3 rather than port 2:

$$p_{B2} \triangleq \mathsf{sw} = B; (\mathsf{pt} = 1; \mathsf{pt} \leftarrow 3 + \mathsf{pt} = 3; \mathsf{pt} \leftarrow 1)$$

Similar policies p_{A1} and p_{A2} define the behavior at switch A. Assume a topology assertion t that captures the topology of the network. By itself, the program

$$((p_{A1} + p_{B1}); t)^*$$

correctly sends traffic from host 1 to host 2. But when the second policy is added in,

$$(((p_{A1} + p_{B1}) + (p_{A2} + p_{B2})); t)^*$$

packets sent from host 1 will be copied to host 4 as well as host 2. In this instance, parallel composition actually produces *too many behaviors*. In the best case, sending additional packets to host 4 from host 1 leads to network congestion. In the worst case, it may violate the security policy for host 1. Either case demonstrates the need for better ways of composing forwarding policies.

Slices. A *network slice* [8] is a lightweight abstraction that facilitates modular construction of routing policies. Intuitively, a slice defines a piece of the network that may be programmed independently. A slice comes with ingress (in) and egress (out) predicates, which define its boundaries, as well as an internal policy p. Packets that match in are injected into the slice. Once in a slice, packets stay in the slice and obey p until they match the predicate out, at which point they are ejected. We write slices as follows:

$$\{in\}\ w : (p)\ \{out\}$$

where in and out are the ingress and egress predicates and p defines the internal policy. Each slice also has a unique identifier w to differentiate it from other slices in a network program.[2]

It is easy to define slices by elaboration into NetKAT. We first create a new header field tag to record the slice to which the packet currently belongs.[3] In order for our elaboration to have the desired properties, however, the tag field must not be used elsewhere in the policy or in the ingress or egress predicates. We call a predicate *tag-free* if it commutes with any modification of the tag field, and a policy *tag-free* if it commutes with any test of the tag field.

Given tag-free predicates in, out and policy p, and a tag w_0 representing packets not in any slice, we can compile a slice into NetKAT as follows:

$$\begin{aligned}&(\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0} \triangleq \\ &\quad \mathsf{let}\ pre = (\mathsf{tag} = w_0; in; \mathsf{tag} \leftarrow w + \mathsf{tag} = w)\ \mathsf{in} \\ &\quad \mathsf{let}\ post = (out; \mathsf{tag} \leftarrow w_0 + \neg out)\ \mathsf{in} \\ &\quad (pre; p; post)\end{aligned}$$

Slice compilation wraps the slice policy with pre- and post-processing policies, pre and $post$. The pre policy tests whether a packet (i) is outside the slice (tagged with w_0) and matches the ingress predicate, in which case it is injected by tagging it with w, or (ii) has already been injected (already tagged with w). Once injected, packets are processed by p. If p emits a packet that matches the egress predicate out, then $post$ strips the tag, restoring w_0. Otherwise, the packet remains in the slice and is left unmodified.

Isolation. A key property of slices is that once a packet enters a slice, it is processed solely by the slice policy until it is ejected, even across multiple hops in the topology.

Theorem 6 (Slice/Slice Composition)**.** *For all* tag*-free slice ingress and egress predicates* in_1, out_1, in_2, out_2*, identifiers* w_1, w_2*, policies* s_1, s_2*,* tag*-free policies* p_1, p_2*, and topologies* t*, such that*

- $s_1 = (\!|\{in_1\}\ w_1 : (p_1)\ \{out_1\}|\!)^{w_0}$*,*
- $s_2 = (\!|\{in_2\}\ w_2 : (p_2)\ \{out_2\}|\!)^{w_0}$*,*
- *H0:* $w_1 \neq w_2$*,* $w_1 \neq w_0$*,* $w_2 \neq w_0$
- *H1:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$*, and*
- *H2:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$*, then*

$$((s_1 + s_2); t; \mathsf{dup})^* \equiv (s_1; t; \mathsf{dup})^* + (s_2; t; \mathsf{dup})^*.$$

If the preconditions are met, including that the two slices have different identifiers and that the slice boundaries are disjoint, then the theorem says that pushing a packet through a network executing s_1 and s_2 in parallel is the same as copying that packet and pushing it through two separate copies the network, each containing one of the slices. The proof of the theorem is by equational reasoning and makes use of the KAT-Denesting (see Figure 3).

An interesting corollary of the result above is that when the slice boundaries between s_1 and s_2 do not overlap, and one restricts attention to traffic destined for the ingress of s_1, running s_1 in parallel with s_2 is equivalent to running s_1 alone.

[2] The unique identifier w may be defined by the compiler and need not actually appear in the surface syntax.

[3] In practice, the vlan field is often used to differentiate different classes of network traffic [31].

Corollary 1. *For all* tag*-free slice ingress and egress predicates* in_1, out_1, in_2, out_2*, identifiers* w_1, w_2*, policies* s_1, s_2*,* tag*-free policies* p_1, p_2*, and topologies* t*, such that*

- $s_1 = (\!|\{in_1\}\ w_1 : (p_1)\ \{out_1\}|\!)^{w_0}$,
- $s_2 = (\!|\{in_2\}\ w_2 : (p_2)\ \{out_2\}|\!)^{w_0}$,
- *H0:* $w_1 \neq w_2, w_1 \neq w_0, w_2 \neq w_0$
- *H1:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$,
- *H2:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$,
- *H3:* $in_1; in_2 \equiv \mathsf{drop}$, *then*

$$\begin{aligned} & in_1; \mathsf{tag} = w_0; ((s_1 + s_2); t; \mathsf{dup})^* \\ \equiv\ & in_1; \mathsf{tag} = w_0; (s_1; t; \mathsf{dup})^* \end{aligned}$$

Looking closely at Corollary 1, one can see a connection to traditional language-based information flow properties [25]. Think of s_1 as defining the public, low-security data and s_2 as defining the private, high security data. Under this interpretation, observable behavior of the network remains unchanged regardless of whether the high-security data (s_2) is present or replaced by some alternate high security data (s_2').

Example, redux. Slices provide a solution to the scenario described in the example at the beginning of the section. We can assign each programmer a unique slice with boundaries that correspond to the locations of the end hosts under control of that slice. For instance, the first programmer's in and out predicates include the network access points for hosts 1 and 2, while the second programmer's in and out predicates include the network access points for hosts 3 and 4.

$$\begin{array}{lcl} in_1 & = & \mathsf{sw} = A; \mathsf{pt} = 1 + \mathsf{sw} = B; \mathsf{pt} = 2 \\ out_1 & = & \mathsf{sw} = A; \mathsf{pt} = 1 + \mathsf{sw} = B; \mathsf{pt} = 2 \\ s_1 & = & \{in_1\}\ w_1 : (p_{A1} + p_{B1})\ \{out_1\} \\ \\ in_2 & = & \mathsf{sw} = A; \mathsf{pt} = 3 + \mathsf{sw} = B; \mathsf{pt} = 3 \\ out_2 & = & \mathsf{sw} = A; \mathsf{pt} = 3 + \mathsf{sw} = B; \mathsf{pt} = 3 \\ s_2 & = & \{in_2\}\ w_2 : (p_{A2} + p_{B2})\ \{out_2\} \end{array}$$

The original difficulty with this example manifested as packet duplication when a packet was sent from host 1 to host 2. Corollary 1 shows that slices solve the problem: host 1 is connected to slice 1, and restricting the input to that of slice 1 implies that the behavior of the entire program is precisely that of slice 1 alone.

7. Compilation

A major challenge for the designers of network programming languages is compiling their high-level abstractions to the low-level operations that network hardware supports. For example, OpenFlow switches use a flow table of rules to process packets. Because NetKAT supports richer expressions than OpenFlow, we must design compilation algorithms to "flatten" NetKAT policies into OpenFlow rule tables.

Other network programming languages have compilation algorithms with proofs of correctness [5, 7, 22, 29] that painstakingly relate their high-level semantics with the low-level semantics of OpenFlow rules. We take a different approach that allows us to fully exploit our equational theory: we define a syntactic restriction of NetKAT based on OpenFlow tables, which we call *OpenFlow Normal Form* (ONF) and then prove by induction (using purely syntactic arguments) that NetKAT can be normalized to ONF.

The inductive proof is lengthy and relegated to the long version of this paper. A particularly challenging aspect of compilation is proving that sequential composition can be normalized: in an OpenFlow rule table (and thus in ONF) tests cannot be applied after actions. Therefore, we have to rewrite NetKAT terms by carefully applying the commutativity axioms. This section highlights some of the important steps that enable the inductive normalization proof. The inductive proof can also be interpreted as a recursive function that compiles NetKAT to OpenFlow Normal Form.

ONF Action Sequence	a	$::=$	$\mathsf{id} \mid f \leftarrow n; a$
ONF Action Sum	as	$::=$	$\mathsf{drop} \mid a + as$
ONF Predicate	b	$::=$	$\mathsf{id} \mid f = n; b$
ONF Local	ℓ	$::=$	$as \mid \mathsf{if}\ b\ \mathsf{then}\ as\ \mathsf{else}\ \ell$
ONF	p	$::=$	$\mathsf{drop} \mid (\mathsf{sw} = sw; \ell) + p$

Figure 7. OpenFlow Normal Form.

OpenFlow Normal Form (ONF). An OpenFlow switch processes a packet using a table of prioritized rules. In essence, each rule has a bit-pattern with wildcards to match packets and a list of actions to apply to packets that match the pattern. If a packet matches several rules, only the highest-priority rule is applied.

Instead of compiling NetKAT to rule tables directly, we define ONF, a syntactic subset of NetKAT that translates almost directly to rule tables. An ONF term is a sum of policies, where each sub-term is predicated on a switch number:

$$\mathsf{sw} = sw_1; \ell_1 + \cdots + \mathsf{sw} = sw_n; \ell_n$$

Above, $\ell_1 \cdots \ell_n$ are policies that do not test or modify the switch. Therefore, each of these policies can be run locally on a switch. These switch-local policies are syntactically restricted to a shape that resembles a rule table. Each is a cascade of if-then-else expressions, where the conditionals are a conjunction of positive literals. Note that if-then-else is simple syntactic sugar:

$$\mathsf{if}\ b\ \mathsf{then}\ as\ \mathsf{else}\ \ell \stackrel{\mathsf{def}}{=} (b; as) + (\neg b; \ell)$$

A policy p is in Openflow Normal Form ($p \in$ ONF) when it satisfies the grammar in Figure 7 and includes no occurrences of the field sw. Note that ONF excludes expressions that modify the switch or contain dup. These operations are meaningless on a switch: only the topology can update switch numbers, and dup only exists to enable equational reasoning about packet histories.

7.1 Compiling to ONF.

This section steps through a sequence of proofs that transform switch policies to ONF. Each proof eliminates or restricts an element of NetKAT syntax. In other words, each lemma translates from one intermediate representation to another, until we arrive at low-level ONF. As a naming convention, we write $Net\text{KAT}^{-(op)}$, where op to stand for NetKAT expressions that do not use the op operator. For example, if:

$$p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$$

then p does not contain dup and does not modify the switch field (*i.e.*, p is a switch policy).

Step 1: star elimination. The first step is to eliminate Kleene star from the input policy. This step is critical as switches do not support iterated processing of packets—indeed, many switches only support a single phase of processing by a table! Formally, we prove any program without dup, or, less importantly, assignment to sw, is equivalent to a Kleene star-free program (again without the dup primitive or assignments to sw).

Lemma 7 (Star Elimination)**.** *If $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$, then there exists $p' \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ and $p \equiv p'$.*

Proof. The proof begins by showing that p' can be obtained from the normal form used in the completeness theorem. More specifically, let p'' be the policy obtained from p by the normalization construction of lemma 4. By construction, dup can only appear in the normal form of an expression already containing dup, therefore p'' does not contain dup. Moreover, $R(p'') \subseteq I$ and p'' does not contain dup, therefore $R(p'') \subseteq At; P$. Consequently, p'' does not contain Kleene star.

Let us now prove that any assignment of the form $\mathsf{sw} \leftarrow sw_i$ in p'' is preceded in the same term by the corresponding test $\mathsf{sw} = sw_i$. Because p does not contain any assignment of the form $\mathsf{sw} \leftarrow sw_i$, it commutes

with any test of the form $\mathsf{sw} = sw_i$, and therefore p'' also commutes with any test of the form $\mathsf{sw} = sw_i$. p'' can be written as a sum of $\alpha;p$ for some atoms α and complete assignments p. Suppose for a contradiction that term, α contains a test $\mathsf{sw} = sw_i$, and p contains an assignment $\mathsf{sw} \leftarrow sw_j$, with $sw_i \neq sw_j$. Then

$$\alpha; (\mathsf{sw} = sw_i); p''; (\mathsf{sw} = sw_j) \geq \alpha; p \neq 0$$
$$\alpha; (\mathsf{sw} = sw_j); p''; (\mathsf{sw} = sw_i) = 0$$

but those two terms are also equal, which is a contradiction.

Therefore any assignment of the form $\mathsf{sw} \leftarrow sw_i$ in p'' is preceded, in the same term, by the corresponding test $\mathsf{sw} = sw_i$, and can be removed using axiom PA-Filter-Mod to produce the desired p'. Tests and assignments to other fields than sw could appear in between, but we can use the commutativity axioms PA-Mod-Mod-Comm and PA-Mod-Filter-Comm to move the assignment $\mathsf{sw} \leftarrow sw_i$ to just after the test $\mathsf{sw} = sw_i$. □

Step 2: switch specialization. Next, we show that it is possible to specialize star-free policies to each switch and remove nested tests of the switch field sw. This puts the policy into a form where it can readily compiled to local configurations for each switch.

Lemma 8 (Switch Specialization)**.** *If $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, then for all switches sw_i, there exists $p' \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ such that $\mathsf{sw} = sw_i; p \equiv \mathsf{sw} = sw_i; p'$.*

Proof. Let g be the unique homomorphism of NetKAT defined on primitive programs by:

$$g(\mathsf{sw} = sw) \triangleq \begin{cases} \mathsf{id} & \text{if } sw = sw_i \\ \mathsf{drop} & \text{otherwise} \end{cases}$$
$$g(f \leftarrow v) \triangleq f \leftarrow v$$
$$g(\mathsf{dup}) \triangleq \mathsf{dup}$$

For every primitive program element x of NetKAT$^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, we have both:

$$\mathsf{sw} = sw_i; x \equiv g(x); \mathsf{sw} = sw_i$$
$$g(x); \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i; g(x)$$

Hence, applying Lemma 4.4 in [2] twice shows:

$$\mathsf{sw} = sw_i; p \equiv g(p); \mathsf{sw} = sw_i$$
$$g(p); \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i; g(p)$$

By the definition of g, any occurrence of $\mathsf{sw} = v$ in p is replaced by either id or drop in $g(p)$. Moreover, since $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, it follows that $g(p)$ does not contain any occurrence of $\mathsf{sw} = v$ and since $p' = g(p) \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ we also have

$$\mathsf{sw} = sw_i; p \equiv \mathsf{sw} = sw_i; p'$$ □

Step 3: Converting to ONF. Third, we show any policy in NetKAT$^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ can be transformed in to ONF.

Lemma 9 (Switch-local Compilation)**.**
If $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,^*,\mathsf{sw})}$ then there exists a policy p' such that $p \equiv p'$ and $p' \in$ ONF.

The proof goes by induction on the structure of p.

Step 4: combining results. Lemmas 7, 8 and 9 suffice to prove any policy p in $Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$ may be converted to OpenFlow normal form.

Theorem 7 (ONF). *If $p_{in} \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$, then there exists $p_{out} \equiv p_{in}$ such that $p_{out} \in$ ONF.*

Optimizations. Naive compilation of network programs can produce switch rule tables that are unmanageably large [22]. Hence, existing systems implement optimizations to generate smaller tables. For example, the following lemma describes a common optimization called fall-through elimination.

Lemma 10 (Fall-through Elimination). *If $b_1 \leq b_2$ then* $\mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else}\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell\ \equiv\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell$.

This transformation removes an unnecessary rule from a table.

8. Related Work

Kleene algebra is named for its inventor, Stephen Cole Kleene. Much of the basic algebraic theory of KA was developed by John Horton Conway [4]. Kleene algebra with tests was introduced by Kozen [13, 14]. KA and KAT have been successfully applied in many practical verification tasks, including verification of compiler optimizations [16], pointer analysis [21], concurrency control [3], and device drivers [15]. This is the first time KA has been used as a network programming language or applied to verification of networks. The proof of the main result in this paper—completeness of the equational axioms—is based on a novel model of KAT.

While many other systems have been proposed for analyzing networks, we believe ours is the first to provide a complete, high-level algebra for reasoning about network programs *as they are written*. Systems such as Anteater [19], FlowChecker [1], Header Space Analysis [10], Veriflow [11], and Formally Verifiable Networking [30], encode information about network topology and forwarding policies into SAT formulae (Anteater), graph-based representations (Veriflow, Header Space Analysis), or higher-order logic (Formally Verifiable Networking). These systems then define custom algorithms over these models to check specific properties such as reachability or packet loss. Such systems can check for violations of important network invariants, but do not provide sound and complete systems for reasoning *directly* about programs. Moreover, although these systems have expressive languages for encoding properties, they do not connect these encodings back to denotational or operational models of the network. In contrast, in section 5, we show how to encode a reachability property as a NetKAT equation and then prove that the reachability equation is equivalent to a semantic definition of reachability.

