Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis

Haoxian Chen
Tsinghua University
hc865@cornell.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Jake Silverman
Cornell University
jrs546@cornell.edu

Michael Whittaker
Cornell University
mjw297@cornell.edu

Brandon Zhang
Cornell University
bwz6@cornell.edu

Rene Zhang
Cornell University
rz99@cornell.edu

ABSTRACT

Network measurement is an essential component of many SDN applications, but most existing controller platforms force programmers to implement measurement tasks by installing fine-grained forwarding rules on switches—an approach that significantly increases configuration and management complexity. This paper proposes a radically different approach: rather than implementing measurement tasks directly on network switches, we argue for pushing measurement to the edge and utilizing the abundant resources available on end hosts. At a technical level, our approach is based on two key ideas: (i) we express measurement tasks using programs in a high-level, declarative query language, and (ii) we use program analysis to calculate predicates that can be used to answer queries at the edge of the network. We present an implementation of our approach on top of the NetKAT framework, we develop case studies illustrating the benefits of our approach, and we conduct experiments that quantify performance on realistic benchmarks.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

Keywords

Network measurement, domain-specific languages, program analysis, Kleene algebra with tests, NetKAT.

1. INTRODUCTION

Network measurement is an essential component of many SDN applications. Programmers must measure the flow of traffic across the network for a variety of reasons ranging from discovering hosts to billing customers to detecting congestion to debugging errors, among many others. Unfortunately, existing SDN platforms offer only rudimentary support for implementing network measurement tasks—typically programmers must install fine-grained forwarding rules on switches and poll the byte and packet counters associated with those rules in a tight loop.

At first glance, implementing measurement tasks using counters might seem like an attractive approach, since it allows programmers to use the same constructs to specify both how to forward and measure traffic. However, in practice, this approach has a number of disadvantages:

- It complicates network configurations, which is likely to increase the rate of software bugs and other errors. On a single switch, constructing rules to separate out the traffic being measured is already quite complicated; measuring traffic across multiple switches can require adding state to keep track of the paths taken by packets as they traverse the network.

- It increases the amount of memory needed to store configurations on switches. If measurement and forwarding can be implemented using independent tables, then the increase is merely linear, but if they must be encoded into the same table, there can be a quadratic blowup. Either way, this is a key limitation on current switches, which offer relatively small numbers of tables and forwarding rules.

- It increases the load on switches since the local control plane must retrieve the values stored in hardware counters, aggregate them together, and send the results back to the controller. In extreme cases, this can prevent the switch from being able to quickly process other control messages, such as commands for inserting new rules into a TCAM.

- These problems are exacerbated in situations where the controller must make frequent updates to the configuration of the network in response to changes in the topology, traffic patterns, security policies, etc. There are also questions about the consistency of results computed during configuration updates [22].

Although a number of recent SDN-based systems have proposed new hardware and software abstractions designed to make it easy to implement measurement tasks, many of them suffer from these fundamental limitations [8, 12, 20, 27, 24].

SOSR '16, March 14-15, 2016, Santa Clara, CA, USA
© 2016 ACM. ISBN 978-1-4503-4211-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2890955.2890971

Approach. This paper explores a different approach to implementing measurement tasks. Rather than attempting to implement measurement by explicitly programming network devices, we push measurement to the edge and take advantage of the abundant computational resources available on end hosts. In our system, called Felix, hosts execute a custom network stack that provides hooks for inserting local predicates that are evaluated on all incoming and outgoing packets. Multiple queries can be composed together, since hosts typically have plentiful memories, unlike most hardware switches. Handling updates is also straightforward since Felix's architecture cleanly decouples forwarding and measurement. To enable efficient computation of network-wide statistics, Felix provides a tree-structured overlay that aggregates information collected at each host into a final value at the controller.

The simplest way to use a system like Felix is to install local predicates at each host that collect the required information and configure the overlay to aggregate this information into the desired result. For example, to compute the total amount of traffic going across a given network link, the programmer would install predicates that count the amount of outgoing traffic that eventually traverses that link in the current forwarding configuration (for simplicity, assume the configuration implements loop-free paths) and then configure the overlay to aggregate the counters collected at each host into a single numeric value. This idea has been explored in the HONE system [24], which offers a unified, declarative interface for querying state on switches and end hosts. However, although this approach is workable in simple settings, computing the local predicates can be difficult to do by hand, especially in larger networks with complex configurations.

Queries and Analysis. To address this challenge, Felix provides a high-level language for specifying network-wide queries and automated tools for compiling queries into predicates that can be installed on each host. Syntactically, the query language is based on NetKAT [1], which is in turn based on regular expressions—a natural and well-studied formalism for describing paths through a graph. Using this language, a programmer can directly specify advanced monitoring queries, such as "the number of packets processed by the firewall," "the number of packets that traverse the path between Ithaca and New York City," and "the number of packets received from visitor hosts that eventually reach an internal server." Given such a query, Felix uses program analysis to calculate a collection of predicates that describe the set of packets that will satisfy the query when injected into the network. These predicates can then be installed on end hosts to collect the information needed to answer the overall query. The analysis of queries is based on a novel technique for compiling queries into the NetKAT language, as well as a representation of NetKAT programs based on finite automata and binary decision diagrams that was developed by some of the authors in previous work [10, 23].

Limitations. Implementing measurement at the edge does have certain limitations. Most important, the result computed for a given query is based on a model of the network under idealized conditions, which may or may not reflect reality. For example, if the network is congested, packets counted by hosts at ingress may actually be dropped in the core of the network. Likewise, if the switches exhibit hardware or software bugs, the paths specified by the configuration may not correspond to the actual paths used in the network. This limitation can be mitigated, to some extent, by using packet probes to detect congestion and bugs—at the very least, such probes could be used to check whether the results of the query are likely to be correct under current conditions. However, our current prototype does not provide this functionality. Despite this limitation, we believe that the division of labor embodied in Felix strikes a good balance between simplicity, flexibility, and performance, while making reasonable tradeoffs about the precision of query answers under extreme operating conditions.

Experience. To evaluate our design for Felix, we have built a prototype implementation in C, Python, and OCaml. The system comprises several components: an end-host monitor (based on `netfilter`) that applies predicates to every incoming and outgoing packet and maintains statistics in tables; an end-host agent that aggregates the information collected by the kernel module into a network-wide result using a tree-structured overlay; a declarative query language and program analysis tool based on NetKAT; and a simple SDN controller that orchestrates the behavior of all of these components. Using our prototype, we have built several applications that illustrate the use of our system for implementing rich measurement tasks. We have also conducted quantitative experiments to evaluate the performance of our analysis on realistic topologies and configurations.

Contributions. The main contributions of this paper are as follows:

- We make the case for implementing network measurement at the edge and present the design of a practical system based on this idea.
- We present a high-level language for specifying monitoring queries based on regular expressions and show how to analyze queries in this language using automata.
- We discuss a prototype implementation based on the NetKAT framework.
- We conduct case studies and experiments illustrating the use of our system on a variety of example applications and evaluate its performance on real-world topologies and configurations.