As a programming language, NetKAT is most similar to NetCore [7, 22] and Pyretic [23], which both stemmed from earlier work on Frenetic [6]. NetCore defined the fragment of NetKAT that included parallel composition and Pyretic extended NetCore with sequential composition, though Pyretic gave neither a formal semantics nor a compiler. Neither system defined an equational theory for reasoning about programs, nor did it include Kleene star—unlike these previous languages, NetKAT programs can describe potentially infinite behaviors.

NDLog [18] is a logic programming language with an explicit notion of location and a distributed execution model. In contrast to NDLog, NetKAT and NetCore are designed for programming centralized (not distributed) SDN controllers. Because NDLog is based around Datalog (with general recursion and pragmatic extensions that complicate its semantics), equivalence of NDLog programs is undecidable [27]. NetKAT's Kleene star is able to model network behavior, but has decidable (PSPACE-complete) equivalence.

9. Conclusion

This paper defines NetKAT, a language for programming and reasoning about networks that is based on a solid semantic foundation—Kleene algebra with tests. NetKAT's denotational semantics describes network programs as functions from packet histories to sets of packets histories and its equational theory is sound and complete with respect to this model. The language enables programmers to create expressive, compositional network programs and reason effectively about their semantics. We demonstrate the power of our framework on a range of practical applications including reachability, traffic isolation, access control, and compiler correctness.

Acknowledgments The authors wish to thank Timothy Griffin, Shriram Krishnamurthi, Nick McKeown, Jennifer Rexford and the members of the Cornell PLDG for helpful comments, as well as Alec Story and Stephen Gutz for work on a preliminary version of slices, and Rebecca Coombes and Matthew Milano for developing an early NetCore verification tool. This work is supported in part by the NSF under grant CNS-1111698, the ONR under award N00014-12-1-0757, a Sloan Research Fellowship, and a Google Research Award.

References

[1] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[2] Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report TR2001-1844, Computer Science Department, Cornell University, July 2001.

[3] Ernie Cohen. Using Kleene algebra to reason about concurrency control. Technical report, Telcordia, Morristown, N.J., 1994.

[4] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[5] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory networking: An API for application control of SDNs. In *SIGCOMM*, 2013.

[6] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, Sep 2011.

[7] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, June 2013.

[8] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[9] James Hamilton. Networking: The last bastion of mainframe computing, Dec 2009. Available at `http://tinyurl.com/y9uz64e`.

[10] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[11] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[12] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *I&C*, 110(2):366–390, May 1994.

[13] Dexter Kozen. Kleene algebra with tests and commutativity conditions. In *TACAS*, pages 14–33, Passau, Germany, March 1996.

[14] Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[15] Dexter Kozen. Kleene algebras with tests and the static analysis of programs. Technical Report TR2003-1915, Computer Science Department, Cornell University, November 2003.

[16] Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using Kleene algebra with tests. In *CL*, July 2000.

[17] Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, September 1996.

[18] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, 2005.

[19] Haohui Mai, Ahmed Khurshid, Raghit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[20] James McCauley, Aurojit Panda, Martin Casado, Teemu Koponen, and Scott Shenker. Extending SDN to large-scale networks. In *ONS*, 2013.

[21] B. Möller. Calculating with pointer structures. In *Algorithmic Languages and Calculi. Proc. IFIP TC2/WG2.1 Working Conference*, February 1997.

[22] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, Jan 2012.

[23] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *NSDI*, Apr 2013.

[24] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

[25] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[26] Gunther Schmidt. *Relational Mathematics*. Cambridge University Press, 2010.

[27] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *PODS*, pages 237–249, 1987.

[28] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

[29] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, 2013.

[30] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally verifiable networking. In *HotNets*, 2009.

[31] Minlan Yu, Jennifer Rexford, Xin Sun, Sanjay G. Rao, and Nick Feamster. A survey of virtual LAN usage in campus networks. *IEEE Communications Magazine*, 49(7):98–103, 2011.

This appendix presents additional details of the lemmas and theorems in the main body of the paper.

A. Soundness and Completeness

This section presents details of the proofs and supporting lemmas of the theorems in Section 4.

A.1 Relational Semantics

For soundness, it is helpful to reformulate the standard packet-history semantics introduced in §3 in terms of binary relations. In the standard semantics, policies and predicates are modeled as functions $[\![p]\!] : \mathsf{H} \to \mathcal{P}(\mathsf{H})$, where H is the set of packet histories and $\mathcal{P}(\mathsf{H})$ denotes the powerset of H. Intuitively, $[\![p]\!]$ takes an input packet history h and produces a set of output packet histories $[\![p]\!]\,(h)$ in response.

Formally, the maps $[\![p]\!] : \mathsf{H} \to \mathcal{P}(\mathsf{H})$ are morphisms of type $\mathsf{H} \to \mathsf{H}$ in $\mathsf{Kl}\,\mathcal{P}$, the Kleisli category of the powerset monad. Composition in a Kleisli category of a monad is called *Kleisli composition*. For the powerset monad, it is defined as:

$$[\![p; q]\!] = [\![p]\!] \bullet [\![q]\!] = \lambda h\,.\bigcup \{\, [\![q]\!]\,(h') \mid h' \in [\![p]\!]\,(h) \,\}.$$

The identity morphisms are $[\![\mathsf{id}]\!] = \lambda h\,.\,\{h\}$.

We can lift $[\![p]\!]$ to $L\,[\![p]\!] : \mathcal{P}(\mathsf{H}) \to \mathcal{P}(\mathsf{H})$ by:

$$L\,[\![p]\!]\,(A) = \bigcup \{\, [\![p]\!]\,(h) \mid h \in A \,\}\,.$$

Intuitively, given a set of input packet histories $A \subseteq \mathsf{H}$, $L\,[\![p]\!]$ produces a set of output packet histories by applying $[\![p]\!]$ to each element of A individually and accumulating the results. This is just the familiar subset construction of automata theory.

The lifting functor L embeds the Kleisli category $\mathsf{Kl}\,\mathcal{P}$ faithfully, but not fully, in the category Set of sets and set functions. Thus

$$L\,[\![p; q]\!]\,(A) = L\,[\![q]\!]\,(L\,[\![p]\!]\,(A)) \qquad\qquad L\,[\![\mathsf{id}]\!]\,(A) = A.$$

It is well known that the Kleisli category $\mathsf{Kl}\,\mathcal{P}$ is isomorphic to the category Rel of sets and binary relations. This is simply a matter of currying:

$$X \to \mathcal{P}(Y) \cong X \to Y \to \mathbf{2} \cong X \times Y \to \mathbf{2} \cong \mathcal{P}(X \times Y).$$

The isomorphism gives an equivalent view of the semantics in which each policy and predicate is interpreted as a binary relation $[p] \subseteq \mathsf{H} \times \mathsf{H}$:

$$(h_1, h_2) \in [p] \;\Leftrightarrow\; h_2 \in [\![p]\!]\,(h_1).$$

In relational models, product is ordinary relational composition

$$[p; q] = [p] \circ [q] = \{(h_1, h_2) \mid \exists h_3\ (h_1, h_3) \in [p] \wedge (h_3, h_2) \in [q]\}$$

with identity $[\mathsf{id}] = \{(h, h) \mid h \in \mathsf{H}\}$. The remaining KAT operations $+$, *, drop, and $\neg$ translate under the isomorphism to the usual KAT operations on binary relations

$$[p + q] = [p] \cup [q] \qquad [p^*] = \bigcup_{n \geq 0} [p^n] \qquad [\mathsf{drop}] = \emptyset \qquad [\neg b] = [\mathsf{id}] \setminus [b].$$

The relation $[p^*]$ is the reflexive transitive closure of $[p]$. For every predicate b, by definition there exists $B \subseteq \mathsf{H}$ such that

$$[\![b]\!] = \lambda h\,.\begin{cases} \{h\} & h \in B \\ \emptyset & h \notin B \end{cases} \qquad\qquad [b] = \{(h, h) \mid h \in B\}$$

in the Kleisli and relational view, respectively. Intuitively, the packet is passed through unaltered if it passes the test b and dropped if not. The Boolean operations on tests are the usual set-theoretic ones.

Since relational models with these distinguished operations satisfy the axioms of KAT (see e.g. [14, 17]), so do NetKAT models with the packet-history semantics of §3.

A.2 Soundness

Theorem 1 (Soundness)**.** *The* KAT *axioms with the equational premises of Figure 2 are sound with respect to the semantics of Section 3. That is, if* $\vdash p \equiv q$*, then* $[\![p]\!] = [\![q]\!]$*.*

Proof. We have already argued that any packet history model is isomorphic to a relational KAT, therefore satisfies all the axioms of KAT in fig. 2.

It remains to show that the specialized NetKAT axioms on are satisfied. These can all be verified by elementary arguments in relation algebra (see e.g. [26]). Some are special cases of [2, Equations (6)–(11)], whose soundness is proved in [2, Theorem 4.3].

- (PA-OBS-FILTER-COMM) $[\![\mathsf{dup}; f = n]\!] = [\![f = n; \mathsf{dup}]\!]$.
  In terms of relational algebra, we must argue that

  $$[\mathsf{dup}] \circ [f = n] = [f = n] \circ [\mathsf{dup}].$$

  The dup operation duplicates the head packet in the packet history. Since the new head packet is identical to the old, any test on the head packet will yield the same truth value before and after the dup; thus $[\mathsf{dup}]$ respects the partition $B, \bar{B}$, that is, $[\mathsf{dup}] \subseteq (B \times B) \cup (\bar{B} \times \bar{B})$, where B is the domain of $[f = n]$. It follows from relation algebra that

  $$[f = n] \circ [\mathsf{dup}] = [\mathsf{dup}] \circ [f = n] = [\mathsf{dup}] \cap (B \times B).$$

- (PA-MOD-MOD) $[\![f \leftarrow n_1; f \leftarrow n_2]\!] = [\![f \leftarrow n_2]\!]$.
  This is a special case of [2, Equation (8)]. Intuitively, an assignment to f immediately followed by another assignment to f has the same effect as if the first assignment was never done.
- (PA-MOD-MOD-COMM)
  $[\![f_1 \leftarrow n_1; f_2 \leftarrow n_2]\!] = [\![f_2 \leftarrow n_2; f_1 \leftarrow n_1]\!]$, where $f_1 \neq f_2$.
  This is a special case of [2, Equation (10)]. Intuitively, assignments to different fields can be done in either order.
- (PA-MOD-FILTER) $[\![f \leftarrow n; f = n]\!] = [\![f \leftarrow n]\!]$.
  This is a special case of [2, Equation (9)] (allowing the universally true test $n = n$). Intuitively, assigning n to f makes the test $f = n$ true.
- (PA-MOD-FILTER-COMM)
  $[\![f_1 \leftarrow n_1; f_2 = n_2]\!] = [\![f_2 = n_2; f_1 \leftarrow n_1]\!]$, where $f_1 \neq f_2$.
  This is a special case of [2, Equation (11)]. Intuitively, an assignment to a field does not affect the value of another field.
- (PA-FILTER-MOD) $[\![f = n; f \leftarrow n]\!] = [\![f = n]\!]$.
  Relationally, $[f = n] \circ [f \leftarrow n] = [f = n]$. This is true because the assignment $[f \leftarrow n]$ acts as the identity on the codomain of $[f = n]$. Intuitively, if the value of f is already n, there is no need to assign it again.
- (PA-CONTRA)
  $[\![f = n_1; f = n_2]\!] = [\![\mathsf{drop}]\!]$, where $n_1 \neq n_2$.
  Equivalently, $[f = n_1] \circ [f = n_2] = \emptyset$. This is true because the codomain of $[f = n_1]$ and the domain of $[f = n_2]$ are disjoint. Intuitively, no packet can have two distinct values for f.
- (PA-MATCH-ALL) $[\![\sum_n f = n]\!] = [\![\mathsf{id}]\!]$.
  Equivalently, $\bigcup_n [f = n] = \mathsf{H}$. This is true because the domains of $[f = n]$, accumulated over all possible values n for the field f, exhaust H. Intuitively, every packet has a value for f.

□

A.3 Completeness

Lemma 1. *For all reduced policies p, we have $[\![p]\!] = \bigcup_{x \in G(p)} [\![x]\!]$.*

Proof. By structural induction on p. For the basis,

$$\begin{aligned}
[\![\pi]\!] &= \left[\!\!\left[\sum_{\alpha \in At} \alpha ; \pi \right]\!\!\right] = \bigcup_{\alpha \in At} [\![\alpha ; \pi]\!] = \bigcup_{x \in G(\pi)} [\![x]\!] , \\
[\![\alpha]\!] &= [\![\alpha ; \pi_\alpha]\!] = \bigcup_{x \in G(\alpha)} [\![x]\!] , \\
[\![\mathsf{dup}]\!] &= \left[\!\!\left[\sum_{\alpha \in At} \alpha ; \pi_\alpha ; \mathsf{dup} ; \pi_\alpha \right]\!\!\right] \\
&= \bigcup_{\alpha \in At} [\![\alpha ; \pi_\alpha ; \mathsf{dup} ; \pi_\alpha]\!] = \bigcup_{x \in G(\mathsf{dup})} [\![x]\!] .
\end{aligned}$$

For the induction step,

$$\begin{aligned}
[\![p + q]\!] = [\![p]\!] \cup [\![q]\!] &= \left(\bigcup_{x \in G(p)} [\![x]\!] \right) \cup \left(\bigcup_{x \in G(q)} [\![x]\!] \right) \\
&= \bigcup_{x \in G(p) \cup G(q)} [\![x]\!] = \bigcup_{x \in G(p+q)} [\![x]\!] ,
\end{aligned}$$

$$\begin{aligned}
[\![p ; q]\!] = [\![p]\!] \bullet [\![q]\!] &= \left(\bigcup_{x \in G(p)} [\![x]\!] \right) \bullet \left(\bigcup_{y \in G(q)} [\![y]\!] \right) \\
&= \bigcup_{x \in G(p)} \bigcup_{y \in G(q)} [\![x]\!] \bullet [\![y]\!] \\
&= \bigcup_{x \in G(p)} \bigcup_{y \in G(q)} [\![x ; y]\!] \\
&= \bigcup_{x \in G(p)} \bigcup_{y \in G(q)} [\![x \diamond y]\!] = \bigcup_{z \in G(pq)} [\![z]\!] ,
\end{aligned}$$

$$[\![p^*]\!] = \bigcup_{n \geq 0} [\![p^n]\!] = \bigcup_{n \geq 0} \bigcup_{x \in G(p^n)} [\![x]\!] = \bigcup_{x \, \in \bigcup_n G(p^n)} [\![x]\!] = \bigcup_{x \in G(p^*)} [\![x]\!] .$$

□

Lemma 2. *If $x, y \in I$, then $[\![x]\!] = [\![y]\!]$ if and only if $x = y$.*

Proof. If $x = \alpha ; \pi_0 ; \mathsf{dup} ; \cdots ; \mathsf{dup} ; \pi_n$ and h is any single packet satisfying α, then $[\![x]\!](h) = \{h'\}$, where h' consists of $n+1$ packets satisfying $\alpha_{\pi_0}, \ldots, \alpha_{\pi_n}$ in chronological order. The only string in I that could produce this h' from h is x. Thus x is uniquely determined by $[\![x]\!]$. □

Lemma 3. *For all reduced NetKAT policies p and q, we have $[\![p]\!] = [\![q]\!]$ if and only if $G(p) = G(q)$.*

Proof. Using Lemma 2,

$$\begin{aligned}
[\![p]\!] = [\![q]\!] &\Rightarrow \bigcup_{x \in G(p)} [\![x]\!] = \bigcup_{y \in G(q)} [\![y]\!] \\
&\Rightarrow \forall h \bigcup_{x \in G(p)} [\![x]\!](h) \subseteq \bigcup_{y \in G(q)} [\![y]\!](h) \\
&\Rightarrow \forall h\, \forall x \in G(p)\ [\![x]\!](h) \subseteq \bigcup_{y \in G(q)} [\![y]\!](h) \\
&\Rightarrow \forall h\, \forall x \in G(p)\, \exists y \in G(q)\ [\![x]\!](h) \subseteq [\![y]\!](h) \\
&\Rightarrow \forall x \in G(p)\, \exists y \in G(q)\ x = y \\
&\Rightarrow G(p) \subseteq G(q),
\end{aligned}$$

and similarly for the reverse inclusion. The converse follows from Lemma 1. □

Lemma 4. *Every policy is normalizable.*

Proof. The proof proceeds by induction on the structure of policies. To begin, the primitive symbols are normalizable:

$$\begin{array}{lcl}
h \leftarrow n & \equiv & \sum_{\alpha \in At} \alpha; \pi'_\alpha \\
\mathsf{dup} & \equiv & \sum_{\alpha \in At} \alpha; \pi_\alpha; \mathsf{dup}; \pi_\alpha \\
b & \equiv & \sum_{\alpha \leq b} \alpha; \pi_\alpha
\end{array}$$

Next, we show that sums and products of normalizable expressions are normalizable. The case for sums is trivial, and the case for products follows by a simple argument:

$$\left(\sum_i s_i\right); \left(\sum_j t_j\right) \equiv \sum_i \sum_j s_i; t_j \equiv \sum_i \sum_j s_i \diamond t_j.$$

The most interesting case is for Kleene star. Consider an expression p^*, where p is in normal form. We first prove the uniform case: when all guarded terms in the sum p have the same initial atom α, that is, $p = \alpha; t$ where t is a sum of terms each with a leading and trailing π, and $R(t) \subseteq P; (\mathsf{dup}; P)^*$. Let u be t with all terms whose trailing π is not π_α deleted and with the trailing π_α deleted from all remaining terms. By the simplified axioms of Figure 5, we have $t; \alpha; t \equiv u; t$, therefore $t; \alpha; t; \alpha \equiv u; t; \alpha$. Using [2, Lemma 4.4],

$$\begin{array}{lcl}
(t; \alpha)^*; t & \equiv & t + t; \alpha; (t; \alpha)^*; t \\
& \equiv & t + u^*; t; \alpha; t \\
& \equiv & t + u^*; u; t \\
& \equiv & u^*; t,
\end{array}$$

and hence

$$\begin{array}{lcl}
p^* & \equiv & \mathsf{id} + p^*; p \\
& \equiv & \mathsf{id} + (\alpha; t)^*; \alpha; t \\
& \equiv & \mathsf{id} + \alpha; (t; \alpha)^*; t \\
& \equiv & \mathsf{id} + \alpha; u^*; t \\
& \equiv & \mathsf{id} + \alpha; t + \alpha; u; u^*; t,
\end{array}$$

which after normalizing the id is in normal form.