In outline, the rest of this paper is structured as follows. Section 2 motivates the design of Felix in further detail, using a simple running example. Sections 3 and 4 present NetKAT and our query language respectively. Section 5 describes our approach to analyzing queries. Section 6 presents our implementation, and Section 7 evaluates it. We discuss related work in Section 8 and conclude in Section 9.

2. OVERVIEW

As a simple example to illustrate the main ideas behind our approach, consider the diamond topology shown in Figure 1. It consists of four switches, with a single host connected to each switch. There are links around the perimeter and a single "shortcut" link going from north to south across the interior of the diamond.

Initially the network is configured to forward traffic using shortest paths—e.g., traffic from h_1 to h_3 traverses a direct

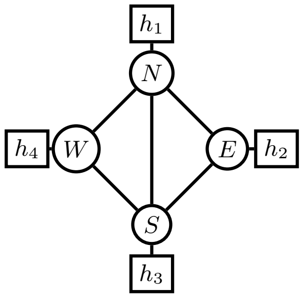


Figure 1: Diamond topology.

path across the shortcut link while traffic from h_2 to h_4 traverses a two-hop path via N or S. The configuration for switch N would look like this (for simplicity, we write names of switches and hosts rather than port numbers):

N:

Match	**Actions**
$Dst = h_1$	*Forward* h_1
$Dst = h_2$	*Forward* E
$Dst = h_3$	*Forward* S
$Dst = h_4$	*Forward* W

Now suppose the programmer decides to add measurement functionality to the application. Depending on the details of the application, there are a variety of measurement queries that might be needed such as:

- How much total traffic is flowing across the network?
- How congested is each link in the network?
- How much HTTP traffic is traversing the shortcut link?
- How much traffic traverses a two-hop path?

To implement these queries on current SDN platforms, the programmer would have to generate additional rules for the traffic being measured. For example, to measure the amount of HTTP traffic flowing across the shortcut link, they might modify the configuration of N to the following:

N:

Match	**Actions**
$Dst = h_1$	*Forward* h_1
$Dst = h_2$	*Forward* E
$Dst = h_3$, $Type = HTTP$	*Forward* S
$Dst = h_3$	*Forward* S
$Dst = h_4$	*Forward* W

Compared to the previous configuration, we have added an additional rule to separate out (and count!) HTTP traffic going from N to S. The counters associated with these rules could then be polled by the controller to compute the answer to the query. Alternatively, to measure the amount of traffic being generated by h_1 they might instead modify the configuration of N to the following:

N:

Match	**Actions**
$Dst = h_1$	*Forward* h_1
$Src = h_1, Dst = h_2$	*Forward* E
$Dst = h_2$	*Forward* E
$Src = h_1, Dst = h_3$	*Forward* S
$Dst = h_3$	*Forward* S
$Src = h_1, Dst = h_4$	*Forward* W
$Dst = h_4$	*Forward* W

Here, most rules have been split in two: one for traffic generated by h_1 and another for all other traffic.

As these scenarios illustrate, even in extremely simple applications, using forwarding rules to implement measurement tasks quickly becomes complicated. Moreover, the situation would be even worse if the programmer needed to implement multiple queries simultaneously, or if the forwarding configuration were being updated frequently in response to events such as topology changes.

Felix offers a dramatically simpler approach to implementing measurement in SDN. Rather than modifying switch configurations so that traffic statistics can be collected using hardware-level counters, Felix cleanly decouples measurement from forwarding and pushes all measurement tasks to the edge. End hosts are responsible for collecting fine-grained information at the edge of the network and a tree-structured overlay aggregates the results computed at each local host into the overall query result. We believe this approach offers a good division of labor between the network and end hosts: the network is responsible for forwarding traffic using efficient packet-processing hardware, while complex monitoring queries are implemented at the edge using the plentiful resources offered by end hosts.

At a technical level, the key advance that makes this design possible is a language-based framework for expressing queries and analyzing configurations. To allow programmers to formulate measurement queries in terms of the paths traversed by packets, we use a simple query language based on NetKAT, which is in turn based on regular expressions. Regular expressions offer natural primitives for describing paths through a graph and have been extensively studied in the literature. To compute the local predicates that are installed on end hosts, we develop a program analysis that takes a measurement query and a forwarding configuration and automatically calculates predicates that denote the set of input packets that satisfy the query.

Returning to the running example, given the forwarding configuration and the query involving HTTP traffic on the shortcut link, our system would automatically compute a predicate for h_1 that matches all traffic destined for h_3, and vice versa. Similarly, given the forwarding configuration and the query involving traffic generated by h_1, our system would simply compute a single predicate that matches all outgoing traffic on h_1. By counting the number of packets that match each predicate, Felix is able to efficiently compute the overall result of the query.

Overall, Felix offers dramatically simpler mechanisms for implementing rich measurement queries compared to competing approaches. The following sections present the technical insights behind our approach.

3. THE NETKAT LANGUAGE

This section briefly reviews the syntax and semantics of the NetKAT language, to set the stage for the new contributions described in the following sections. NetKAT is a domain-specific programming language for specifying and reasoning about network behavior [1, 10, 23]. The language offers high-level and modular constructs for constructing network programs, as well as sound and complete mechanisms for verifying formal properties automatically.

Syntax and Semantics. NetKAT models SDN programs as functions on packets histories, where a packet (pk) is a

NetKAT Syntax

$$
\begin{array}{rrcll}
\text{Naturals} & n & ::= & 0 \mid 1 \mid 2 \mid \dots & \\
\text{Fields} & f & ::= & f_1 \mid \cdots \mid f_k & \\
\text{Packets} & pk & ::= & \{f_1 = n_1, \cdots, f_k = n_k\} & \\
\text{Histories} & h & ::= & \langle pk \rangle \mid pk{::}h & \\
\text{Predicates} & a, b & ::= & \mathit{true} & \mathit{Identity} \\
 & & \mid & \mathit{false} & \mathit{Drop} \\
 & & \mid & f{=}n & \mathit{Test} \\
 & & \mid & a + b & \mathit{Disjunction} \\
 & & \mid & a \cdot b & \mathit{Conjunction} \\
 & & \mid & \neg a & \mathit{Negation} \\
\text{Programs} & p, p' & ::= & a & \mathit{Filter} \\
 & & \mid & f{\leftarrow}n & \mathit{Modification} \\
 & & \mid & p + p' & \mathit{Union} \\
 & & \mid & p \cdot p' & \mathit{Sequencing} \\
 & & \mid & p^* & \mathit{Iteration} \\
 & & \mid & sw_1 \twoheadrightarrow sw_2 & \mathit{Link}
\end{array}
$$

NetKAT Semantics

$$
\llbracket p \rrbracket \in \mathsf{History} \to \mathcal{P}(\mathsf{History})
$$

$$
\begin{array}{rl}
\llbracket \mathit{true} \rrbracket\ h & \triangleq \{h\} \\
\llbracket \mathit{false} \rrbracket\ h & \triangleq \{\} \\
\llbracket f{=}n \rrbracket\ (pk{::}h) & \triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases} \\
\llbracket \neg a \rrbracket\ h & \triangleq \{h\} \setminus (\llbracket a \rrbracket\ h) \\
\llbracket f{\leftarrow}n \rrbracket\ (pk{::}h) & \triangleq \{pk[f := n]{::}h\} \\
\llbracket p + p' \rrbracket\ h & \triangleq \llbracket p \rrbracket\ h \cup \llbracket p' \rrbracket\ h \\
\llbracket p \cdot p' \rrbracket\ h & \triangleq (\llbracket p \rrbracket \bullet \llbracket p' \rrbracket)\ h \\
\llbracket p^* \rrbracket\ h & \triangleq \bigcup_i F^i\ h \\
\multicolumn{2}{c}{\text{where } F^0\ h \triangleq \{h\} \text{ and } F^{i+1}\ h \ \triangleq\ (\llbracket p \rrbracket \bullet F^i)\ h} \\
\llbracket sw_1 \twoheadrightarrow sw_2 \rrbracket\ (pk{::}h) & \triangleq \begin{cases} \{pk'{::}pk'{::}pk{::}h\} \\ \quad \text{where } pk' = pk[sw := sw_2] \\ \quad \text{if } pk.sw = sw_1 \\ \{\}\ \text{otherwise} \end{cases}
\end{array}
$$

Figure 2: NetKAT syntax and semantics.

record of fields and a history (h) is a non-empty list of packets. This is unlike the programming interfaces offered by most controllers, which are based on lower-level construct such as forwarding table rules. Fields f range over standard packet headers such as Ethernet source and destination addresses, VLAN tags, etc., as well as special fields that indicate the switch (sw) and physical port (pt) where the packet is located in the network.

More formally, NetKAT is defined by the definitions given in Figure 2. The syntax is described by a grammar in standard BNF notation while the semantics is described by a set of equations of the form $\llbracket p \rrbracket\ h = H$. Intuitively, this notation means that the function described by p maps input history h to a set of output histories H. Each predicate a describes a boolean predicate on packets and includes primitives tests $f{=}n$, which check whether field f is equal to n, as well as the standard boolean operators. Each program p describes a function that takes a packet history as input and generates a set of packet histories as output. A filter a drops packets that do not satisfy a; a modification $f{\leftarrow}n$ updates the f field to n; a union $p + p'$ copies the input packet, processes one copy using p and the other copy using p', and takes the union of the resulting sets; a sequence $p \cdot p'$ processes the input packet using p and then feeds each output of p into p' (the symbol $\bullet$ indicates this form of composition, which is also known as Kleisli composition); an iteration p^* behaves like the union of p composed with itself zero or more times; and a link $sw_1 \twoheadrightarrow sw_2$ forwards from sw_1 to sw_2.

Encoding Network-Wide Behavior. One way to use the NetKAT language is as an SDN programming framework: the programmer specifies a collection of network-wide forwarding paths using boolean predicates and regular operators, and the compiler generates local forwarding rules that implement that behavior. However, NetKAT is also useful in situations where the configurations are expressed directly in terms of low-level forwarding tables: the programmer encodes the topology and configurations as NetKAT programs and uses the tools provided by the language to reason about their behavior. In this paper, we focus on this latter use.

To encode network-wide behavior in NetKAT, we proceed in several steps. A single forwarding rule can be encoded as the sequential composition of a predicate that encodes the pattern of the rule and a program that encodes the action. An action that forwards a packet on a switch is encoded as a modification to the pt field; multiple actions can be composed using union and sequence. To model tables, we use conditionals, which can be encoded as follows:

$$
\mathit{if}\ a\ \mathit{then}\ p_1\ \mathit{else}\ p_2 \triangleq (a \cdot p_1) + (\neg a \cdot p_2)
$$

A table is a cascade of nested conditionals, sorted in order of priority. A configuration can then be encoded as a union of tables, one for each switch. Similarly, a topology can be encoded as a union of links. Finally, given predicates in and out that capture ingress and egress locations, we can model the end-to-end behavior of the network with forwarding policy p and topology t as follows:

$$
\mathit{in} \cdot (p \cdot t)^* \cdot p \cdot \mathit{out}
$$

Intuitively, this program accepts incoming packets and repeatedly forwards them across switches and links until they exit the network.

Importantly, programmers do not need to write NetKAT programs to use Felix—we use the langauge only as a model of the forwarding behavior of the network. Although NetKAT programs denote deterministic functions, richer features can also be encoded, provided one only needs to model reachability. For example, a configuration that uses ECMP to forward traffic randomly along multiple paths can be encoded using NetKAT's union operator, with one sub-term for each path. Hence, we believe our solution is broadly applicable.

Language Model and NetKAT Automata. A unique feature of NetKAT is that programs can be characterized either in terms of the standard model based on packet-processing functions or equivalently in terms of a language-theoretic model based on regular sets. The sets used in the latter model can be encoded using finite automata, which provides a concrete basis for analyzing and verifying NetKAT programs. NetKAT automata are similar to classic finite automata, but are extended to handle functions on packets rather than recognizing sets of strings.

Query Syntax

$$\begin{array}{lrll} \text{Queries} & q, q' ::= & (a, b) & \textit{Filter} \\ & | & q + q' & \textit{Union} \\ & | & q \cdot q' & \textit{Sequencing} \\ & | & q^* & \textit{Iteration} \end{array}$$

Predicate Semantics

$$\mathcal{A}[\![a]\!] \in \mathcal{P}(\mathsf{Packet})$$
$$\mathcal{A}[\![a]\!] \triangleq \{pk \mid [\![a]\!]\langle pk\rangle \neq \{\}\}$$

Query Semantics

$$\begin{aligned} &\mathcal{Q}[\![q]\!] \in \mathcal{P}((\mathsf{History} \ \cup \ \{\langle\rangle\})) \\ &\mathcal{Q}[\![(a,b)]\!] \triangleq \{pk'::\langle pk\rangle \mid pk \in \mathcal{A}[\![a]\!] \text{ and } pk' \in \mathcal{A}[\![b]\!]\} \\ &\mathcal{Q}[\![q + q']\!] \triangleq \mathcal{Q}[\![q]\!] \cup \mathcal{Q}[\![q']\!] \\ &\mathcal{Q}[\![q \cdot q']\!] \triangleq \{h' \ @ \ h \mid h \in \mathcal{Q}[\![q]\!] \text{ and } h' \in \mathcal{Q}[\![q']\!]\} \\ &\mathcal{Q}[\![q^*]\!] \triangleq \{\langle\rangle\} \cup \left(\textstyle\bigcup_{i \in \mathbb{N}^+} \mathcal{Q}[\![q^i]\!]\right) \\ &\qquad \text{where } q^1 \ \triangleq q \text{ and } q^{i+1} \ \triangleq \ q \cdot q^i \end{aligned}$$