Finally, for the case p^* where the initial tests in the sum p are not uniform, the argument is by induction on the number of terms in the sum. If $p = \alpha; x + q$, then by induction hypothesis, q^* has an equivalent normal form $\hat{q}^*$. Using KAT-Denesting (Figure 3), we obtain

$$p^* \equiv (\alpha; x + q)^* \equiv q^*; (\alpha; x; q^*)^* \equiv \hat{q}^*; (\alpha; x; \hat{q}^*)^*,$$

and then proceed as in the previous case. □

Lemma 5.

$$G(p) = \bigcup_{x \in R(p)} G(x).$$

Proof. By structural induction on p. For the basis, since $R(\pi) = \{\pi\}$,

$$G(\pi) = \bigcup_{x \in \{\pi\}} G(x) = \bigcup_{x \in R(\pi)} G(x),$$

and similarly for α and dup. For the induction step,

$$\begin{aligned} G(p+q) &= G(p) \cup G(q) \\ &= \left(\bigcup_{x \in R(p)} G(x) \right) \cup \left(\bigcup_{x \in R(q)} G(x) \right) \\ &= \bigcup_{x \in R(p) \cup R(q)} G(x) = \bigcup_{x \in R(p+q)} G(x), \end{aligned}$$

$$\begin{aligned} G(p;q) &= G(p) \diamond G(q) \\ &= \left(\bigcup_{x \in R(p)} G(x) \right) \diamond \left(\bigcup_{y \in R(q)} G(y) \right) \\ &= \bigcup_{x \in R(p)} \bigcup_{y \in R(q)} G(x) \diamond G(y) \\ &= \bigcup_{x \in R(p)} \bigcup_{y \in R(q)} G(x;y) = \bigcup_{z \in R(p;q)} G(z), \end{aligned}$$

$$\begin{aligned} G(p^*) &= \bigcup_{n \geq 0} G(p^n) = \bigcup_{n \geq 0} \bigcup_{x \in R(p^n)} G(x) \\ &= \bigcup_{x \,\in \bigcup_n R(p^n)} G(x) = \bigcup_{x \in R(p^*)} G(x). \end{aligned}$$

□

Lemma 6. *If $R(p) \subseteq I$, then $R(p) = G(p)$.*

Proof. Suppose $R(p) \subseteq I$. Since $G(x) = \{x\}$ for $x \in I$, by Lemma 5 we have

$$G(p) = \bigcup_{x \in R(p)} G(x) = \bigcup_{x \in R(p)} \{x\} = R(p).$$ □

Theorem 2 (Completeness)**.** *Every semantically equivalent pair of expressions is provably equivalent in Net*KAT*. That is, if $[\![p]\!] = [\![q]\!]$, then $\vdash p \equiv q$.*

Proof. Let $\hat{p}$ and $\hat{q}$ be the normal forms of p and q. By Lemma 4, we can prove that each is equivalent to its normal form: $\vdash p \equiv \hat{p}$ and $\vdash q \equiv \hat{q}$. By soundness we have $[\![p]\!] = [\![\hat{p}]\!]$ and $[\![q]\!] = [\![\hat{q}]\!]$. Hence $[\![\hat{p}]\!] = [\![\hat{q}]\!]$ by transitivity. By Lemma 3, we have $G(\hat{p}) = G(\hat{q})$. Moreover, by Lemma 6, we have $G(\hat{p}) = R(\hat{p})$ and $G(\hat{q}) = R(\hat{q})$, thus $R(\hat{p}) = R(\hat{q})$ by transitivity. Since $R(\hat{p})$ and $R(\hat{q})$ are regular sets, by the completeness of KA, we also have $\vdash \hat{p} \equiv \hat{q}$. Finally, as $\vdash p \equiv \hat{p}$ and $\vdash q \equiv \hat{q}$ and $\vdash \hat{p} \equiv \hat{q}$, by transitivity we conclude that $\vdash p \equiv q$, as required. □

Syntax

$$\text{Atoms} \quad \alpha, \beta \triangleq f_1 = n_1; \cdots; f_k = n_k$$

$$\text{Assignments} \quad \pi \triangleq f_1 \leftarrow n_1; \cdots; f_k \leftarrow n_k$$

$$\text{Join-irreducibles} \quad x, y \in At; (P; \mathsf{dup})^*; P$$

Simplified axioms for At and P

$$\pi \equiv \pi; \alpha_\pi \quad \alpha; \mathsf{dup} \equiv \mathsf{dup}; \alpha \quad \sum_\alpha \alpha \equiv \mathsf{id},$$

$$\alpha \equiv \alpha; \pi_\alpha \qquad \pi; \pi' \equiv \pi' \qquad \alpha; \beta \equiv \mathsf{drop},\ \alpha \neq \beta$$

Join-irreducible concatenation

$$\alpha; p; \pi \diamond \beta; q; \pi' = \begin{cases} \alpha; p; q; \pi' & \text{if } \beta = \alpha_\pi \\ \text{undefined} & \text{if } \beta \neq \alpha_\pi \end{cases}$$

$$A \diamond B = \{p \diamond q \mid p \in A,\ q \in B\} \subseteq I$$

Regular Interpretation: $R(p) \subseteq (P + At + \mathsf{dup})^*$

$$\begin{aligned} R(\pi) &= \{\pi\} \\ R(p+q) &= R(p) \cup R(q) \\ R(\alpha) &= \{\alpha\} \\ R(p;q) &= \{xy \mid x \in R(p), y \in R(q)\} \\ R(\mathsf{dup}) &= \{\mathsf{dup}\} \\ R(p^*) &= \bigcup_{n \geq 0} R(p^n) \end{aligned}$$

with $p^0 = 1$ and $p^{n+1} = p; p^n$

Language Model: $G(p) \subseteq I = At; (P; \mathsf{dup})^*; P$

$$\begin{aligned} G(\pi) &= \{\alpha; \pi \mid \alpha \in At\} \\ G(p+q) &= G(p) \cup G(q) \\ G(\alpha) &= \{\alpha; \pi_\alpha\} \\ G(p;q) &= G(p) \diamond G(q) \\ G(\mathsf{dup}) &= \{\alpha; \pi_\alpha; \mathsf{dup}; \pi_\alpha \mid \alpha \in At\} \\ G(p^*) &= \bigcup_{n \geq 0} G(p^n) \end{aligned}$$

Figure 5. Completeness definitions.

A.4 Decidability

Theorem 3. *The equational theory of Net*KAT *is* PSPACE*-complete.*

Proof sketch. To show PSPACE-hardness, reduce KA to NetKAT as follows. Let Σ be a finite alphabet. For a regular expression e over Σ, let $R(e)$ be the regular set of strings over Σ as defined in §4. Transform e to a NetKAT expression e' by replacing each occurrence in e of a symbol $p \in \Sigma$ with $(p; \mathsf{dup})$ and prepending with an arbitrary but fixed atom α. It follows from Lemmas 3 and 6 that $R(e_1) = R(e_2)$ if and only if $R(e'_1) = R(e'_2)$ if and only if $G(e'_1) = G(e'_2)$ if and only if $[\![e'_1]\!] = [\![e'_2]\!]$.

To show that the problem is in PSPACE, given two NetKAT expressions e_1 and e_2, we know that $[\![e_1]\!] \neq [\![e_2]\!]$ if and only if there is a packet pk and packet history h such that $h \in [\![e_1]\!](pk) \setminus [\![e_2]\!](pk)$ or $h \in [\![e_2]\!](pk) \setminus [\![e_1]\!](pk)$; let us say the former without loss of generality. We guess pk nondeterministically and follow a nondeterministically-guessed trajectory through e_1 that produces some $h \in [\![e_1]\!](pk)$. At the same time, we trace all possible trajectories through e_2 that could generate a prefix of h, ensuring that none of these produce $h \in [\![e_2]\!]$. It takes only polynomial space to represent the current values of the fields of the head packet and the possible positions in e_2 for the current prefix of h. The algorithm is nondeterministic, but can be made deterministic using Savitch's theorem. □

B. Application: Reachability

This section contains the full proofs and supporting lemmas for the theorems in Section 5. Figure 5 summarizes many of the definitions used here. Recall that atoms and complete assignments are in a one-to-one correspondence according to the values $\bar{n}$. Hence, if π is a complete assignment, we write α_π to denote the corresponding atom, and if α is an atom, then π_α denotes the corresponding complete assignment.

We can also see a close correspondence between join-irreducible terms in the language model and traces in the denotational semantics. For example, consider a NetKAT program p and a trace $[pk_n \cdots pk_1] \in \mathsf{rng}(\llbracket p \rrbracket)$. The trace corresponds to a join-irreducible term $\alpha; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(p)$, where each complete assignment π_{pk_k} is responsible for assigning the values that lead to the observation of pk_k in the trace.

Lemma 11 (Remember First Hop)**.** *If* $\alpha; \pi_{pk_n}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$ *then* $\alpha = \alpha_{\pi_{pk_n}}$.

Proof.

	Assertion	**Reasoning**
	$\alpha; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$	Hypothesis
	$\alpha; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\mathsf{dup}) \diamond G((p; t; \mathsf{dup})^*)$	By definition of $\diamond$
	$\alpha; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} = \alpha'; \pi'_{pk_1}; \mathsf{dup} \diamond \alpha'; I$	By definition of $G(\mathsf{dup})$
	$\pi'_{pk_1} = \pi_{pk_1}$	By definition of $\diamond$
	$\alpha; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} = \alpha_{\pi_{pk_1}}; \pi_{pk_1}; \mathsf{dup}; \pi_{pk_1} \diamond \alpha_{\pi_{pk_1}}; I$	by definition of $G(\mathsf{dup})$
Goal	$\alpha = \alpha_{\pi_{pk_1}}$	Immediate $\square$

Lemma 12 (Range to Language)**.** *If* $[pk_n \cdots pk_1] \in \mathsf{rng}(\llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket)$ *then* $\exists \alpha . \alpha; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$

Proof.

Assertion	**Reasoning**
$[pk_n \cdots pk_1] \in \mathsf{rng}(\llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket)$	Hypothesis
$\exists h [pk_n \cdots pk_1] \in \llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket\ h$	Definition of range
$[pk_n \cdots pk_1] \in (\bigcup_{x \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)} \llbracket x \rrbracket)\ h$	Lemma 1

Therefore there exists an α', such that $\alpha'; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$ and $[pk_n \cdots pk_1] \in \llbracket \alpha'; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; pk_{mn} \rrbracket\ h$.

We show that $\alpha = \alpha'$ and $[pk_{mn} \cdots pkm1] = [pk_n \cdots pk_1]$. Since we have $\alpha'; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$, this will be enough to establish the goal.

Case 1: $h = [pk_x]$.

	Assertion	**Reasoning**
	$[pk_n \cdots pk_1]$	
=	$\llbracket \alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \rrbracket\ [pk_x]$	
=	$\llbracket \alpha \rrbracket \bullet \llbracket \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \rrbracket\ [pk_x]$	by function composition
=	$\llbracket \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \rrbracket\ [pk_x]$	α must be a complete test for pk_x
=	$\llbracket \pi_{pk_{m1}}; \mathsf{dup} \rrbracket \bullet \llbracket \cdots \mathsf{dup}; \pi_{pk_{mn}} \rrbracket\ [pk_x]$	by function composition
=	$\llbracket \cdots \mathsf{dup}; \pi_{pk_{mn}} \rrbracket\ [pk_{m1}\ pk_{m1}]$	by definition of dup and assignment
=	$\llbracket \cdots \mathsf{dup} \rrbracket \bullet \llbracket \pi_{pk_{mn}} \rrbracket\ [pk_{m1}\ pk_{m1}]$	by function composition
=	$\llbracket \pi_{pk_{mn}} \rrbracket\ [pk_{mn-1}\ pk_{mn-1} \cdots pk_{m1}]$	by definition of dup and assignment
=	$[pk_{mn}\ pk_{mn-1}\ pk_{mn-1} \cdots pk_{m1}]$	by definition of assignment

Case 2: h has more than 1 element. Because join-irreducible elements alternate between dup and modifications, the lengths of the lists are different, hence we have a contradiction. $\square$

Lemma 13 (Linguistic Test). $[\![p]\!]\ [pk] = \{[pk]\}$, *if and only if* $\alpha_{\pi_{pk}}; \pi_{pk} \in G(p)$

Proof.

	Assertion	Reasoning
	$[\![p]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	
$\iff$	$\bigcup_{x \in G(p)} [\![x]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	by lemma 1
$\iff$	$[\![\alpha; \pi_{pk_0}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n}]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	
	where $\alpha; \pi_{pk_0}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(p)$	by definition of $\bigcup$.
$=$	$[\![\alpha; \pi_{pk_0}]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	$n = 0$, since output has length 1
$=$	$[\![\alpha_{\pi_{pk}}; \pi_{pk_0}]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	since the result in non-empty, $\alpha = \alpha_{\pi_{pk}}$
$=$	$[\![\alpha_{\pi_{pk}}]\!] \bullet [\![\pi_{pk_0}]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	by $\bullet$
$=$	$[\![\pi_{pk_0}]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	matching test
$=$	$[\![\pi_{pk}]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	$pk = pk_0$, since complete assignment
$=$	$[\![\alpha_{\pi_{pk}}; \pi_{pk}]\!]\ \langle pk\rangle = \{\langle pk\rangle\}$	by $\bullet$
	$\alpha_{\pi_{pk}}; \pi_{pk} \in G(p)$ $\square$	

Theorem 7 (Reachability). *For predicates a and b, policy p, and topology t,*

$$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \neq 0$$

if, and only if, there exists $pk_1 \cdots pk_n$ such that

- $[pk_1 \cdots pk_n] \in \mathsf{rng}(\ [\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$, *and*
- $[\![a]\!]\ [pk_n] = \{[pk_n]\}$, *and*
- $[\![b]\!]\ [pk_1] = \{[pk_1]\}$.

Proof. Without loss of generality, assume that all tests and assignments are complete.

Direction $\Rightarrow$.

	Assertion	Reasoning
	$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \neq 0$	Hypothesis
$=$	$[\![a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b]\!] \neq [\![0]\!]$	Theorem 1
$=$	$G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b) \neq G(0)$	Lemma 3
	$\exists \alpha, \pi_{pk_0}, \cdots, \pi_{pk_m}$ where	
	$\alpha; \pi_{pk_0} \cdots \pi_{pk_m} \in G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b)$	

Let $\pi_{pk_1} = \pi_{pk_m}, \cdots$, and $\pi_{pk_n} = \pi_{pk_0}$, and let $\alpha = \alpha_{\pi_{pk_n}}$. We must satisfy three goals:

G1. $[\![a]\!]\ [pk_n] = \{[pk_n]\}$
G2. $[\![b]\!]\ [pk_1] = \{[pk_1]\}$
G3. $[pk_k \cdots pk_0] \in \mathsf{rng}[\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!]$

We can rewrite our hypothesis in terms of $\pi_{pk_0'}; \ldots; \pi_{pk_{m'}}$:

	Assertion	Reasoning
	$\alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1}; \in G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b)$	
$=$	$\alpha_{\pi_{pk_n}}; \pi_{pk_n} \diamond \alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \diamond \alpha_{\pi_{pk_1}}; \pi_{pk_1}$	
$\in$	$G(a) \diamond G(\mathsf{dup}; (p; t; \mathsf{dup})^*) \diamond G(b)$	By $\diamond$.

Hence, our hypothesis is comprised of three components:

- H1. $\alpha_{\pi_{pk_n}}; \pi_{pk_n} \in G(a)$
- H2. $\alpha_{\pi_{pk_1}}; \pi_{pk_1} \in G(b)$
- H3. $\alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$

G1 and G2. Both goals G1 and G2 follow from Lemma 13, H1, and H2.

G3.

	Assertion	Reasoning
	$[pk_k \cdots pk_0] \in \mathsf{rng}[\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!]$	
$\Rightarrow$	$\exists \alpha, \alpha_; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$	By Range to Language (Lemma 12).
$\Rightarrow$	$\alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$	By Remember First Hop (Lemma 11).

The goal G3 then follows from H3.

Direction $\Leftarrow$. We have that there exist $pk_1 \cdots pk_n$ such that the following hypotheses hold.