Figure 3: Query Language Syntax and Semantics

Definition 1 (NetKAT Automaton)**.** *A NetKAT automaton is a tuple* (S, s_0, E, D)*, where:*

- *S is a finite set of states,*
- *$s_0 \in S$ is the start state,*
- *$E : S \to \mathsf{Pk} \to \mathcal{P}(\mathsf{Pk})$ is the* observation function*, and*
- *$D : S \to \mathsf{Pk} \to \mathcal{P}(\mathsf{Pk} \times S)$ is the* continuation function*.*

Intuitively, the observation function E encodes the input-output behavior at each state, while the transition function D encodes the forwarding behavior from the current state across a link in the topology. The packet histories used in the standard semantics are encoded in terms of transitions from the initial state ending with an observation. Prior work by some of the authors developed efficient algorithms for translating NetKAT programs to automata based on derivatives [10], and a compact representation of automata based on (a small extension) of binary decision diagrams (BDDs) [23].

The key point for the purposes of this paper is that the observation and transition functions provide compact representations of network behavior that can be used as a basis for analysis. If we syntactically replace all occurrences of the link primitive $sw_1 \twoheadrightarrow sw_2$ with $sw{=}sw_1 \cdot sw{\leftarrow}sw_2$, which move the packet from sw_1 to sw_2 but do not extend the packet history, then the observation function E function encodes reachability directly. This syntactic translation will be defined formally as $\Phi(p)$ in Section 4. Moreover, the data structures used in our implementation support extracting predicates that represent the domain of the observation function—i.e., the inputs it maps to non-empty outputs.

4. QUERY LANGUAGE

This section presents the high-level language used to define measurement queries in Felix. We define the syntax and semantics of the language formally and provide a number of example queries. Intuitively, queries match the histories (as defined in Section 3) that satisfy the path property we are measuring.

Syntax. The syntax of Felix's query language is defined by the grammar in Figure 3. The simplest query is a pair of NetKAT predicates ((a, b)) that, intuitively, describes the input-output behavior at the current location in the network. More complicated queries can be expressed using the regular operators: union ($q + q'$), sequencing ($q \cdot q'$), and iteration (q^*). These operators allow programmers to naturally express queries that measure traffic on end-to-end paths through the network.

Semantics. Semantically, a query denotes a set of histories. Queries can be understood as regular expressions over an "alphabet" of packet pairs where a query's denotation corresponds to the regular expression's language of histories. A history models the path a packet takes through the network where the elements of the history model the state of the packet before and after traversing each link. Unlike NetKAT, the "empty" history $\langle\rangle$ is a possible query result.

For example, the query $(\mathsf{sw}{=}sw_1, \mathsf{sw}{=}sw_2)$ describes the set of all histories $pk_2{::}\langle pk_1\rangle$ where the switch field sw of pk_2 is sw_2 and the switch field sw of pk_1 is sw_1. Such a history is produced by any packet that traverses the link from switch sw_1 to switch sw_2. The query $(\mathit{true}, \mathit{true})$ matches traffic across any link while the query $(\mathit{false}, \mathit{false})$ matches nothing. The concatenation of two queries $(q \cdot q')$ denotes the set of histories obtained by concatenating a result from each sub-query. For example, the query $(\mathsf{sw}{=}sw_1, \mathsf{sw}{=}sw_2) \cdot (\mathsf{sw}{=}sw_2, \mathsf{sw}{=}sw_3)$ matches traffic that flows from switch sw_1 to sw_2 and then from switch sw_2 to sw_3. The union of two queries $(q + q')$ denotes the union of the histories matched by the queries. For example, $(\mathsf{sw}{=}sw_1, \mathsf{sw}{=}sw_2) + (\mathsf{sw}{=}sw_2, \mathsf{sw}{=}sw_1)$ matches traffic that flows from switch sw_1 to sw_2 or from switch sw_2 to sw_1. That is, it matches traffic on the bidirectional link between switch sw_1 and sw_2. The iteration of a query (q^*) represents the infinite union of repeated sequencing of a query with itself. For example, the query $(\mathit{true}, \mathit{true})^*$ matches packets that traverse an arbitrary number of links—i.e., all traffic in the network.

The semantics of our query language is defined by the equations in Figure 3. To streamline the definition, we use an alternate formulation of the semantics for predicates that is equivalent to the standard version given in Figure 2.

Example Queries. Many common measurement tasks can be expressed using Felix's query language.

- ***n*-hop Traffic:** The query $(\mathit{true}, \mathit{true})$ matches traffic along a single link. The query $(\mathit{true}, \mathit{true}) \cdot (\mathit{true}, \mathit{true})$ matches traffic along 2-hop paths across two links. The following query matches traffic across n-hop paths:

  $$(\mathit{true}, \mathit{true})^n$$

  Similarly, we can measure the traffic across paths with n or fewer hops by constructing the union over each path length:

  $$\sum_{i=1}^{n} (\mathit{true}, \mathit{true})^i$$

- **Link Monitoring:** We can measure the traffic along any path that traverses the link from sw_1 to sw_2:

$$\begin{gathered}(true, true)^* \cdot \\ (\mathsf{sw}{=}sw_1, \mathsf{sw}{=}sw_2) \cdot \\ (true, true)^*\end{gathered}$$

  Intuitively, the query matches paths with an arbitrary prefix and suffix so long as the path includes a link between switch sw_1 and switch sw_2. We can easily extend this query to match all paths that include a subpath. For example, the following query matches any paths that include a path from switch sw_1 to switch sw_2, from switch sw_2 to switch sw_3, and from switch sw_3 to switch sw_4.

$$\begin{gathered}(true, true)^* \cdot \\ (\mathsf{sw}{=}sw_1, \mathsf{sw}{=}sw_2) \cdot \\ (\mathsf{sw}{=}sw_2, \mathsf{sw}{=}sw_3) \cdot \\ (\mathsf{sw}{=}sw_3, \mathsf{sw}{=}sw_4) \cdot \\ (true, true)^*\end{gathered}$$

  Furthermore, we are able to extend this to measure all traffic between two switches. For example, the following query matches all paths from sw_1 to sw_4. Note that we include a $(true, true)^*$ between sw_1 and sw_4 to represent all paths between the two switches.

$$\begin{gathered}(true, true)^* \cdot \\ (\mathsf{sw}{=}sw_1, true) \cdot \\ (true, true)^* \cdot \\ (true, \mathsf{sw}{=}sw_4) \cdot \\ (true, true)^*\end{gathered}$$

  In all three of these queries, we are only measuring directed traffic from one switch to another. If we wanted to measure bidirectional traffic in a network, we could take the union of both unidirectional paths.