- H: $[pk_1 \cdots pk_n] \in \mathsf{rng}(\, [\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$
- H0: $[\![a]\!]\ [pk_n] = \{[pk_n]\}$
- H1: $[\![b]\!]\ [pk_1] = \{[pk_1]\}$

We must show that

$$\text{Goal: } a; \mathsf{dup}; (p; t; \mathsf{dup})^*; ; b \neq \mathsf{drop}$$

	Assertion	Reasoning
	$[pk_1 \cdots pk_n] \in \mathsf{rng}(\, [\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$	H.
$\Rightarrow$	$\exists \alpha . \alpha; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$	By Range to Language (Lemma 12).
$\Rightarrow$	$\alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$	By Remember First Hop (Lemma 11).
	$[\![a]\!]\ [pk_n] = \{[pk_n]\}$	H0.
$\Rightarrow$	$\alpha_{\pi_{pk_n}}; \pi_{pk_n} \in G(a)$	By Linguistic Test (Lemma 13).
	$[\![b]\!]\ [pk_1] = \{[pk_1]\}$	H1.
$\Rightarrow$	$\alpha_{\pi_{pk_1}}; \pi_{pk_1} \in G(b)$	By Linguistic Test (Lemma 13).
	$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \neq \mathsf{drop}$	Goal.
$\Rightarrow$	$G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b) \neq \{\}$	By Theorem 2
$\Rightarrow$	$\exists \alpha'; \pi_{pk_j}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_k}$	
$\in$	$G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; ; b)$	By the meaning of $\neq \{\}$.

Let $\alpha'; \pi_{pk_j}; \mathsf{dup} \cdots \pi_{pk_k} = \alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1}$. After substitution, we have:

	Assertion	**Reasoning**
	$\alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \in G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; ; b)$	
$=$	$\alpha_{\pi_{pk_n}}; \pi_{pk_n} \diamond \alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1} \diamond \alpha_{\pi_{pk_1}}; \pi_{pk_1}$	
$\in$	$G(a) \diamond G(\mathsf{dup}; (p; t; \mathsf{dup})^*;) \diamond G(b)$	By the definition of $\diamond$.

Hence, our goal is comprised of three subgoals.

- G1: $\alpha_{\pi_{pk_n}}; \pi_{pk_n} \in G(a)$
- G2: $\alpha_{\pi_{pk_n}}; \pi_{pk_n}; \mathsf{dup} \cdots \pi_{pk_1}; \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*;)$
- G3: $\alpha_{\pi_{pk_1}}; \pi_{pk_1} \in G(b)$

Subgoal G1 follows from H0, while G2 follows from H1, and G3 follows from H. □

Lemma 14 (Language to Applied Policy). *If $\alpha_{\pi_{pk}}; \pi_{pk} \diamond G(p) = \alpha_{\pi_{pk}}; \pi_{pk} \diamond G(q)$ then for all h, $[\![p]\!]\ [pk :: h] = [\![q]\!]\ [pk :: h]$.*

Proof. From the completeness theorem, we can rewrite our goal as follows.

$$\text{Goal: } \bigcup_{x \in G(p)} [\![x]\!]\ [pk :: h] = \bigcup_{y \in G(q)} [\![y]\!]\ [pk :: h]$$

The proof proceeds by cases on the value of x.

Case 1: $x = \alpha_{\pi_{pk}}; \pi_{pk}; \mathsf{dup}; \pi_{pk_m} \cdots \pi_{pk_n}$. Our goal is now to show:

$$\alpha_{\pi_{pk}}; \pi_{pk}; \mathsf{dup}; \pi_{pk_m} \cdots \pi_{pk_n}; \in \bigcup_{y \in G(q)} [\![y]\!]\ [pk :: h]$$

From the hypothesis, we know:

$$\alpha_{\pi_{pk}}; \pi_{pk} \diamond x = \alpha_{\pi_{pk}}; \pi_{pk} \diamond G(q)$$

Therefore $x = G(q)$.

Case 2: $x \neq \alpha_{\pi_{pk}}; \pi_{pk}; \mathsf{dup}; \pi_{pk_m} \cdots \pi_{pk_n}$. It follows from the value of x and the definition of the denotational semantics that:

$$[\![x]\!]\ [pk :: h] = \{\}$$

Hence, the following is undefined:

$$\alpha_{\pi_{pk}}; \pi_{pk} \diamond G(p)$$

It then follows from the hypothesis that the following is also undefined:

$$\alpha_{\pi_{pk}}; \pi_{pk} \diamond G(q)$$

Therefore:

$$[\![y]\!]\ [pk :: h] = \{\}$$

This satisfies our goal. □

Lemma 15 (Pass the Test). *If $\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G((p; t; \mathsf{dup})^n)$ and for all i, $\alpha_{\pi_{pk_i}}; \pi_{pk_i} \in G(a)$, then $\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(a; (p; t; \mathsf{dup})^n)$.*

Proof. The proof proceeds by induction on n.

Base case: n = 0. H: $\alpha_{\pi_{pk_1}}; \pi_{pk_2} \in G(1)$. Contradiction.

Inductive case. We have the following hypotheses:

- H1: $\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G((p; t; \mathsf{dup})^n)$
- H2: for all i, $\alpha_{\pi_{pk_i}}; \pi_{pk_i} \in G(a)$
- IH: $\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{n-1}} \in G(a; (p; t; \mathsf{dup})^{n-1})$

Our goal is

$$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(a; (p; t; \mathsf{dup})^n)$$

We can rewrite the goal using $\diamond$:

	Assertion	**Reasoning**
	$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{n-1}} \diamond \alpha_{\pi_{pk_{n-1}}}; \pi_{pk_{n-1}}; \mathsf{dup}; \pi_{pk_n}$	
$\in$	$G(a; (p; t; \mathsf{dup})^{n-1}) \diamond G(a; (p; t; \mathsf{dup}))$	By the definition of $\diamond$.

Hence, our goal is comprised of two subgoals:

- G1: $\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{n-1}} \in G(a; (p; t; \mathsf{dup})^{n-1})$
- G2: $\alpha_{\pi_{pk_{n-1}}}; \pi_{pk_{n-1}}; \mathsf{dup}; \pi_{pk_n} \in G(a; (p; t; \mathsf{dup}))$

The subgoal G1 follows immediately from IH. Moving next to G2, we can again rewrite using $\diamond$.

	Assertion	**Reasoning**
	$\alpha_{\pi_{pk_{n-1}}}; \pi_{pk_{n-1}} \diamond \alpha_{\pi_{pk_{n-1}}}; \pi_{pk_{n-1}}; \mathsf{dup}; \pi_{pk_n}$	
$\in$	$G(a) \diamond G(p; t; \mathsf{dup})$	By the definition of $\diamond$.

Hence, we again have two subgoals:

- G1′: $\alpha_{\pi_{pk_{n-1}}}; \pi_{pk_{n-1}} \in G(a)$
- G2′: $\alpha_{\pi_{pk_{n-1}}}; \pi_{pk_{n-1}}; \mathsf{dup}; \pi_{pk_n} \in G(p; t; \mathsf{dup})$

The subgoal G1′ follows immediately from H2. Next, note that we can rewrite H1 as follows:

$$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{n-1}} \diamond \alpha_{\pi_{pk_{n-1}}}; \pi_{pk_{n-1}}; \mathsf{dup}; \pi_{pk_n} \in G((p; t; \mathsf{dup})^{n-1}) \diamond G(p; t; \mathsf{dup})$$

The subgoal G2′ follows from H1. □

Definition 2 (Waypoint)**.** *For predicates a, b, and w, packets from a to b are* waypointed *through w if and only if for all packet histories $\langle pk_1 \cdots pk_n \rangle \in \mathit{rng}(\llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket)$ where $\llbracket a \rrbracket \langle pk_n \rangle = \{\langle pk_n \rangle\}$ and $\llbracket b \rrbracket \langle pk_1 \rangle = \{\langle pk_1 \rangle\}$, there exists $pk_x \in \langle pk_1 \cdots pk_n \rangle$ such that $\llbracket w \rrbracket \langle pk_x \rangle = \{\langle pk_x \rangle\}$ and for all i where $1 < i < x$, we have $\llbracket b \rrbracket\ pk_i = \{\}$, and for all i where $x < i < n$, we have $\llbracket a \rrbracket\ pk_i = \{\}$.*

Theorem 2 (Waypointing)**.** *For all user policies p, topologies t, tests a and b, and waypoints w, w is a semantic waypoint of p and t between a and b if, and only if,*

$$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \subseteq a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b$$

Proof. We first unfold the definition of *waypoint* in the theorem statement. We swap the order of the statement for readability.

$$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \subseteq a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b$$

if and only if, for all packet histories $\langle pk_1 \cdots pk_n \rangle$, if

- $\langle pk_1 \cdots pk_n \rangle \in \mathsf{rng}(\, [\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$,
- $[\![a]\!]\, \langle pk_n \rangle = \{\langle pk_n \rangle\}$, and
- $[\![b]\!]\, \langle pk_1 \rangle = \{\langle pk_1 \rangle\}$

then there exists and x, such that

- $pk_x \in \langle pk_1 \cdots pk_n \rangle$,
- $[\![w]\!]\, \langle pk_x \rangle = \{\langle pk_x \rangle\}$,
- $\forall i. 1 < i < x \Rightarrow [\![b]\!]\ pk_i = \{\}$, and
- $\forall i. x < i < n \Rightarrow [\![a]\!]\ pk_i = \{\}$

Direction $\Leftarrow$. We first simplify the goal as follows.

	Assertion	**Reasoning**
Goal:	$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \subseteq a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg b; p; t; \mathsf{dup})^*; b$	
=	$\forall [pk :: h].\ [\![a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	
	$\subseteq [\![a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg b; p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	by definition of $\subseteq$
$\Leftarrow$	$G(\pi_{pk}) \diamond G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b)$	
	$\subseteq G(\pi_{pk}) \diamond G(a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg b; p; t; \mathsf{dup})^*; b)$	by Lemma 14
=	$G(\pi_{pk}) \diamond G(a) \diamond G(\mathsf{dup}; (p; t; \mathsf{dup})^*) \diamond G(b)$	
	$\subseteq G(\pi_{pk}) \diamond G(a) \diamond G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg b; p; t; \mathsf{dup})^*) \diamond G(b)$	by definition of $\diamond$
$\Leftarrow$	$G(\mathsf{dup}; (p; t; \mathsf{dup})^*) \subseteq G(\mathsf{dup}; (\neg q; p; t; \mathsf{dup})^*; w; (\neg q; p; t; \mathsf{dup})^*)$	by definition of $\diamond$
=	$\forall \alpha pk_{m1} \cdots pk_{mn}.$	
	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*) \Rightarrow$	
	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg b; p; t; \mathsf{dup})^*)$	by definition of $\subseteq$

Case 1. $[pk :: h]$ satisfies the conditions of Definition 2. The hypotheses are thus:

H1. $[pk :: h] \in \mathsf{rng}(\, [\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$
H2. $[\![a]\!]\ [pk_1] = \{[pk_1]\}$
H3. $[\![b]\!]\ [pk_n] = \{[pk_n]\}$
H4. There exists an x, such that:
- $1 < x < n$,
- $[\![w]\!]\ \langle pk_x \rangle = \{\langle pk_x \rangle\}$,
- $\forall i. 1 < i < x\ [\![b]\!]\ \langle pk_i \rangle = \emptyset$, and
- $\forall i. x < i < n\ [\![a]\!]\ pk_n = \emptyset$

H5. $\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$

We can rewrite hypotheses 1-3 as follows:

H1′. $\alpha_{\pi_{pk_1}}; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^*)$
H2′. $\alpha_{\pi_{pk_1}}; \pi_{pk_1} \in G(a)$
H3′. $\alpha_{\pi_{pk_n}}; \pi_{pk_n} \in G(b)$

Transform the goal as follows:

	Assertion	Reasoning
	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}}$ $\in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*)$	
$\Rightarrow$	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}}$ $\in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^{n_1}; w; (\neg a; p; t; \mathsf{dup})^{n_2})$	$\exists n_1, n_2$ by definition of *
$\Rightarrow$	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mn}}$ $\in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^{z}; w; (\neg a; p; t; \mathsf{dup})^{n-z})$	let $z = n_1$ and by $n_1 + n_2 = n$
$\Rightarrow$	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}} \diamond \alpha_{\pi_{pk_{mz}}}; \pi_{pk_{mz}} \cdots \mathsf{dup}; \pi_{pk_{mn}}$ $\in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^{z}) \diamond G(w; (\neg a; p; t; \mathsf{dup})^{n-z})$	by definition of $\diamond$

Transform **H5** to **H5′** as follows:

	Assertion	Reasoning
	$\alpha_{\pi_{pk_{m1}}}; \pi_{pk_{mz}} \cdots \mathsf{dup}; \pi_{pk_{mn}}$ $\in G(\mathsf{dup}; (p; t; \mathsf{dup})^{n})$	by definition of *, $\exists n$
$\Rightarrow$	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}} \diamond \alpha_{\pi_{pk_{mz}}}; \pi_{pk_{mz}} \cdots \mathsf{dup}; \pi_{pk_{mn}}$ $\in G(\mathsf{dup}; (p; t; \mathsf{dup})^{n})$	by definition of $\diamond$
$\Rightarrow$	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}} \diamond \alpha_{\pi_{pk_{mz}}}; \pi_{pk_{mz}} \cdots \mathsf{dup}; \pi_{pk_{mn}}$ $\in G(\mathsf{dup}; (p; t; \mathsf{dup})^{z}) \diamond G(\mathsf{dup}; (p; t; \mathsf{dup})^{n-z})$	since $z \leq n$

Now we can define two subgoals:

G1. $\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}} \in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^{z})$
G2. $\alpha_{\pi_{pk_{mz}}}; \pi_{pk_{mz}} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(w; (\neg a; p; t; \mathsf{dup})^{n-z})$

Proof of **G1**:

	Assertion	Reasoning
	$\alpha_{\pi_{pk_{m1}}}; \pi_{pk_{m1}}; \mathsf{dup}; \pi_{pk_{m1}} \diamond \alpha_{\pi_{pk_{m1}}}; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}}$ $\in G(\mathsf{dup}) \diamond G((p; t; \mathsf{dup})^{z})$	by **H5′**
$\Rightarrow$	$\alpha; \pi_{pk_{m1}}; \mathsf{dup}; \pi_{pk_{m1}} \diamond \alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}}$ $\in G(\mathsf{dup}) \diamond G((\neg b; p; t; \mathsf{dup})^{z})$	by **H4**
$\Rightarrow$	$\alpha_{\pi_{pk_{m1}}}; \pi_{pk_{m1}}; \mathsf{dup}; \pi_{pk_{m1}} \diamond \alpha_{\pi_{pk_{m1}}}; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}}$ $\in G(\mathsf{dup}) \diamond G((\neg b; p; t; \mathsf{dup})^{z})$	by $\diamond$
$\Rightarrow$	$\alpha; \pi_{pk_{m1}}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_{mz}}$ $\in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^{z})$	by $\diamond$

Proof of **G2**:

Assertion	Reasoning
$\alpha_{\pi_{pk_{mz}}}; \pi_{pk_{mz}} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(\mathsf{dup}; (p; t; \mathsf{dup})^{n-z})$	by **H5**
$\alpha_{\pi_{pk_{mz}}}; \pi_{pk_{mz}} \cdots \mathsf{dup}; \pi_{pk_{mn}} \in G(w; (\neg a; p; t; \mathsf{dup})^{n-z})$	by **H4**

Case 2. $[\![a]\!]\ [pk_n \cdots pk_1] \neq [pk_n \cdots pk_1]$

	Assertion	Reasoning
	$[\![a]\!]\ [pk_n \cdots pk_1] \neq [pk_n \cdots pk_1]$	case 2 condition
$\Rightarrow$	$[\![a]\!]\ [pk_n \cdots pk_1] = \emptyset$	since a is a predicate
$\Rightarrow$	$[\![a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	
	$\subseteq [\![a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	by definition of $\bullet$

Case 3. $[\![b]\!]\ [pk_n \cdots pk_1] \neq [pk_n \cdots pk_1]$

	Assertion	Reasoning
	$[\![b]\!]\ [pk_n \cdots pk_1] \neq [pk_n \cdots pk_1]$	case 3 condition
$\Rightarrow$	$[\![b]\!]\ [pk_n \cdots pk_1] = \emptyset$	since b is a predicate
$\Rightarrow$	$[\![a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	
	$\subseteq [\![a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	by definition of $\bullet$

Case 4. There does not exist $[pk_n \cdots pk_1] \in \mathsf{rng}([\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$.