- **Switch Monitoring:** Finally, the query

$$(true, true)^* \cdot (\mathsf{sw}{=}sw_1 \cdot true) \cdot (true, true)^*$$

  matches all traffic that exits switch sw_1 at some point. Similarly, the query

$$(true, true)^* \cdot (true \cdot \mathsf{sw}{=}sw_1) \cdot (true, true)^*$$

  matches all traffic that enters switch sw_1 at some point. We can combine these two queries into a single query that matches all traffic that passes through switch sw_1 at some point.

$$\begin{gathered}(true, true)^* \cdot \\ ((\mathsf{sw}{=}sw_1, true) + (true, \mathsf{sw}{=}sw_1)) \cdot \\ (true, true)^*\end{gathered}$$

Additional examples of Felix queries are given in Table 1.

5. QUERY COMPILATION

Existing SDN platforms often implement measurement tasks by installing forwarding rules on switches. Felix takes a different approach and instead uses predicates evaluated on end hosts. Given a query and a configuration, the Felix compiler computes these predicates automatically, using a program analysis based on NetKAT automata.

As an example to illustrate, suppose that we want to measure the traffic along the link from switch sw_1 port pt_2 to switch sw_2 port pt_1 in the linear topology given in Figure 4. A *switch* based measuring approach could install a forwarding rule on switch sw_1 to tally packets outbound on port pt_2. Felix would instead analyze the network via NetKAT and produce a set of predicates to install on host h_1 and host h_2. In this simple example, since all outgoing traffic from host h_1 will wind up traveling through sw_1, we would only need to measure outgoing traffic from h_1.

The analysis proceeds in two steps. First, we compile a NetKAT encoded network and a query into a NetKAT term that forwards packets according to the network and drops packets that aren't matched by our query. Second, we "read off" the predicates from the E function of the NetKAT automata associated with the compiled term and install them as predicates on end hosts. We currently assume that the configuration is loop and blackhole free, but we do not believe this is an essential restriction.

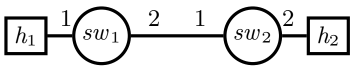


Figure 4: Example linear topology. Packets destined for host h_1 are forwarded out on port 1 of each switch, and packets destined for host h_2 are forwarded out on port 2.

$$\begin{aligned}&\Phi(p) \in \mathsf{NetKAT}\\ \Phi(a) &\triangleq a\\ \Phi(f{\leftarrow}n) &\triangleq f{\leftarrow}n\\ \Phi(p + p') &\triangleq \Phi(p) + \Phi(p')\\ \Phi(p \cdot p') &\triangleq \Phi(p) \cdot \Phi(p')\\ \Phi(p^*) &\triangleq \Phi(p)^*\\ \Phi(sw_1 \to sw_2) &\triangleq \mathsf{sw}{=}sw_1 \cdot \mathsf{sw}{\leftarrow}sw_2\end{aligned}$$

$$\begin{aligned}&c_{net}(q) \in \mathsf{NetKAT}\\ c_{net}((a,b)) &\triangleq p \cdot a \cdot \Phi(t) \cdot b\\ c_{net}(q + q') &\triangleq c_{net}(q) + c_{net}(q')\\ c_{net}(q \cdot q') &\triangleq c_{net}(q) \cdot c_{net}(q')\\ c_{net}(q^*) &\triangleq c_{net}(q)^*\end{aligned}$$

$$\begin{aligned}&C_{net}(q) \in \mathsf{NetKAT}\\ C_{net}(q) &\triangleq in \cdot c_{net}(q) \cdot p \cdot out\end{aligned}$$

Figure 5: Query Compilation Rules.

Compilation Rules. The rules for compiling a query q into a NetKAT term is given in Figure 5. The compiler takes a model of the network being measured as input, so the compilation function $C_{net}(\cdot)$ is parameterized on a network 4-tuple encoding $net = (in, p, t, out)$. The function $C_{net}(\cdot)$ uses a helper functions $c_{net}(\cdot)$ and Φ—the latter replaces all links $sw_1 \to sw_2$ in a NetKAT term with a corresponding filter-modification pair: $\mathsf{sw}{=}sw_1 \cdot \mathsf{sw}{\leftarrow}sw_2$.

Name	Description	Query
DROP	no paths	$(\mathit{false}, \mathit{false})$
i-HOP	i-hop paths	$(\mathit{true}, \mathit{true})^i$
i-ALL	all paths	$(\mathit{true}, \mathit{true})^{*i}$
HTTP	HTTP traffic	$(\mathsf{dport}{=}80, \mathit{true})^*$
SW4OR5	paths through sw_4 or sw_5	$((\mathit{true}, \mathit{true})^* \cdot ((\mathsf{sw}{=}sw_4, \mathit{true}) + (\mathit{true}, \mathsf{sw}{=}sw_4)) \cdot (\mathit{true}, \mathit{true})^*) +$ $((\mathit{true}, \mathit{true})^* \cdot ((\mathsf{sw}{=}sw_5, \mathit{true}) + (\mathit{true}, \mathsf{sw}{=}sw_5)) \cdot (\mathit{true}, \mathit{true})^*)$
LONG-PATH	long path	$(\mathit{true}, \mathit{true})^* \cdot$ $(\mathsf{sw}{=}sw_1 + \mathsf{sw}{=}sw_2 + \mathsf{sw}{=}sw_3, \mathsf{sw}{=}sw_4 + \mathsf{sw}{=}sw_5 + \mathsf{sw}{=}sw_6) \cdot$ $(\mathsf{sw}{=}sw_4 + \mathsf{sw}{=}sw_5 + \mathsf{sw}{=}sw_6, \mathsf{sw}{=}sw_7 + \mathsf{sw}{=}sw_8 + \mathsf{sw}{=}sw_9) \cdot$ $(\mathsf{sw}{=}sw_7 + \mathsf{sw}{=}sw_8 + \mathsf{sw}{=}sw_9, \mathsf{sw}{=}sw_{10} + \mathsf{sw}{=}sw_{11} + \mathsf{sw}{=}sw_{12}) \cdot$ $(\mathit{true}, \mathit{true})^*$

Table 1: Example queries.

Recall that a link records the state of the packet in the history before and after traversing the link. Intuitively, $C_{net}(q)$ is a modified version of $in \cdot (p \cdot t)^* \cdot p \cdot out$ where the links in t are surrounded by pairs of predicates in q. The compiled term forwards traffic identically to $in \cdot (p \cdot t)^* \cdot p \cdot out$, but rather than recording the packet state into the history before and after traversing a link, it instead drops packets that are not matched by the query q.

For example, again consider the linear topology in Figure 4. In this example, we are using the *net* modeling the network in the figure. We can measure the traffic from switch sw_1 port pt_2 to switch sw_2 port pt_1 with the following query:

$$q_{1\to 2} \triangleq (sw_1 : 2, sw_2 : 1)$$

If we abbreviate predicates $\mathsf{sw}{=}sw_i \cdot \mathsf{pt}{=}pt_j$ as $sw_i : j$, then the program $q_{1\to 2}$ is compiled as follows:

$$C_{net}(q_{1\to 2}) = in \cdot p \cdot (sw_1 : 2) \cdot \Phi(t) \cdot (sw_2 : 1) \cdot p \cdot out$$

The program $C_{net}(q_{1\to 2})$ behaves like $in \cdot (p \cdot t)^* \cdot p \cdot out$; it filters packets that enter the network (in), forwards packets through switches (p), transports packets across links (t), and filters traffic exiting the network (out). Unlike $in \cdot (p \cdot t)^* \cdot p \cdot out$, however, $C_{net}(q_{1\to 2})$ includes only a single t, meaning that it transports traffic across a link exactly once. Thus, $C_{net}(q_{1\to 2})$ models a 1-hop network as intended. Moreover, before it transports traffic across a link, it filters packets that begin at switch sw_1 port pt_2 before traversing the link and filters packets that end at switch sw_2 port pt_1 after traversing the link. In general, $C_{net}(q)$ only delivers packets that fully traverse the network and produce histories in $\mathcal{Q}[\![q]\!]$.

Installing Predicates. After compiling the query q into a NetKAT term $C_{net}(q)$, we "read off" a collection of predicates to install on hosts, to count the number of packets that traverse paths specified by q. Semantically, the predicate set for a query q and network *net* is the following set where α and β are "complete" predicates that test the value of every field in the packet and pkt_α is the packet satisfying the predicate α:

$$\{(\alpha, \beta) \mid \langle pkt_\beta \rangle \in [\![C_{net}(q)]\!] \langle pkt_\alpha \rangle\}$$

This set includes a pair of predicates (α, β) for each each packet pkt_α that satisfies the query q and exits the network matching β. Before a host sends a packet, it first checks to see if the packet is matched by some α_i. If it is, the host tags the packet with a unique identifier i. Similarly, whenever a host receives a packet, it first checks to see if the packet is tagged with i. If it is, then the host tallies the packet if and only if it is matched by β_i.

Each host keeps track of the total number of packets matched by its predicate set. Hosts also record the number of matching packets sent by each host. This is useful for generating traffic matrices, as discussed in Section 7. Moreover, we use a virtual overlay to aggregate these statistics and perform real-time queries. This is described in detail in Section 6.

In theory, a predicate set could be very large. We exploit a compact representation of the observation function E for $C_{net}(q)$ using forwarding decision diagrams (FDDs). Rather than generating a set of complete tests, we read off predicates from the FDDs where each predicate is a simple conjunction of atomic tests. In practice, these sets are typically much smaller. However, by using incomplete tests, hosts must tag packets differently because in general, a packet may be matched by several incomplete tests. Thus, before an end host sends a packet, it must tag it with i for every α_i that matches the packet. Similarly, an end host tallies a packet with a set of tags T if and only if the packet matches some β_j and there is a tag $i \in T$ where (α_i, β_j) is in the predicate set installed on the end host. If all incomplete tests happen to be disjoint, then packets only require a single tag similar to the scenario of using complete tests. Currently, Felix assumes all incomplete tests are disjoint.

As an example, Felix generates the following singleton set of predicates for $C_{net}(q_{1\to 2})$:

$$\{(sw_1 : 1 \cdot \mathsf{dst}{=}h_2, sw_2 : 2 \cdot \mathsf{dst}{=}h_2)\}$$

Intuitively, all traffic along the link from sw_1 to sw_2 is made up of traffic originating on host h_1 and destined for h_2.

Formal Properties. Whenever a packet is received and tallied by an end host for a query q, we would like the guarantee that the history associated with the packet is in the denotation of q. Likewise, whenever a packet is *not* tallied, we would like the guarantee that the history associated with the packet is *not* in the denotation of q. We formalize this intuition with Theorem 1. For simplicity, we assume all tests are complete, though extending the theorem to handle incomplete tests is not difficult.

Theorem 1. *Let $P_{net}(q)$ denote the predicate set of q compiled on network net. For all predicates (in, out), forwarding policies (p), topologies (t), queries (q), and packets*

```
# configure overlay
{
    'type': 'add_leaf_agent',
    'agent_addr': '10.0.0.2'
}
# configure counters
{
    'type': 'config_counter',
    'counter_key_type': 'src',
    'increment': 'pkt'
}
# query counters
{
    'type': 'query_counter',
    'counter_key_type': 'src',
    'counter_key': '10.0.0.2'
}
```

Figure 6: Example agent API operations.

(pkt_α), we have:

$$\exists(\alpha,\beta) \in P_{net}(q)$$
$$\Longleftrightarrow$$
$$\exists pkt_\beta{::}h \in [\![in \cdot (p \cdot t)^* \cdot p \cdot out]\!]\,\langle pkt_\alpha\rangle.\ h \in \mathcal{Q}[\![q]\!]$$

The left hand side of the implication says that if the packet pkt_α is sent through the network, it will be tagged by some sending host and will arrive at some receiving host as pkt_β where it will be tallied by Felix. The right hand side of the implication says that there is some history $pkt_\beta{::}h$ that pkt_α takes through the network that is matched by the query.

Note that some "obvious" stronger versions of Theorem 1 do not hold. For example, the property for all in, out, p, t, q, pkt_α, and all $pkt_\beta{::}h \in [\![in \cdot (p \cdot t)^* \cdot p \cdot out]\!]\,\langle pkt_\alpha\rangle$,

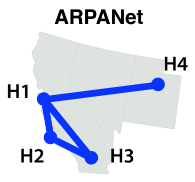
$$(\alpha,\beta) \in P_{net}(q) \iff h \in \mathcal{Q}[\![q]\!]$$

in which we universally quantify $pkt_\beta{::}h$ instead of existentially quantifying it is not valid! If pkt_α produces multiple histories of the form $pkt_\beta{::}\_$, then it is possible that some of the histories of pkt_α would not be matched by the query. However, assuming that whenever a packet is duplicated, the two copies do not subsequently reconverge and depart the network as identical packets, this limitation is irrelevant.