	Assertion	Reasoning
	There does not exist $[pk_n \cdots pk_1] \in \mathsf{rng}([\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$	case 4 condition
$\Rightarrow$	$\forall pk, h.\ [\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!]\ [pk :: h] = \emptyset$	by definition
$\Rightarrow$	$[\![a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	
	$\subseteq [\![a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b]\!]\ [pk :: h]$	by definition of $\bullet$

Direction $\Rightarrow$. If:

H1. $a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \subseteq a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b$
H2. $\langle pk_2 \cdots pk_n \rangle \in \mathsf{rng}(\ [\![\mathsf{dup}; (p; t; \mathsf{dup})^*]\!])$
H3. $[\![a]\!]\ [pk_1] = \{[pk_1]\}$
H4. $[\![b]\!]\ [pk_n] = \{[pk_n]\}$

then there exists a x such that:

G1. $1 < x < n$,
G2. $[\![w]\!]\ [pk_x] = \{[pk_x]\}$,
G3. $\forall i. 1 < i < x\ [\![b]\!]\ [pk_i] = \emptyset$, and
G4. $\forall i. x < i < n\ [\![a]\!]\ [pk_n] = \emptyset$

Simplify $H1$:

	Assertion	Reasoning
	$a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b \subseteq a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b$	definition of H1
=	$G(a; \mathsf{dup}; (p; t; \mathsf{dup})^*; b) \subseteq G(a; \mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*; b)$	by theorem 2
=	$G(a) \diamond G(\mathsf{dup}; (p; t; \mathsf{dup})^*;) \diamond G(b)$	
	$\subseteq G(a) \diamond G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*;) \diamond G(b)$	by definition of $\diamond$
$\Rightarrow$	$G(\mathsf{dup}; (p; t; \mathsf{dup})^*;) \subseteq G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*;)$	by definition of $\diamond$

Simplify $H2$:

	Assertion	Reasoning
	$[pk :: h] \in \mathsf{rng}(\llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket)$	definition of $H2$
$\Rightarrow$	$\exists \alpha. \alpha; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket)$	by Lemma 12
$=$	$\alpha_{\pi_{pk_1}}; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket)$	by Lemma 11

We can rewrite our hypotheses as follows:

H3. $\alpha_{\pi_{pk_1}}; \pi_{pk_1} \in G(a)$
H4. $\alpha_{\pi_{pk_n}}; \pi_{pk_n} \in G(b)$

From H2 and H1′, we can deduce a new hypothesis:

	Assertion	Reasoning
	$\alpha_{\pi_{pk_1}}; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\llbracket \mathsf{dup}; (p; t; \mathsf{dup})^* \rrbracket)$	definition of $H2'$
$\Rightarrow$	$\alpha_{\pi_{pk_1}}; \pi_{pk_1}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G(\mathsf{dup}; (\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*;)$	definition of $H1'$
$\Rightarrow$	$\alpha_{\pi_{pk_1}}; \pi_{pk_1}; \mathsf{dup}; \pi_{pk_1} \diamond \alpha_{\pi_{pk_1}}; pk_2; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n}$	
	$\in G(\mathsf{dup}) \diamond G((\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*;)$	by $\diamond$
$\Rightarrow$	$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G((\neg b; p; t; \mathsf{dup})^*; w; (\neg a; p; t; \mathsf{dup})^*;)$	by $\diamond$
$\Rightarrow$	$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_x} \diamond \alpha_{\pi_{pk_x}}; \pi_{pk_x}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n}$	
	$\in G((\neg b; p; t; \mathsf{dup})^*) \diamond G(w; (\neg a; p; t; \mathsf{dup})^*;)$	by $\diamond$
$\Rightarrow$	$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_x} \diamond \alpha_{\pi_{pk_x}}; \pi_{pk_x} \diamond \alpha_{\pi_{pk_x}}; \pi_{pk_x}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n}$	
	$\in G((\neg b; p; t; \mathsf{dup})^*) \diamond G(w) \diamond G((\neg a; p; t; \mathsf{dup})^*)$	by $\diamond$
$\Rightarrow$	$\exists n_1 n_2. \alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_x} \diamond \alpha_{\pi_{pk_x}}; \pi_{pk_x}$	
	$\diamond \alpha_{\pi_{pk_x}}; \pi_{pk_x}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n}$	
	$\in G((\neg b; p; t; \mathsf{dup})^{n_1}) \diamond G(w) \diamond G((\neg a; p; t; \mathsf{dup})^{n_2})$	by definitinon of *
$\Rightarrow$	$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_x} \diamond \alpha_{\pi_{pk_x}}; \pi_{pk_x} \diamond \alpha_{\pi_{pk_x}}; \pi_{pk_x}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n}$	
	$\in G((\neg b; p; t; \mathsf{dup})^{n_1}) \diamond G(w) \diamond G((\neg a; p; t; \mathsf{dup})^{n_2})$	introduce n_1 and n_2
$\Rightarrow$	**H5:** $\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_x} \in G((\neg b; p; t; \mathsf{dup})^{n_1}) \wedge$	
	H6: $\alpha_{\pi_{pk_x}}; \pi_{pk_x} \in G(w) \wedge$	
	H7: $\alpha_{\pi_{pk_x}}; \pi_{pk_x}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G((\neg a; p; t; \mathsf{dup})^{n_2})$	immediate

Let x in the goal be n_1 defined above.

Proof of G1. We have that $1 \leq x \leq n$. If $x = 1$, then $\alpha_{\pi_{pk_1}} \pi_{pk_1}; \in G(a)$ and $\alpha_{\pi_{pk_1}} \pi_{pk_1}; \in G(\neg a)$, which is a contradiction. Similarly, if $x = n$, then $\alpha_{\pi_{pk_n}} \pi_{pk_n}; \in G(b)$ and $\alpha_{\pi_{pk_n}} \pi_{pk_n}; \in G(\neg b)$, which is a contradiction.

Proof of G2.

	Assertion	Reasoning
	$\alpha_{\pi_{pk_x}}; \pi_{pk_x} \in G(w)$	H6
$\Rightarrow$	$\llbracket w \rrbracket \langle pk_x \rangle = \langle pk_x \rangle$	by lemma 13

Proof of G3.

$$
\begin{array}{l}
(\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0} ::= \\
\quad \mathsf{let}\ pre = (\mathsf{vlan} = w_0; in; \mathsf{vlan} \leftarrow w + \mathsf{vlan} = w)\ \mathsf{in} \\
\quad \mathsf{let}\ post = (out; \mathsf{vlan} \leftarrow w_0 + \neg out)\ \mathsf{in} \\
\quad pre; p; post
\end{array}
$$

Figure 2. Slice desugaring.

	Assertion	**Reasoning**
	$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_x} \in G((\neg b; p; t; \mathsf{dup})^{n_1})$	H5
=	$\alpha_{\pi_{pk_1}}; \pi_{pk_2}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_x} \in G((\neg b; p; t; \mathsf{dup})^{x})$	since $x = n_1$
$\Rightarrow$	$\forall i. \alpha_{\pi_{pk_i}}; \pi_{pk_i} \in G(\neg b)$	by induction on x
$\Rightarrow$	$\forall i. 1 < i < x\ [\![b]\!]\ [pk_i] = \emptyset$	by lemma 13

Proof of G4.

	Assertion	**Reasoning**
	$\alpha_{\pi_{pk_x}}; \pi_{pk_x}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G((\neg a; p; t; \mathsf{dup})^{n_2})$	H7
=	$\alpha_{\pi_{pk_x}}; \pi_{pk_x}; \mathsf{dup} \cdots \mathsf{dup}; \pi_{pk_n} \in G((\neg a; p; t; \mathsf{dup})^{n-x})$	since $n_2 = n - x$
$\Rightarrow$	$\forall i. \alpha_{\pi_{pk_i}}; \pi_{pk_i} \in G(\neg a)$	by induction on $n - x$
$\Rightarrow$	$\forall i. x < i < n\ [\![a]\!]\ [pk_n] = \emptyset$	by lemma 13 $\square$

C. Application: Isolation

This section contains the full proofs and supporting lemmas for theorems in Section 6.

As Figure 2 shows, the desugared translation of slice policies relies heavily on predicates on worlds, which we have written $\mathsf{vlan} = w$. For the rest of this section, we will sometimes abbreviate this as simply w. Hence, a policy written $w; p$ is equivalent to $\mathsf{vlan} = w; p$. We also rely on a *denesting* lemma, drawn from [12], that shows how to transform a summation under a star into a series of conjunctions and stars.

Lemma 16 (Denesting)**.**

$$p^*(qp^*)^* \equiv (p + q)^*$$

Proof. Proposition 7 in [12].

$\square$

Slice policies and predicates must be tag-free. Intuitively, tag-freedom asserts that policies and predicates neither test nor modify the tag field. Formally, we use the following definitions.

Definition 3 (tag-free Policy)**.** *A policy p is* tag*-free when it commutes with any test of the slice field:*

$$\textit{For all } w, \mathsf{tag} = w; p \equiv p; \mathsf{tag} = w.$$

Definition 4 (tag-free Predicate)**.** *A predicate b is* tag*-free when it commutes with any modification of the* tag *field:*

$$\textit{For all } w, \mathsf{tag} \leftarrow w; b \equiv b; \mathsf{tag} \leftarrow w.$$

Unsurprisingly, certain commutativity properties also hold on the topology. In particular, the topology may only modify the location information associated with each packet but not the packet itself. Hence, any tests on the packet commute with the topology.

Lemma 17 (Topology Preserves Packets)**.** *For all topologies t and predicates b, if b does not include tests of the form $\mathsf{sw} = v$ for any value v, then $b; t \equiv t; b$.*

Proof. By induction on the structure of t. The base case follows immediately from KA-Seq-Zero. The inductive case is $t = \mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + t'$.

	Assertion	**Reasoning**
	$b; (\mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + t')$	
	$\equiv b; \mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + b; t'$	KA-Seq-Dist-L.
	$\equiv \mathsf{sw} = sw; \mathsf{pt} = pt; b; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt' + b; t'$	BA-Seq-Comm.
	$\equiv \mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt'; b + b; t'$	KA-Match-PA-Mod-Mod-Comm.
	$\equiv \mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt'; b + t'; b$	IH.
Goal	$\equiv (\mathsf{sw} = sw; \mathsf{pt} = pt; \mathsf{sw} \leftarrow sw'; \mathsf{pt} \leftarrow pt'; +t'); b$	KA-Seq-Dist-R.

□

Suppose there exist two slices. The first only emits packets that, after traversing the topology, do not match the ingress predicate of the second. A sequence composed of the first slice, the topology, and the second slice is equivalent to drop.

Lemma 18 (No Slice Sequencing)**.** *For all slice ingress and egress predicates in_1, out_1, in_2, out_2, slice identifiers w_1, w_2, and policies s_1, s_2, p, q, and topologies t, such that*

- $s_1 \equiv (\!|\{in_1\}\ w_1 : (p)\ \{out_1\}|\!)^{w_0}$,
- $s_2 \equiv (\!|\{in_2\}\ w_2 : (q)\ \{out_2\}|\!)^{w_0}$,
- *H0:* $w_1 \neq w_2$,
- *H1: p, q, in_1, in_2, out_1, out_2 are all VLAN-free.*
- *H2:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$,

then the following equivalence holds:

$$s_1; t; \mathsf{dup}; s_2 \equiv \mathsf{drop}$$

Proof. First, note that s_1 and s_2 have the following desugared forms.

$$\begin{aligned}
pre_1 &= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1) \\
post_1 &= (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \\
s_1 &= pre_1; p; post_1 \\
&= (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1)
\end{aligned}$$

$$\begin{aligned}
pre_2 &= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2) \\
post_2 &= (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \\
s_2 &= pre_2; q; post_2 \\
&= (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2)
\end{aligned}$$

	Assertion	Reasoning
	$s_1; t; \mathsf{dup}; s_2$	
$\equiv$	$pre_1; p; post_1; t; \mathsf{dup}; pre_2; q; post_2$	Substitution for s_1 and s_2.
$\equiv$	$pre_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1); t; \mathsf{dup}; pre_2; q; post_2$	Substitution for $post_1$.
$\equiv$	$(w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1);$ $t; \mathsf{dup}; pre_2; q; post_2$	Substitution for pre_1.
$\equiv$	$(w_0; in_1; \mathsf{vlan} \leftarrow w_1 + \mathsf{id}); w_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1);$ $t; \mathsf{dup}; pre_2; q; post_2$	PA-Mod-Filter and KA-Seq-Dist-R.
$\equiv$	$(w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); w_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1);$ $t; \mathsf{dup}; pre_2; q; post_2$	BA-Seq-Idem, PA-Mod-Filter, and KA-Seq-Dist-R.
$\equiv$	$pre_1; w_1; p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1); t; \mathsf{dup}; pre_2; q; post_2$	Substitution for pre_1.
$\equiv$	$pre_1; p; w_1; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1); t; \mathsf{dup}; pre_2; q; post_2$	H1.
$\equiv$	$pre_1; p; w_1; out_1; \mathsf{vlan} \leftarrow w_0; t; \mathsf{dup}; pre_2; q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	KA-Seq-Dist-L.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0; out_1; t; \mathsf{dup}; pre_2; q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	H1.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0; w_0; out_1; t; \mathsf{dup}; pre_2; q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	PA-Mod-Filter.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0; w_0; out_1; t; \mathsf{dup};$ $(w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	Substitution for pre_2.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0;$ $(w_0; out_1; t; \mathsf{dup}; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_0; out_1; t; \mathsf{dup}; w_2); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	KA-Seq-Dist-L.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0;$ $(w_0; out_1; t; \mathsf{dup}; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	H1, Lemma 17, and PA-Contra.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0;$ $(w_0; out_1; t; \mathsf{dup}; in_2; w_0; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	BA-Seq-Comm.
$\equiv$	$pre_1; p; w_1; \mathsf{vlan} \leftarrow w_0; (w_0; \mathsf{drop}; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$ $+ pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	H2.
$\equiv$	$pre_1; p; w_1; \neg out_1; t; \mathsf{dup}; pre_2; q; post_2$	KA-Seq-Zero, KA-Zero-Seq, and KA-Plus-Zero.
$\equiv$	$pre_1; p; \neg out_1; t; \mathsf{dup}; w_1; pre_2; q; post_2$	H1 and Lemma 17.
$\equiv$	$pre_1; p; \neg out_1; t; \mathsf{dup}; w_1; (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2$	Substitution for pre_2.
$\equiv$	$pre_1; p; \neg out_1; t; \mathsf{dup}; (w_1; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_1; w_2); q; post_2$	KA-Seq-Dist-L.
$\equiv$	drop	PA-Contra, KA-Seq-Zero, KA-Zero-Seq, and Plus-Zero. $\square$

Now, suppose there are two slices that neither admit the same packets nor does one eject any packets the other may inject. Every packet that enters the network will either be admitted to the first slice, or the second, or dropped. Intuitively, if the two slices are indeed isolated, then if we restrict the packets that enter the network to only those that will be injected into the first slice, then running both slices together should be equivalent to running the first slice alone.

Theorem 4 (Slice/Slice Isolation)**.** *For all slice ingress and egress predicates in_1, out_1, in_2, out_2, slice identifiers w_1, w_2, and policies s_1, s_2, p, q, and topologies t, such that*

- $s_1 \equiv (\!|\{in_1\}\ w_1 : (p)\ \{out_1\}|\!)^{w_0}$,
- $s_2 \equiv (\!|\{in_2\}\ w_2 : (q)\ \{out_2\}|\!)^{w_0}$,
- *H0:* $w_1 \neq w_2$,
- *H1: p, q, in_1, in_2, out_1, out_2 are all VLAN-free.*
- *H2:* $in_1; in_2 \equiv \mathsf{drop}$,
- *H3:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$,
- *H4:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$,
- *H5:* $out_1; out_2 \equiv \mathsf{drop}$,

then the following equality holds:

$$w_0; in_1; (s_1; t; \mathsf{dup})^* \equiv w_0; in_1; ((s_1 + s_2); t; \mathsf{dup})^*$$

Proof. First, note that s_1 and s_2 have the following desugared forms.

$$\begin{array}{lcl} pre_1 & = & (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1) \\ post_1 & = & (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \\ s_1 & = & pre_1; p; post_1 \\ & = & (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \end{array}$$

$$\begin{array}{lcl} pre_2 & = & (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2) \\ post_2 & = & (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \\ s_2 & = & pre_2; q; post_2 \\ & = & (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \end{array}$$

Next, note that the following equivalence holds, which we will call L1: $in_1; w_0; s_2 \equiv \mathsf{drop}$.

	Assertion	**Reasoning**
L1	$in_1; w_0; s_2$	
	$\equiv in_1; w_0; pre_2; q; post_2$	Substitution for s_2.
	$\equiv in_1; w_0; (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2$	Substitution for pre_2.
	$\equiv (in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + in_1; w_0; w_2); q; post_2$	KA-Seq-Dist-L.
	$\equiv (in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + \mathsf{drop}); q; post_2$	PA-Contra and KA-Seq-Zero.
	$\equiv in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2; q; post_2$	KA-Plus-Zero.
	$\equiv in_1; in_2; w_0; w_0; \mathsf{vlan} \leftarrow w_2; q; post_2$	BA-Seq-Comm.
	$\equiv \mathsf{drop}; w_0; w_0; \mathsf{vlan} \leftarrow w_2; q; post_2$	H2.
	$\equiv \mathsf{drop}$	KA-Zero-Seq.

With L1, we can now show that $w_0; in_1; (s_1; t; \mathsf{dup})^* \equiv w_0; in_1; (s_1 + s_2; t; \mathsf{dup})^*$.

	Assertion	Reasoning
	$w_0; in_1; ((s_1 + s_2); t; \mathsf{dup})^*$	
$\equiv$	$in_1; w_0; ((s_1 + s_2); t; \mathsf{dup})^*$	BA-Seq-Comm
$\equiv$	$in_1; w_0; (s_1; t + s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-R.
$\equiv$	$in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 16.
$\equiv$	$in_1; w_0; (\mathsf{id} + (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup})); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Star-Unroll-R.
$\equiv$	$in_1; w_0; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	
	$+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L, KA-Seq-Dist-R, and KA-Seq-One.
$\equiv$	$in_1; w_0; (\mathsf{id} + (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*)$	
	$+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Star-Unroll-L.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*)$	
	$+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-R.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	L1 and KA-Plus-Zero.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup});$	
	$(\mathsf{id} + (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*); (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Star-Unroll-L.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup})$	
	$+in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}); s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*;$	
	$(s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup})$	
	$+in_1; w_0; (s_1; t; \mathsf{dup})^*; \mathsf{drop}; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 18.
$\equiv$	$in_1; w_0 + in_1; w_0; (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup})$	KA-Seq-Zero and KA-Zero-Seq.
$\equiv$	$in_1; w_0; (\mathsf{id} + (s_1; t; \mathsf{dup})^*; (s_1; t; \mathsf{dup}))$	KA-Seq-Dist-L.
$\equiv$	$in_1; w_0; (s_1; t; \mathsf{dup})^*$	KA-Star-Unroll-R.
$\equiv$	$w_0; in_1; (s_1; t; \mathsf{dup})^*$	BA-Seq-Comm. □

C.1 Shared Inbound Edges

In the previous section, we showed that two slices with unshared edges are isolated when composed. Next, we relax the restriction on slice edges: given two slices, s_1 and s_2, the ingresses of the two slices may overlap, and so may the egresses. Intuitively, this captures the case where a packet may be copied and processed by both slices simultaneously. Clearly the behavior of the slices, when composed, is not equivalent to one of the slices acting alone; but neither can one slice interfere with the copy of the packet traversing the other slice.