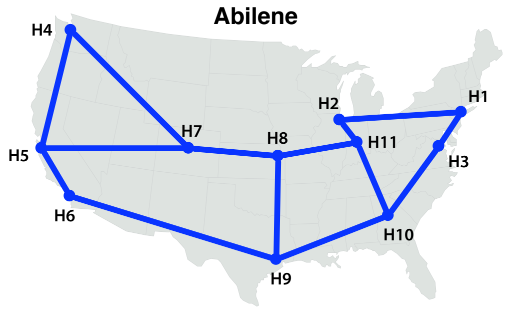
6. IMPLEMENTATION

We have built a prototype implementation of Felix in C, OCaml, and Python. Our implementation includes the Felix query compiler, an end-host monitor, end-host agent, and SDN controller. The end-host monitor gathers local statistics about incoming and outgoing traffic; the end-host agent communicates statistics to the controller using a tree-structured overlay; and the controller manages the configurations installed on switches and coordinates the behavior of the end hosts.

End-Host Monitor. The *end-host monitor* monitors incoming and outgoing traffic and then tallies the traffic that matches an installed set of predicates. We implement the monitor using `iptables` [16]. Given an incoming or outgoing packet, we first match it against the filtering rules managed by `iptables`. If the packet matches one of the rules, we send it to the monitor, which uses `scapy` [3] to parse the packet, extract certain fields relevant to the query,



(a)



(b)

Figure 7: Case studies: topologies.

and update counters. For example, if an end-host is configured to group packets by source IP address, the monitor would extract the source IP address of every incoming packet and increment the counter that corresponds to the extracted IP address. The monitor can aggregate counters in several ways: by packet fields, top-k heavy-hitters, sliding-windows, and using count-min sketches [6].

End-Host Agent. The end-host agent implements two essential functions: it parses messages from the SDN controller and from other end-hosts to configure and query the monitor, and it establishes a tree-structured overlay among end-hosts for aggregating data. The agent provides a JSON API for network operators to configure measurement tasks and query data on end-hosts. The agent on an end-host processes requests and forwards them to the agent on the same end-host. The agent provides a JSON API for network operators to set up a virtual overlay of end-hosts by connecting hosts in the overlay to form a virtual tree topology. Some example commands from this API are shown in Figure 6.

Controller. Finally, the controller manages the forwarding rules installed on switches, invokes the Felix compiler to generate the predicates associated with each query, installs predicates on end-hosts monitors, and issues queries to end host agent (either directly or using the overlay).

7. EVALUATION

To evaluate our design for Felix, we built case studies and conducted experiments to quantify the performance of the compiler.

7.1 Case Studies

We built two case studies based on realistic (if small) applications, and executed them with Mininet, using Felix to answer a variety of traffic measurement queries.

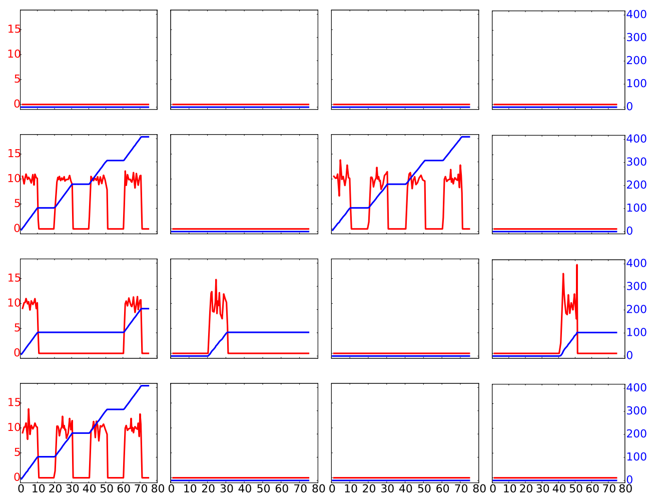
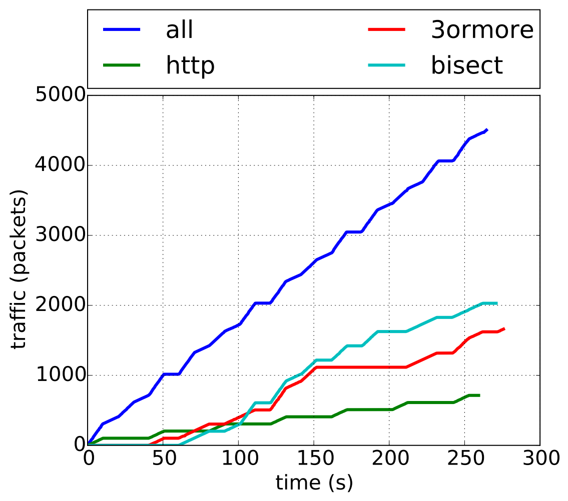


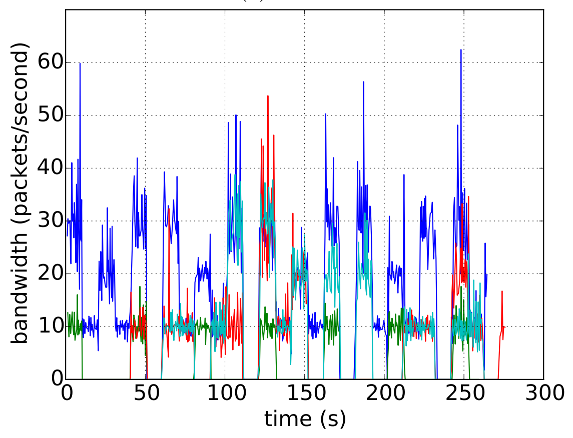
Figure 8: Case study: ARPANet traffic matrices. The plot in row i column j gives the plot of traffic from host i to host j in the four-node Arpanet topology over an 80-second experiment; the red line and left y-axis gives rate (packets per second); the blue line and right y-axis gives total traffic (packets); the x-axis gives time (seconds).

ALL	$(\mathsf{dport}{=}8888 \vee 5001, \mathit{true})^*$
HTTP	$(\mathsf{dport}{=}8888, \mathit{true})^*$
3ORMORE	$(\mathsf{dport}{=}8888 \vee 5001, \mathit{true})^* \cdot$ $(\mathit{true}, \mathit{true})^2 \cdot$ $(\mathit{true}, \mathit{true})^*$
BISECT	$(\mathit{true}, \mathit{true})^* \cdot$ $(\mathsf{dport}{=}8888 \vee 5001 \cdot \mathsf{sw}{=}8, \mathsf{sw}{=}11) \cdot$ $(\mathit{true}, \mathit{true})^* +$ $(\mathit{true}, \mathit{true})^* \cdot$ $(\mathsf{dport}{=}8888 \vee 5001 \cdot \mathsf{sw}{=}9, \mathsf{sw}{=}10) \cdot$ $(\mathit{true}, \mathit{true})^*$

Table 2: Case study: Abilene queries.



(a) Traffic



(b) Bandwidth

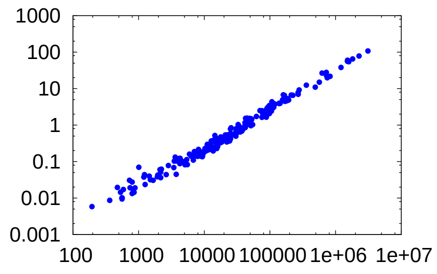
Figure 9: Case study: Abilene results.