Theorem 6 (Slice/Slice Composition). *For all* tag*-free slice ingress and egress predicates in_1, out_1, in_2, out_2, identifiers w_1, w_2, policies s_1, s_2,* tag*-free policies p_1, p_2, and topologies t, such that*

- $s_1 = (\!|\{in_1\}\ w_1 : (p_1)\ \{out_1\}|\!)^{w_0}$,
- $s_2 = (\!|\{in_2\}\ w_2 : (p_2)\ \{out_2\}|\!)^{w_0}$,
- *H0:* $w_1 \neq w_2, w_1 \neq w_0, w_2 \neq w_0$
- *H1:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$*, and*
- *H2:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$*, then*

$$((s_1 + s_2); t; \mathsf{dup})^* \equiv (s_1; t; \mathsf{dup})^* + (s_2; t; \mathsf{dup})^*.$$

Proof. First, note that s_1 and s_2 have the following desugared forms.

$$\begin{array}{lcl} pre_1 & = & (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1) \\ post_1 & = & (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \\ s_1 & = & pre_1; p; post_1 \\ & = & (w_0; in_1; \mathsf{vlan} \leftarrow w_1 + w_1); p; (out_1; \mathsf{vlan} \leftarrow w_0 + \neg out_1) \end{array}$$

$$\begin{array}{lcl} pre_2 & = & (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2) \\ post_2 & = & (out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \\ s_2 & = & pre_2; q; post_2 \\ & = & (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q(out_2; \mathsf{vlan} \leftarrow w_0 + \neg out_2) \end{array}$$

	Assertion	**Reasoning**
	$((s_1 + s_2); t; \mathsf{dup})^*$	
$\equiv$	$(s_1; t + s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-R.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 16.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup}; (\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*))^*$	KA-Star-Unroll-L.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t + s_2; t; \mathsf{dup}; s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L, KA-Seq-One.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t + \mathsf{drop}; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*)^*$	Lemma 18.
$\equiv$	$(s_1; t; \mathsf{dup})^*; (s_2; t; \mathsf{dup})^*$	KA-Zero-Seq, KA-Plus-Zero.
$\equiv$	$(\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*); (\mathsf{id} + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*)$	KA-Star-Unroll-L.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	
	$+s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^*; s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-L, KA-Seq-Dist-R. KA-One-Seq, KA-Seq-One.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	
	$+s_1; t; \mathsf{dup}; (\mathsf{id} + (s_1; t; \mathsf{dup})^*; s_1; t; \mathsf{dup});$	
	$s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Star-Unroll-R.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	
	$+s_1; t; \mathsf{dup}; s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	
	$+(s_1; t; \mathsf{dup})^*; s_1; t; \mathsf{dup}; s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Seq-Dist-R, KA-One-Seq, KA-Star-Unroll-L.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	Lemma 18, KA-Zero-Seq, KA-Seq-Zero, KA-Plus-Zero.
$\equiv$	$\mathsf{id} + s_1; t; \mathsf{dup}; (s_1; t; \mathsf{dup})^* + \mathsf{id} + s_2; t; \mathsf{dup}; (s_2; t; \mathsf{dup})^*$	KA-Plus-Idem, KA-Plus-Comm.
$\equiv$	$(s_1; t; \mathsf{dup})^* + (s_2; t; \mathsf{dup})^*$	KA-Star-Unroll-L. $\square$

Finally, we show that Theorem 4 follows from Theorem 6.

Corollary 1. *For all* tag*-free slice ingress and egress predicates in_1, out_1, in_2, out_2, identifiers w_1, w_2, policies s_1, s_2,* tag*-free policies p_1, p_2, and topologies t, such that*

- $s_1 = (\!|\{in_1\}\ w_1 : (p_1)\ \{out_1\}|\!)^{w_0}$,
- $s_2 = (\!|\{in_2\}\ w_2 : (p_2)\ \{out_2\}|\!)^{w_0}$,
- *H0:* $w_1 \neq w_2, w_1 \neq w_0, w_2 \neq w_0$
- *H1:* $out_1; t; \mathsf{dup}; in_2 \equiv \mathsf{drop}$,
- *H2:* $out_2; t; \mathsf{dup}; in_1 \equiv \mathsf{drop}$,

- *H3:* $in_1; in_2 \equiv \mathsf{drop}$*, then*

$$\begin{aligned} & in_1; \mathsf{tag} = w_0; ((s_1 + s_2); t; \mathsf{dup})^* \\ \equiv\ & in_1; \mathsf{tag} = w_0; (s_1; t; \mathsf{dup})^* \end{aligned}$$

Proof. By Theorem 6 and KA-Seq-Dist-L, we have $in_1; w_0; (s_1)^* + in_1; w_0; (s_2)^*$.

	Assertion	**Reasoning**
	$in_1; w_0; (s_1)^* + in_1; w_0; (s_2)^*$	
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0 + in_1; w_0; s_2; (s_2)^*$	KA-Star-Unroll-L and KA-Seq-Dist-L.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$ $+ in_1; w_0; (w_0; in_2; \mathsf{vlan} \leftarrow w_2 + w_2); q; post_2; (s_2)^*$	Substitution for $\equiv s_2$.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$ $+ (in_1; w_0; w_0; in_2; \mathsf{vlan} \leftarrow w_2 + in_1; w_0; w_2); q; post_2; (s_2)^*$	KA-Seq-Dist-L.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$ $+ (in_1; in_2; w_0; w_0; \mathsf{vlan} \leftarrow w_2 + in_1; w_0; w_2); q; post_2; (s_2)^*$	BA-Seq-Comm.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$ $+ (\mathsf{drop}; w_0; w_0; \mathsf{vlan} \leftarrow w_2 + in_1; \mathsf{drop}); q; post_2; (s_2)^*$	H4 and PA-Contra.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0 + \mathsf{drop}$	KA-Seq-Zero, KA-Zero-Seq, and KA-Plus-Zero.
$\equiv$	$in_1; w_0; (s_1)^* + in_1; w_0$	KA-Plus-Zero.
$\equiv$	$in_1; w_0 + in_1; w_0; s_1; (s_1)^* + in_1; w_0$	KA-Star-Unroll-L.
$\equiv$	$in_1; w_0 + in_1; w_0; s_1; (s_1)^*$	KA-Plus-Comm and KA-Plus-Idem.
$\equiv$	$in_1; w_0; (s_1)^*$	KA-Star-Unroll-L. □

C.2 Weakening the Hypotheses

The hypotheses of Theorem 4 restrict its application to two slices running alone on the network. While such a result provides insight into the nature of slice interaction, we show a stronger result in this section that demonstrates slice isolation in the presence of other slices and activity in the network.

In particular, slices drop traffic with any tag not their own or w_0—this prevents them from interfering with traffic that has been injected into another slice. In turn, if a slice is compiled with a tag w, it can run in isolation on the same network as any NetKATuser policy, so long as that policy drops w-tagged traffic.

Definition 5 (Dropping w-tagged Traffic)**.** *A policy p drops w-tagged traffic when*

- $\mathsf{tag} = w; p \equiv \mathsf{drop}$*, and*
- $p; \mathsf{tag} = w \equiv \mathsf{drop}$*.*

Lemma 19 (No Program Sequencing)**.** *For all slice ingress and egress predicates in, out, slice identifiers w, and policies s, p, q, and topologies t, such that*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$*,*
- *H0:* $w \neq w_0$*,*
- *H1: p, q, in, and out are tag-free,*
- *H2:* $out; t; \mathsf{dup}; q \equiv \mathsf{drop}$ *and* $q; t; \mathsf{dup}; in \equiv \mathsf{drop}$*,*
- *H3: q drops w-tagged traffic,*

then the following equivalences hold:

$$\begin{aligned} s; t; \mathsf{dup}; q &\equiv \mathsf{drop} \\ q; t; \mathsf{dup}; s &\equiv \mathsf{drop} \end{aligned}$$

Proof. First, note that s has the following desugared form.

$$\begin{aligned} pre &= (w_0; in; \mathsf{tag} \leftarrow w + w) \\ post &= (out; \mathsf{tag} \leftarrow w_0 + \neg out) \\ s &= pre; p; post \\ &= (w_0; in; \mathsf{tag} \leftarrow w + w); p; (out; \mathsf{tag} \leftarrow w_0 + \neg out) \end{aligned}$$

Case 1: $s; t; \mathsf{dup}; q \equiv \mathsf{drop}$. We have:

	Assertion	**Reasoning**
	$s; t; \mathsf{dup}; q$	
$\equiv$	$pre; p; post; t; \mathsf{dup}; q$	Substitution for s.
$\equiv$	$pre; p; (out; \mathsf{tag} \leftarrow w_0 + \neg out); t; \mathsf{dup}; q$	Substitution for $post$.
$\equiv$	$pre; p; (out; \mathsf{tag} \leftarrow w_0; t; \mathsf{dup}; q + \neg out; t; \mathsf{dup}; q)$	KA-Seq-Dist-R.
$\equiv$	$pre; p; (\mathsf{tag} \leftarrow w_0; out; t; \mathsf{dup}; q + \neg out; t; \mathsf{dup}; q)$	H1, PA-Mod-Filter-Comm.
$\equiv$	$pre; p; (\mathsf{drop} + \neg out; t; \mathsf{dup}; q)$	H2.
$\equiv$	$pre; p; \neg out; t; \mathsf{dup}; q$	KA-Plus-Comm, KA-Plus-Zero.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + w); p; \neg out; t; \mathsf{dup}; q$	Substitution for pre.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w; w + \mathsf{id}; w); p; \neg out; t; \mathsf{dup}; q$	PA-Mod-Filter, KA-One-Seq.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + \mathsf{id}); w; p; \neg out; t; \mathsf{dup}; q$	KA-Seq-Dist-R.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + \mathsf{id}); p; \neg out; t; \mathsf{dup}; w; q$	H2, Lemma 17.
$\equiv$	$(w_0; in; \mathsf{tag} \leftarrow w + \mathsf{id}); p; \neg out; t; \mathsf{dup}; \mathsf{drop}$	H3.
$\equiv$	drop	KA-Seq-Zero.

Case 2: $q; t; \mathsf{dup}; s \equiv \mathsf{drop}$. We have:

	Assertion	**Reasoning**
	$q; t; \mathsf{dup}; s$	
$\equiv$	$q; t; \mathsf{dup}; pre; p; post$	Substitution for s.
$\equiv$	$q; t; \mathsf{dup}; (w_0; in; \mathsf{tag} \leftarrow w + w); p; post$	Substitution for pre.
$\equiv$	$(q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w + q; t; \mathsf{dup}; w); p; post$	KA-Seq-Dist-L.
$\equiv$	$(q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w + q; w; t; \mathsf{dup}); p; post$	PA-Obs-Filter-Comm, Lemma 17.
$\equiv$	$(q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w + \mathsf{drop}; t; \mathsf{dup}); p; post$	H3.
$\equiv$	$q; t; \mathsf{dup}; w_0; in; \mathsf{tag} \leftarrow w; p; post$	KA-Zero-Seq, KA-Plus-Zero.
$\equiv$	$q; t; \mathsf{dup}; in; w_0; \mathsf{tag} \leftarrow w; p; post$	BA-Seq-Comm.
$\equiv$	$\mathsf{drop}; w_0; \mathsf{tag} \leftarrow w; p; post$	H2.
$\equiv$	drop	KA-Zero-Seq. □

Theorem 1 (Slice/Program Composition)**.** *For all* tag*-free slice ingress and egress predicates* in, out*, identifiers* w*, policies* s, q*,* tag*-free policies* p*, and topologies* t*, such that*

- $s = (\!|\{in\}\ w : (p)\ \{out\}|\!)^{w_0}$,
- *H0:* $w \neq w_0$,
- *H1:* $out; t; \mathsf{dup}; q \equiv \mathsf{drop}$,

- *H2:* $q; t; \mathsf{dup}; in \equiv \mathsf{drop}$,
- *H3: q drops w-tagged traffic, then*

$$((s+q); t; \mathsf{dup})^* \equiv (s; t; \mathsf{dup})^* + (q; t; \mathsf{dup})^*$$

Proof. First, note that s has the following desugared form.

$$\begin{array}{lcl} pre & = & (w_0; in; \mathsf{vlan} \leftarrow w + w) \\ post & = & (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \\ s & = & pre; p; post \\ & = & (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \end{array}$$

	Assertion	**Reasoning**
	$((s+q); t; \mathsf{dup})^*$	
$\equiv$	$(s; t+q; t; \mathsf{dup})^*$	KA-Seq-Dist-R.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup}; (s; t; \mathsf{dup})^*)^*$	Lemma 16.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup}; (\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*))^*$	KA-Star-Unroll-L.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup} + q; t; \mathsf{dup}; s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*)^*$	KA-Seq-Dist-L, KA-Seq-One.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup} + \mathsf{drop}; t; \mathsf{dup}; (s; t; \mathsf{dup})^*)^*$	Lemma 19.
$\equiv$	$(s; t; \mathsf{dup})^*; (q; t; \mathsf{dup})^*$	KA-Zero-Seq, KA-Plus-Zero.
$\equiv$	$(\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*); (\mathsf{id} + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*)$	KA-Star-Unroll-L.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	
	$+ s; t; \mathsf{dup}; (s; t; \mathsf{dup})^*; q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	KA-Seq-Dist-L, KA-Seq-Dist-R. KA-One-Seq, KA-Seq-One.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	
	$+ s; t; \mathsf{dup}; (\mathsf{id} + (s; t; \mathsf{dup})^*; s; t; \mathsf{dup});$	
	$q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	KA-Star-Unroll-R.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	
	$+ s; t; \mathsf{dup}; q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	
	$+ (s; t; \mathsf{dup})^*; s; t; \mathsf{dup}; q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	KA-Seq-Dist-R, KA-One-Seq, KA-Star-Unroll-L.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	Lemma 19, KA-Zero-Seq, KA-Seq-Zero, KA-Plus-Zero.
$\equiv$	$\mathsf{id} + s; t; \mathsf{dup}; (s; t; \mathsf{dup})^* + \mathsf{id} + q; t; \mathsf{dup}; (q; t; \mathsf{dup})^*$	KA-Plus-Idem, KA-Plus-Comm.
$\equiv$	$(s; t; \mathsf{dup})^* + (q; t; \mathsf{dup})^*$	KA-Star-Unroll-L. $\square$

With this theorem, we can see that isolation is preserved across n-ary slice composition.

Lemma 20 (Parallel Composition Preserves w-dropping)**.** *If policies p, q drop w-tagged traffic, then $p + q$ drops w-tagged traffic.*

Proof. We have the following hypotheses:

- $\mathsf{tag} = w; p \equiv p; \mathsf{tag} = w \equiv \mathsf{drop}$
- $\mathsf{tag} = w; q \equiv q; \mathsf{tag} = w \equiv \mathsf{drop}$

And we must show the following goals:

- $\mathsf{tag} = w; (p + q) \equiv \mathsf{drop}$
- $(p + q); \mathsf{tag} = w \equiv \mathsf{drop}$

The result is follows from the hypotheses and KA-SEQ-DIST-L, KA-PLUS-ZERO, and the substitution of equals for equals.

□

Lemma 21 (Parallel Composition Preserves Disjoint Boundaries)**.** *If* $p_i; t; \mathsf{dup}; p_j \equiv \mathsf{drop}$ *for all pairs* (i, j) *for* $i \neq j, 1 <= i <= n, 1 <= j <= n$*, then* $p_1; t; \mathsf{dup}; (p_2 + p_3) \equiv \mathsf{drop}$ *and* $(p_2 + p_3); t; \mathsf{dup}; p_1 \equiv \mathsf{drop}$*.*

Proof. The result follows from the hypotheses, KA-SEQ-DIST-L, and KA-SEQ-DIST-R.

□

Lemma 22 (Slices Drop w-tagged Traffic)**.** *For all slices* $s = (\!|\{in\}\ w\ :\ (p)\ \{out\}|\!)^{w_0}$ *and tags* w' *such that* $w' \neq w \neq w_0$*, s drops* w'*-tagged traffic.*

Proof. First, note that s has the following desugared form.

$$\begin{array}{lcl} pre & = & (w_0; in; \mathsf{vlan} \leftarrow w + w) \\ post & = & (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \\ s & = & pre; p; post \\ & = & (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out) \end{array}$$

Goal 1: $\mathsf{tag} = w'; s \equiv \mathsf{drop}$*.* We have:

	Assertion	**Reasoning**
	$\mathsf{tag} = w'; s$	
$\equiv$	$w'; (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out)$	Substitution for s.
$\equiv$	$(w'; w_0; in; \mathsf{vlan} \leftarrow w + w'; w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out)$	KA-SEQ-DIST-L.
$\equiv$	$(\mathsf{drop}; in; \mathsf{vlan} \leftarrow w + \mathsf{drop}); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out)$	Hyp. and PA-CONTRA.
$\equiv$	drop	KA-SEQ-ZERO, KA-PLUS-ZERO.

Goal 2: $s; \mathsf{tag} = w' \equiv \mathsf{drop}$*.* We have:

Assertion	**Reasoning**
$s; \mathsf{tag} = w'$	
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0 + \neg out); w'$	Substitution for s
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{vlan} \leftarrow w_0; w' + \neg out; w')$	KA-SEQ-DIST-R
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; (out; \mathsf{drop} + \neg out; w')$	Hyp., PA-MOD-FILTER, PA-CONTRA
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); p; \neg out; w'$	KA-SEQ-ZERO, KA-PLUS-ZERO
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w + w); w'; p; \neg out$	As policy and predicates are tag-free
$\equiv (w_0; in; \mathsf{vlan} \leftarrow w; w' + w; w'); p; \neg out$	KA-SEQ-DIST-R
$\equiv (w_0; in; \mathsf{drop} + w; w'); p; \neg out$	Hyp., PA-MOD-FILTER, PA-CONTRA
$\equiv (w_0; in; \mathsf{drop} + \mathsf{drop}); p; \neg out$	Hyp., PA-CONTRA
$\equiv \mathsf{drop}$	KA-SEQ-ZERO, KA-PLUS-ZERO.

□

Proposition 1 (N-ary Slices are Isolated)**.** *For slices* $s_1, \ldots, s_n$ *compiled with tags* $w_1, \ldots, w_n$ *such that for all pairs* (i, j) *for* $i \neq j, 1 <= i <= n, 1 <= j <= n$*,*

- $w_i \neq w_j \neq w_0$, *and*
- $s_i; t; \mathsf{dup}; s_j \equiv s_j; t; \mathsf{dup}; s_i \equiv \mathsf{drop}$, *then*

$$((s_1 + \ldots + s_n); t; \mathsf{dup})^* \equiv (s_1; t; \mathsf{dup})^* + \ldots + (s_n; t; \mathsf{dup})^*$$

Proof. By induction on n. The base case $(n = 1)$ is immediate. We proceed with the inductive step.

1. Let $q = (s_1 + \ldots + s_k)$.
2. From Lemma 20 and the hypotheses, we have that q drops w_{k+1}-tagged traffic.
3. From Lemma 21 and the hypotheses, we have that $s_{k+1}; t; \mathsf{dup}; q \equiv q; t; \mathsf{dup}; s_{k+1} \equiv \mathsf{drop}$.

From Theorem 1 we have:

Assertion	**Reasoning**
$((s_{k+1} + q); t; \mathsf{dup})^* \equiv (s_{k+1}; t; \mathsf{dup})^* + (q; t; \mathsf{dup})^*$	Theorem 1.
$\equiv (s_{k+1}; t; \mathsf{dup})^* + ((s_1 + \ldots + s_k); t; \mathsf{dup})^*$	Substitution for q.
$\equiv (s_{k+1}; t; \mathsf{dup})^* + (s_1; t; \mathsf{dup})^* + \ldots + (s_k; t; \mathsf{dup})^*$	Induction hypothesis.
$\equiv (s_1; t; \mathsf{dup})^* + \ldots + (s_k; t; \mathsf{dup})^* + (s_{k+1}; t; \mathsf{dup})^*$	KA-Plus-Comm. □

D. Application: Compilation

This section presents the full proofs and supporting lemmas of the theorems in Section 7.

Definition 6 (NetKAT Sub-languages)**.** *We define the following sub-languages of* $Net\text{KAT}$*.*

- $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$ *if* p *does not contain* dup *or* $\mathsf{sw} \leftarrow n$*.*
- $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ *if* p *does not contain* dup *or* $\mathsf{sw} \leftarrow n$ *or* p^**.*
- $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ *if* p *does not contain* dup *or* $\mathsf{sw} \leftarrow n$ *or* p^* *or* $\mathsf{sw} = n$*.*
- $p \in Net\text{KAT}^{-(\mathsf{sw})}$ *if* p *has the form* $\sum_i \mathsf{sw} = i; p_i$ *and for* $i : 1..n$, $p_i \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$*.*

We call policies $p \in Net\text{KAT}^{-\mathsf{dup},\mathsf{sw}\leftarrow}$ *user policies*. This is the set of policies that $Net\text{KAT}$ programmers may write and that we an implement. We call policies $p \in Net\text{KAT}^{sw}$ *switch normal form policies*. Such policies have one component that is specialized to each switch in the network.

Compilation is then a series of steps, each refining or eliminating a fragment of the full NetKAT language.

1. **Star elimination.** Transform a *user policy* into an equivalent user policy without Kleene star.
2. **Switch specialization.** Transform a *star-free user policy* into switch normal form.
3. **OpenFlow normal form.** Transform a policy in *switch normal form* into OpenFlow normal form.

D.1 Star Elimination

Lemma 7 (Star Elimination)**.** *If* $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow)}$*, then there exists* $p' \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ *and* $p \equiv p'$*.*

Proof. The proof begins by showing that p' can be obtained from the normal form used in the completeness theorem. More specifically, let p'' be the policy obtained from p by the normalization construction of lemma 4. By construction, dup can only appear in the normal form of an expression already containing dup, therefore p'' does not contain dup. Moreover, $R(p'') \subseteq I$ and p'' does not contain dup, therefore $R(p'') \subseteq At; P$. Consequently, p'' does not contain Kleene star.

Let us now prove that any assignment of the form $\mathsf{sw} \leftarrow sw_i$ in p'' is preceded in the same term by the corresponding test $\mathsf{sw} = sw_i$. Because p does not contain any assignment of the form $\mathsf{sw} \leftarrow sw_i$, it commutes with any test of the form $\mathsf{sw} = sw_i$, and therefore p'' also commutes with any test of the form $\mathsf{sw} = sw_i$. p''

can be written as a sum of $\alpha;p$ for some atoms α and complete assignments p. Suppose for a contradiction that term, α contains a test $\mathsf{sw} = sw_i$, and p contains an assignment $\mathsf{sw} \leftarrow sw_j$, with $sw_i \neq sw_j$. Then

$$\alpha; (\mathsf{sw} = sw_i); p''; (\mathsf{sw} = sw_j) \geq \alpha; p \neq 0$$
$$\alpha; (\mathsf{sw} = sw_j); p''; (\mathsf{sw} = sw_i) = 0$$

but those two terms are also equal, which is a contradiction.

Therefore any assignment of the form $\mathsf{sw} \leftarrow sw_i$ in p'' is preceded, in the same term, by the corresponding test $\mathsf{sw} = sw_i$, and can be removed using axiom PA-FILTER-MOD to produce the desired p'. Tests and assignments to other fields than sw could appear in between, but we can use the commutativity axioms PA-MOD-MOD-COMM and PA-MOD-FILTER-COMM to move the assignment $\mathsf{sw} \leftarrow sw_i$ to just after the test $\mathsf{sw} = sw_i$. □

D.2 Switch Normal Form

First, we show that any policy in $Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ can be specialized to a given switch.

Lemma 8 (Switch Specialization). *If $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, then for all switches sw_i, there exists $p' \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ such that $\mathsf{sw} = sw_i; p \equiv \mathsf{sw} = sw_i; p'$.*

Proof. Let g be the unique homomorphism of NetKAT defined on primitive programs by:

$$g(\mathsf{sw} = sw) \triangleq \begin{cases} \mathsf{id} & \text{if } sw = sw_i \\ \mathsf{drop} & \text{otherwise} \end{cases}$$
$$g(f \leftarrow v) \triangleq f \leftarrow v$$
$$g(\mathsf{dup}) \triangleq \mathsf{dup}$$

For every primitive program element x of NetKAT$^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, we have both:

$$\mathsf{sw} = sw_i; x \equiv g(x); \mathsf{sw} = sw_i$$
$$g(x); \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i; g(x)$$

Hence, applying Lemma 4.4 in [2] twice shows:

$$\mathsf{sw} = sw_i; p \equiv g(p); \mathsf{sw} = sw_i$$
$$g(p); \mathsf{sw} = sw_i \equiv \mathsf{sw} = sw_i; g(p)$$

By the definition of g, any occurrence of $\mathsf{sw} = v$ in p is replaced by either id or drop in $g(p)$. Moreover, since $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$, it follows that $g(p)$ does not contain any occurrence of $\mathsf{sw} = v$ and since $p' = g(p) \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*,\mathsf{sw})}$ we also have

$$\mathsf{sw} = sw_i; p \equiv \mathsf{sw} = sw_i; p'$$ □

As there are finitely many switches in a network (codified in Axiom PA-MATCH-ALL), we can use Lemma 8 to show that any star-free user policy can be put into switch normal form.

Lemma 23 (Switch-Normal-Form). *If $p \in Net\text{KAT}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,*)}$ then there exists $p' \in Net\text{KAT}^{\mathsf{sw}}$ such that $p \equiv p'$.*

ONF Action Sequence	a	$::=$	$\mathsf{id} \mid f \leftarrow n; a$
ONF Action Sum	as	$::=$	$\mathsf{drop} \mid a + as$
ONF Predicate	b	$::=$	$\mathsf{id} \mid f = n; b$
ONF Local	ℓ	$::=$	$as \mid \mathsf{if}\ b\ \mathsf{then}\ as\ \mathsf{else}\ \ell$
ONF	p	$::=$	$\mathsf{drop} \mid (\mathsf{sw} = sw; \ell) + p$

Figure 7. OpenFlow Normal Form.

Proof.

	Assertion	**Reasoning**
	p	
$\equiv$	$\mathsf{id}; p$	KA-One-Seq.
$\equiv$	$(\sum_i \mathsf{sw} = i); p$	PA-Match-All.
$\equiv$	$\sum_i (\mathsf{sw} = i; p)$	KA-Seq-Dist-R.
$\equiv$	$\sum_i (\mathsf{sw} = i; p_i)$ where $p_i \in Net\text{KAT}^{-\mathsf{dup},\mathsf{sw}\leftarrow,^*,\mathsf{sw}}$	Lemma 8.

□

D.3 OpenFlow Normal Form

Finally, we show that any policy in switch normal form is equivalent to a policy in OpenFlow normal form.

Notice that the definition of OpenFlow normal form (ONF) in Figure 7 is nearly identical to the definition of switch normal form. The key difference lies in the form of the switch-local policy fragments. In ONF, these policy fragments must be in ONF local form. Hence, we show how to transform a switch-local policy fragment into ONF local form.

The proof proceeds by induction on the structure of the NetKAT policy. We will see that for each combinator, two policy fragments in ONF local form can be joined to produce a policy in ONF local form. After defining several lemmas useful throughout the remainder of the section, we show that parallel composition, sequential composition, and predicate compilation are all sound. We conclude by showing stating and proving the full compiler soundness theorem.

Coq-based Lemmas. Damien Pous has released a sound and complete decision procedure for Kleene algebras with tests, implemented in the Coq theorem prover [4]. For several lemmas that rely on lengthy program transformations using only the standard KAT axioms, we appeal to this decision procedure. The Coq code for each lemma is reproduced below.

```
Lemma atom_plus_onf_is_onf `{L: laws} n (b : tst n) (p q r : X n n):
  p + [b];q + [!b];r == [b];(p + q) + [!b];(p + r).
Proof. kat. Qed.

Lemma union_is_ite_nf `{L: laws} n (b1 b2 : tst n) (p1 p2 q : X n n):
  [b1];([b2];p1 + [!b2];p2) + [!b1];q ==
  [b1];[b2];p1 + [b1];[!b2];p2 + [!b1];[b2];q + [!b1];[!b2];q.
Proof. kat. Qed.

Lemma demorgans `{L: laws} n (b1 b2 : tst n) (p1 p2 q : X n n):
  [!(b1 \cap b2)];p1 == [!b1];p1 + [!b2];p1.
```

[4] http://perso.ens-lyon.fr/damien.pous/ra

```
Proof. kat. Qed.

Lemma ite_nf_is_onf `{L: laws} n (b1 b2 : tst n) (p1 p2 q : X n n):
  [b1];[b2];p1 + [b1];[!b2];p2 + [!b1];[b2];q + [!b1];[!b2];q ==
  [b1 \cap b2];p1 + [!(b1 \cap b2)];([b1];p2 + [!b1];q).
Proof. kat. Qed.

Lemma nested_onf_is_onf `{L: laws} n (b1 b2 : tst n) (p1 p2 q : X n n):
  [b1];([b2];p1 + [!b2];p2) + [!b1];q ==
  [b1 \cap b2];p1 + [!(b1 \cap b2)];([b1];p2 + [!b1];q).
Proof. kat. Qed.

Lemma union_onf_is_onf `{L: laws} n (b1 b2 : tst n) (a1 a2 p' q' : X n n):
  [b1];a1 + [!b1];p' + [b2];a2 + [!b2];q' ==
  [b1 \cap b2];(a1 + a2) + [!(b1 \cap b2)];([b1];(a1 + q')
  + [!b1];(p' + [b2];a2 + [!b2];q')).
Proof. kat. Qed.

Lemma joined_policies `{L: laws} n (s1 s2 t : X n n):
  ((s1 + s2); t)^* == ((s1; t)^* + (s2; t)^*)^*.
Proof. kat. Qed.
```

Helper lemmas. Several subgoals appear repeatedly in this last part of the proof of compiler correctness. The first simply observes that every test is either true or false (BA-Excluded Middle), and so every program may be broken into a sum, wherein either the test or its negation is sequenced with the program.

Lemma 24 (Predicate Expansion)**.** *For all predicates a, b and policies p, $b; p \equiv a; b; p + \neg a; b; p$.*

Proof.

	Assertion	**Reasoning**
1	$b; p$	
2	$\equiv \mathsf{id}; b; p$	KA-One-Seq
3	$\equiv (a + \neg a); b; p$	BA-Excluded-Middle
Goal	$\equiv a; b; p + \neg a; b; p$	KA-Seq-Dist-R. □

Consider ONF as a series of nested if statements for a moment. During the proofs, we will often conclude that the body of *both* branches of the topmost if statement are themselves nested if statements in ONF. In order to show that the expression as a whole is indeed in ONF, we must show that the structure is equivalent to a single series of nested if statements.

Lemma 25 (Nested ONF is ONF)**.** *For all ONF predicates a and policies p, q in ONF, there exists a policy r in ONF such that $r \equiv a; p + \neg a; q$.*

Proof. By induction on the structure of p.

Case 1. We have $p = as_1$, and the result is immediate.

Case 2. We have $p = b; as + \neg b; q$.

Assertion	Reasoning
$a; (b; as + \neg b; q) + \neg a; p'$ $\equiv (a; b); as + \neg(a; b); (a; q + \neg a; p')$	Coq Lemma nested_onf_is_onf.

Applying the induction hypothesis to $(a; q + \neg a; p')$ yields r' in ONF. Hence, our goal is satisfied with $(a; b); as + \neg(a; b); r'$. □

Finally, there are several commutativity conditions that are essential to reasoning about NetKAT program transformations.

Lemma 26 (Negative Field Commutativity)**.** *For all fields f_1, f_2 and values n_1, n_2 such that $f_1 \neq f_2$, $f_1 \leftarrow n_1; \neg f_2 = n_2 \equiv \neg f_2 = n_2; f_1 \leftarrow n_1$.*

Proof. By PA-Mod-Mod-Comm, we have $f_1 \leftarrow n_1; f_2 = n_2 \equiv f_2 = n_2; f_1 \leftarrow n_1$. The desired result then follows from [14, Lemma 2.3.1]. □

Lemma 27 (Action Atom Seq Filter Commutativity)**.** *For all ONF action sequences a_1, fields f, and values n, one of the following holds:*

- $a_1; f = n \equiv f = n; a_1$ *and* $a_1; \neg f = n \equiv \neg f = n; a_1$,
- $a_1; f = n \equiv a_1$ *and* $a_1; \neg f = n \equiv \mathsf{drop}$, *or*
- $a_1; f = n \equiv \mathsf{drop}$ *and* $a_1; \neg f = n \equiv a_1$.

Proof. By induction on the structure of a_1. We have two cases. The base case, $a_1 \equiv \mathsf{id}$, is trivial. We continue with the induction case, $a_1 \equiv f_1 \leftarrow n_1; a_1'$. Applying the induction hypothesis to a_1', f, and n yields three new cases:

Case 1. We have $f_1 \leftarrow n_1; a_1'; f = n \equiv f_1 \leftarrow n_1; f = n; a_1'$ and $f_1 \leftarrow n_1; a_1'; \neg f = n \equiv f_1 \leftarrow n_1; \neg f = n; a_1'$. There are three cases:

Case 1a. If $f_1 \neq f$, then $f_1 \leftarrow n_1; f = n; a_1' \equiv f = n; f_1 \leftarrow n_1; a_1'$ by PA-Mod-Filter-Comm. By Lemma 26, we also have $f_1 \leftarrow n_1; \neg f_2 = n_2 \equiv \neg f_2 = n_2; f_1 \leftarrow n_1$.

Case 1b. If $f_1 \equiv f$ and $n \equiv n_1$, then $f_1 \leftarrow n_1; f = n; a_1' \equiv f_1 \leftarrow n_1; a_1'$ by PA-Mod-Filter. We also have:

	Assertion	Reasoning
1	$f_1 \leftarrow n_1; \neg f = n; a_1'$	
2	$\equiv f_1 \leftarrow n_1; f = n; \neg f = n; a_1'$	PA-Mod-Filter.
3	$\equiv f_1 \leftarrow n_1; \mathsf{drop}; a_1'$	BA-Contra.
Goal	$\equiv \mathsf{drop}$	KA-Zero-Seq and KA-Seq-Zero.

Case 1c. If $f_1 \equiv f$ and $n \neq n_1$, then $f_1 \leftarrow n_1; f = n; a_1' \equiv \mathsf{drop}$ by PA-Mod-Filter and PA-Contra. We also have:

	Assertion	Reasoning
1	$f \leftarrow n_1; \neg f = n; a_1'$	
2	$\equiv \mathsf{drop} + f \leftarrow n_1; \neg f = n; a_1'$	KA-Plus-Zero and KA-Plus-Comm.
3	$\equiv f \leftarrow n_1; \mathsf{drop}; a_1' + f \leftarrow n_1; \neg f = n; a_1'$	KA-Seq-Zero and KA-Zero-Seq.
4	$\equiv f \leftarrow n_1; f = n_1; f = n; a_1' + f \leftarrow n_1; \neg f = n; a_1'$	PA-Contra.
5	$\equiv f \leftarrow n_1; f = n; a_1' + f \leftarrow n_1; \neg f = n; a_1'$	PA-Mod-Filter-Comm.
6	$\equiv f \leftarrow n_1; (f = n + \neg f = n); a_1'$	KA-Seq-Dist-L, KA-One-Seq, KA-Seq-Dist-R.
Goal	$\equiv f \leftarrow n_1; a_1'$	BA-Excluded-Middle, KA-Seq-One.

Case 2. We know that

- $a_1'; f = n \equiv a_1'$, and
- $a_1'; \neg f = n \equiv \mathsf{drop}$.

Hence, by substitution and KA-Seq-Zero,

- $f_1 \leftarrow n_1; a_1'; f = n \equiv f_1 \leftarrow n_1; a_1'$, and
- $f_1 \leftarrow n_1; a_1'; \neg f = n \equiv \mathsf{drop}$.

Case 3. We know that

- $a_1'; f = n \equiv \mathsf{drop}$, and
- $a_1'; \neg f = n \equiv a_1'$.

Hence, by substitution and KA-Seq-Zero,

- $f_1 \leftarrow n_1; a_1'; f = n \equiv \mathsf{drop}$, and
- $f_1 \leftarrow n_1; a_1'; \neg f = n \equiv f_1 \leftarrow n_1; a_1'$. □

Lemma 28 (Action Atom Seq Predicate Commutativity)**.** *For all ONF action sequences a_1 and ONF predicates b_1, one of three cases holds:*

- *there exists an ONF predicate b_2 such that $a_1; b_1 \equiv b_2; a_1$ and $a_1; \neg b_1 \equiv \neg b_2; a_1$,*
- *$a_1; b_1 \equiv a_1$ and $a_1; \neg b_1 \equiv \mathsf{drop}$, or*
- *$a_1; b_1 \equiv \mathsf{drop}$ and $a_1; \neg b_1 \equiv a_1$.*

Proof. By induction on the structure of b_1. The base case, $b_1 \equiv \mathsf{id}$, is trivial, leaving the inductive case $b_1 \equiv f = n; b_1'$. We have $a_1; f = n; b_1'$. Applying Lemma 27 to $a_1; f = n$ yields three cases.

Case 1. We have that $a_1; f = n \equiv f = n; a_1$. Hence, $a_1; f = n; b_1' \equiv f = n; a_1; b_1'$ and the goal follows from an application of the induction hypothesis.

The same reasoning resolves the negative case, where $a_1; \neg f = n \equiv \neg f = n; a_1$.

Case 2. We have that $a_1; f = n \equiv a_1$. Hence, $a_1; f = n; b_1' \equiv a_1; b_1'$ and the goal follows from an application of the induction hypothesis.

The negative case, where $a_1; \neg f = n \equiv \mathsf{drop}$, follows from rewriting and KA-Seq-Zero.

Case 3. We have that $a_1; f = n \equiv \mathsf{drop}$. Hence, $a_1; f = n; b_1' \equiv \mathsf{drop}$ by KA-Seq-Zero.

The negative case, where $a_1; \neg f = n \equiv a_1$, proceeds as follows. By rewriting, we have $a_1; \neg f = n; b_1' \equiv a_1; b_1'$ and the goal follows from an application of the induction hypothesis. □

Parallel Composition Preserves ONF. In this section, we show that the parallel composition of two policies in ONF is itself equivalent to a policy in ONF. The proof proceeds by mutual induction on the structure of both subpolicies, and the section presents supporting lemmas leading up to the final result.

Lemma 29 (Action Plus Action is ONF)**.** *For all ONF action sums as_1 and as_2, there exists an ONF action sum as_3 such that $as_1 + as_2 \equiv as_3$.*

Proof. Both as_1 and as_2 can take one of two forms, drop or $a + as'$, leading to four cases. The goal follows immediately from KA-Plus-Zero in all but the final case.

Case 4. We have $as_1 \equiv a_1 + as_1'$ and $as_2 \equiv a_2 + as_2'$. $as_3 \equiv a_1 + a_2 + as_1' + as_2'$, which can be put into normal form via list concatenation. □

Lemma 30 (Action Plus ONF is ONF)**.** *For all ONF action sums as_1 and policies $p \equiv b; as_2 + \neg b; q$ in ONF, there exists a policy r in ONF such that $r \equiv as_1 + p$.*

Proof. By induction on p.

Case 1. We have $p \equiv as_2$. The goal follows immediately from Lemma 29.

Case 2. We have $p \equiv b; as_2 + \neg b; q$.

	Assertion	**Reasoning**
1	$as_1 + b; as_2 + \neg b; q$	
2	$\equiv b; as_1 + \neg b; as_1 + b; as_2 + \neg b; q$	Lemma 24.
3	$\equiv (b; as_1) + (b; as_2) + (\neg b; as_1) + (\neg b; q)$	KA-Plus-Comm.
4	$\equiv b; (as_1 + as_2) + \neg b; (as_1 + q)$	KA-Seq-Dist-L.

The goal then follows from an application of the induction hypothesis to $(as_1 + q)$. □

Lemma 31 (Parallel Composition Preserves ONF)**.** *For all policies p, q in OpenFlow normal form, there exists r such that $p + q \equiv r$ and r is in OpenFlow normal form.*

Proof. By induction on p and split in to cases based on the structure of q.

Case 1. We have $p \equiv as_1$ and $q \equiv as_2$, and $r \equiv as_1 + as_2$. The goal follows from Lemma 29.

Case 2. We have $p \equiv as_1$ and $q \equiv b; as_2 + \neg b; q'$, and $r \equiv as_1 + b; as_2 + \neg b; q'$. The goal follows from Lemma 30.

Case 3. We have $p \equiv b; as_1 + \neg b; p'$ and $q \equiv as_2$, and $r \equiv b; as_1 + \neg b; p' + as_2$. The goal follows from Lemma 30 and KA-Plus-Comm.

Case 4. We have $p \equiv b_1; as_1 + \neg b_1; p'$ and $q \equiv b_2; as_2 + \neg b_2; q'$, and $r \equiv b_1; as_1 + \neg b_1; p' + b_2; as_2 + \neg b_2; q'$.

	Assertion	Reasoning
1	$b_1; as_1 + \neg b_1; p' + b_2; as_2 + \neg b_2; q'$	
2	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; (as_1 + q') + \neg b_1; (p' + b_2; as_2 + \neg b_2; q'))$	Coq Lemma union_onf_is_onf.
3	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; r_1 + \neg b_1; (p' + b_2; as_2 + \neg b_2; q'))$	(for some r_1 in ONF) by Lemma 30.
4	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; r_1 + \neg b_1; (p' + q))$	Substitute q.
5	$(b_1; b_2); (as_1 + as_2)$ $+ \neg(b_1; b_2); (b_1; r_1 + \neg b_1; r_2)$	IH.
6	$(b_1; b_2); (as_1 + as_2) + \neg(b_1; b_2); r_3$	Lemma 25. □

Sequential Composition Preserves ONF. In this section, we show that the sequential composition of two policies in ONF is itself equivalent to a policy in ONF. The proof proceeds by mutual induction on the structure of both subpolicies and the structure of ONF action sums in the left-hand policy. The section presents supporting lemmas leading up to the final result.

Lemma 32 (Conjunct Seq Conjunct is ONF)**.** *For all ONF action sequences a_1, a_2, there exists an ONF action sequence $a_3 \equiv a_1; a_2$.*

Proof. By induction on the structure of a_1.

Case 1. We have $a_1 \equiv \mathsf{id}$ and $a_3 \equiv \mathsf{id}; a_2 \equiv a_2$ by KA-One-Seq.

Case 2. We have $a_1 \equiv f \leftarrow n; a_1'$ and $a_3 \equiv (f \leftarrow n; a_1'); a_2$.

	Assertion	Reasoning
1	$a_3 \equiv (f \leftarrow n; a_1'); a_2$	
2	$\equiv f \leftarrow n; a_1'; a_2$	KA-Seq-Assoc
Goal	$\equiv f \leftarrow n; a_3'$	IH □

Lemma 33 (Conjunct Seq Action is ONF)**.** *For all ONF action sequences a and ONF action sums as_1, there exists an ONF action sum $as_2 \equiv a; as_1$.*

Proof. We begin with induction on as_1. The first case is trivial: $as_2 \equiv a; \mathsf{drop} \equiv \mathsf{drop}$ by KA-Seq-Zero. In the second case, $as_1 \equiv a_1 + as_1'$.

	Assertion	Reasoning
1	$as_2 \equiv a; (a_1 + as_1')$	
2	$\equiv a; a_1 + a; as_1'$	KA-Seq-Dist-L.
3	$\equiv a_2 + a; as_1'$	Lemma 32.
Goal	$\equiv a_2 + as_2'$	IH □

Lemma 34 (Action Seq Action is ONF)**.** *For all ONF action sums as_1 and as_2, there exists an ONF action sum as_3 such that $as_1; as_2 \equiv as_3$.*

Proof. By induction on as_1. In the first case, $as_1 \equiv \mathsf{drop}$, and the goal follows from KA-Zero-Seq. In the latter case, $as_1 \equiv a_1 + as_1'$.

	Assertion	Reasoning
1	$as_3 \equiv (a_1 + as_1'); as_2$	
2	$\equiv a_1; as_2 + as_1'; as_2$	KA-Seq-Dist-R.
3	$\equiv as_{31} + as_1'; as_2$	Lemma 33.
4	$\equiv as_{31} + as_{32}$	IH.
Goal	$\equiv as_3'$	Lemma 29. $\square$

Lemma 35 (Action Atom Seq ONF is ONF)**.** *For all ONF action sequences a_1 and policies p in ONF, there exists a policy q in ONF such that $q \equiv a_1; p$.*

Proof. By induction on p. The base case, $p \equiv as$, is discharged by Lemma 33. That leaves the inductive case, $p \equiv b; as + \neg b; p'$ and $q \equiv a_1; (b; as + \neg b; p')$.

	Assertion	Reasoning
1	$q \equiv a_1; (b; as + \neg b; p')$	
2	$\equiv a_1; b; as + a_1; \neg b; p'$	KA-Seq-Dist-L.

Applying Lemma 28 to a_1 and b yields three cases:

Case 1. We have that $a_1; b \equiv b; a_1$ and $a_1; \neg b \equiv \neg b; a_1$. After rewriting line 2 from the proof table above, we have:

	Assertion	Reasoning
1	$b; a_1; as + \neg b; a_1; p'$	
2	$b; a_1; as + \neg b; q'$	(for some q' in ONF) by IH
2	$b; as_2 + \neg b; q'$	(for some as_2) by Lemma 33

Case 2. We have that $a_1; b \equiv a_1$ and $a_1; \neg b \equiv \mathsf{drop}$. After rewriting, we have:

	Assertion	Reasoning
1	$a_1; as + \mathsf{drop}; p'$	
2	$a_1; as$	KA-Zero-Seq and KA-Plus-Zero.
3	as_2	(for some as_2) by Lemma 33

Case 3. We have that $a_1; b \equiv \mathsf{drop}$ and $a_1; \neg b \equiv a_1$. After rewriting, we have:

	Assertion	Reasoning
1	$\mathsf{drop}; as + a_1; p'$	
2	$a_1; p'$	KA-Zero-Seq and KA-Plus-Zero.
3	q'	by IH $\square$

Lemma 36 (Action Seq ONF is ONF)**.** *For all ONF action sums as and policies p in ONF, there exists a policy q in ONF such that $q \equiv as; p$.*

Proof. By induction on the structure of as. The base case, $as = \mathsf{drop}$, is trivial. This leaves the inductive case, where $as = a + as'$.

	Assertion	Reasoning
1	$(a + as'); p$	
2	$\equiv a; p + as'; p$	KA-Seq-Dist-R.
3	$\equiv r + as'; p$	Lemma 35.
4	$\equiv r + q'$	IH.

The goal then follows from applying Lemma 31. □

Lemma 37 (Sequential Composition Preserves ONF)**.** *For all policies p, q in OpenFlow normal form, there exists r such that $p; q \equiv r$ and r is in OpenFlow normal form.*

Proof. By induction on p.

Case 1. We have $p = as_1$ and $r \equiv as_1; q$. The result follows from Lemma 36.

Case 2. We have $p = b; as_1 + \neg b; p'$ and $r \equiv (b; as_1 + \neg b; p'); q$.

	Assertion	Reasoning
1	$(b; as_1 + \neg b; p'); q$	
2	$\equiv b; as_1; q + \neg b; p'; q$	KA-Seq-Dist-R.
3	$\equiv b; r_1 + \neg b; p'; q$	(for some r_1 in ONF) by Lemma 36.
4	$\equiv b; r_1 + \neg b; r_2$	IH.

The result then follows from applying Lemma 25. □

Predicate Compilation is Sound. We can also see that NetKAT predicates have equivalents in ONF.

Lemma 38 (Filter is ONF)**.** *For all policies $p = a$, there exists a policy q in ONF such that $p \equiv q$.*

Proof. By induction on the structure of a. The cases of id and drop are immediate.

Case: $f = n$.

	Assertion	Reasoning
1	$f = n$	
2	$\equiv f = n + \mathsf{drop}$	KA-Plus-Zero.
Goal	$\equiv f = n; \mathsf{id} + +\neg f = n; \mathsf{drop}$	KA-Seq-Zero, KA-Seq-One.

Case: $\neg b$.

	Assertion	Reasoning
1	$\neg b$	
2	$\equiv \neg b; \mathsf{id}$	KA-Seq-One.
3	$\equiv \mathsf{drop} + \neg b; \mathsf{id}$	KA-Plus-Zero, KA-Plus-Comm.
Goal	$\equiv b; \mathsf{drop} + \neg b; \mathsf{id}$	KA-Seq-Zero.

Case: $b + c$. From the induction hypothesis, we have $q \equiv q' + q''$. The result then follows from an application of Lemma 31.

Case: b; c. From the induction hypothesis, we have $q \equiv q'; q''$. The result then follows from an application of Lemma 37. □

Compiler Soundness. Finally, we show that any switch fragment of a policy in switch normal form has an equivalent policy in OpenFlow normal form. Hence, any user policy has an equivalent policy in OpenFlow normal form.

Lemma 9 (Switch-local Compilation)**.**
If $p \in Net_{\mathrm{KAT}}{}^{-(\mathsf{dup},\mathsf{sw}\leftarrow,^*,\mathsf{sw})}$ then there exists a policy p' such that $p \equiv p'$ and $p' \in$ ONF.

Proof. By induction on the structure of p.

Case 1. We have $p = a$. The goal follows from applying Lemma 38.

Case 2. We have $p = f \leftarrow n$ (and $f \neq \mathsf{sw}$). Applying KA-Seq-One yields $p \equiv f \leftarrow n; \mathsf{id}$, which satisfies our goal.

Case 3. We have $p = q + r$. Applying the induction hypothesis to q and r yields $p \equiv q' + r'$, after substitution. The goal then follows from Lemma 31.

Case 4. We have $p = q; r$. Applying the induction hypothesis to q and r yields $p \equiv q'; r'$, after substitution. The goal then follows from Lemma 37.

Case 5. We have $p = p'^*$. This case is inconsistent with the hypothesis ($p'^* \notin Net_{\mathrm{KAT}}{}^{-\mathsf{dup},\mathsf{sw}\leftarrow,^*,\mathsf{sw}}$). □

Theorem 38 (User policies can be compiled to G-ONF)**.** *For all user policies $p \in Net_{\mathrm{KAT}}{}^{-\mathsf{dup},\mathsf{sw}\leftarrow}$ there exists a policy $p' \in$ G-ONF such that $p \equiv p'$.*

Proof. By Lemmas Star-Elimination, Switch-Normal-Form and Switch-Local-Compilation. □

D.4 Optimizations

Lemma 39 (If Compress)**.** $\mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else}\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell \equiv \mathsf{if}\ b_1 + b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell$

Proof. By desugaring the *if* statements and then applying boolean algebra and distributivity axioms. □

Lemma 10 (Fall-through Elimination)**.** *If $b_1 \leq b_2$ then* $\mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else}\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell \equiv \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell$.

Proof.

$$
\begin{aligned}
&\mathsf{if}\ b_1\ \mathsf{then}\ as\ \mathsf{else}\ \mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell \\
\equiv\ &\mathsf{if}\ b_1 + b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell && \text{by If Compress} \\
\equiv\ &\mathsf{if}\ b_2\ \mathsf{then}\ as\ \mathsf{else}\ \ell && \text{by } b_1 \leq b_2 \quad \square
\end{aligned}
$$