Topologies and Traffic. The topologies for our case studies are drawn from the Topology Zoo: a public data set comprising 260 real-world network topologies [15]. We wrote NetKAT policies that forward traffic along shortest paths and generated traffic synthetically using `iperf`. More specifically, we configured each host to run `iperf` servers on ports 5001 and 8888, and generated periodic bursts of UDP traffic to other hosts using simple patterns as described below.

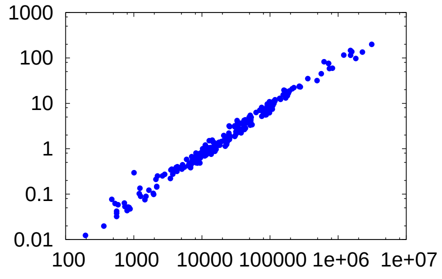
Traffic Matrices. For our first case study, we used Felix to collecte traffic matrices for each host in the four-node ARPANet topology shown in Figure 7a over an 80-second experiment. We configured hosts 2, 3, and 4 to periodically send 10-second bursts of traffic at a rate of roughly 10 packets per second, with 10 seconds delays between bursts. Host 2 sends traffic concurrently to hosts 1 and 3. Host 3 sends traffic to host 1, then host 2, then host 4. Host 3 repeatedly sends traffic to host 1. Using these patterns we compiled and installed the predicates corresponding to the query $(\mathit{true}, \mathit{true})^*$ on each host. We then repeatedly queried each end host for packet counts aggregated by source. This generates a 4×4 matrix indexed by source hosts on the left axis and destination hosts on the top axis where entry (i, j) shows the traffic and bandwidth sent from host i to host j, as shown in Figure 8. The plots in Figure 8 show traffic rates in red and total traffic in blue.

Multiple Queries. For our second case study, we ran multiple queries using the 11-node Abilene topology shown in Figure 7b, using a simple traffic pattern. We used three hosts to generate bursts of traffic as follows: First, host 8 sends traffic in a round-robin fashion to hosts 7, 4, 5, 6, 9, 10, 3, 1, 2, and 11 on port 5001. Second, host 9 sends traffic in a round-robin fashion to hosts 6, 5, 4, 7, 8, 11, 2, 1, 3, and 10; traffic to hosts 6, 4, 8, 2, and 3 is sent to port 8888, and the other traffic is sent to port 5001. Finally, host 4 sends traffic to hosts 7, 8, 11, 2, and 1 all on port 5001; these paths are of length 1, 2, 3, 4, and 5 respectively. Hosts 8 and 9 generate 10 second bursts of traffic with 10 second delays. Host 4 generates 20 second bursts of traffic with 10 second delays. We then ran four queries over this traffic pattern, as shown in Table 2: ALL measures the total amount of traffic; HTTP measures the amount of traffic destined for port 8888; 3ORMORE measures the amount of traffic that traverses a path of three or more hops; and BISECT measures the amount of

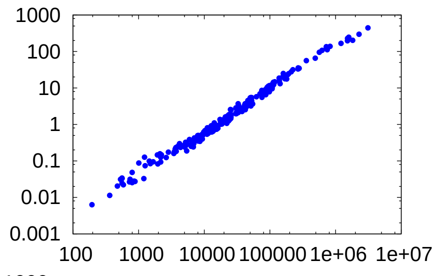
(a) DROP



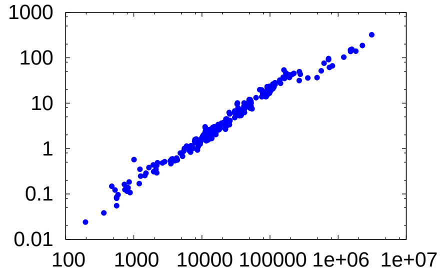
(b) 5-HOP



(c) 1-ALL



(d) SW4OR5



(e) LONG-PATH

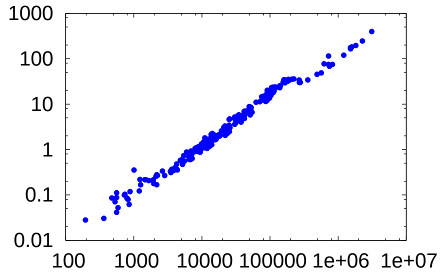


Figure 10: Experimental results: compilation for a variety of queries on Topology Zoo. The x-axis is program size (# syntax nodes) and y-axis gives is running time (seconds).

traffic that traverses either the link from switch 8 to 11 or the link from switch 9 to 10—these links bisect the network.

The results of these queries are shown in Figure 9; Figure 9a and Figure 9b chart the total traffic and traffic rate respectively. As expected, traffic comes in periodic bursts; the traffic peaks when all three sending hosts are active and drops to zero when all three hosts are inactive. HTTP traffic is generated by host 9. 3ORMORE and BISECT traffic is produced by all three hosts.

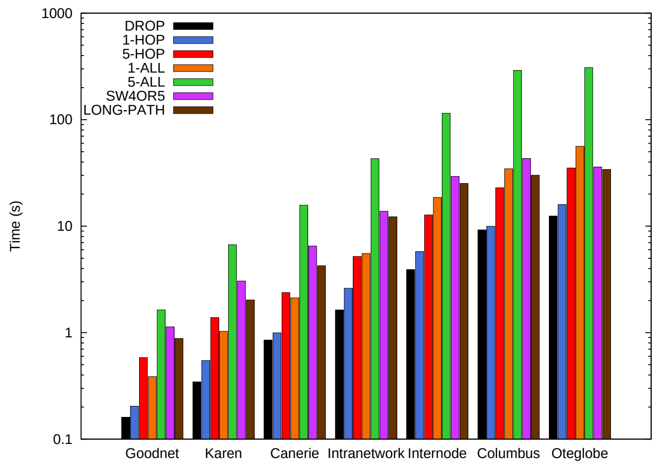


Figure 11: Experimental results on representative Topology Zoo networks. Each bar depicts the running time for the Felix compiler in seconds for a query in a given topology.

7.2 Experiments

To evaluate the performance of the query compiler, we implemented a variety of queries across a large number of topologies of varying sizes. When running these experiments, we had two main questions in mind: how quickly does the compiler calculate predicates for a variety of inputs and how well does it scale?

Benchmarks. To benchmark our system, we used topologies from the Topology Zoo, which have widely varying structure and scale, and shortest-path forwarding policies expressed in NetKAT. Topologies in this dataset range from 4-197 switches. When implemented in NetKAT, the forwarding policies range in size from 195-3111079 syntax tree nodes.

Queries. For each topology, we ran each of the queries defined in Table 1. We selected these queries for their diversity in function and complexity. They allow us to see how Felix scales and they make use of all of the query language's operators. The i-HOP, SW4OR5, HTTP, and LONG-PATH are especially interesting because they represent queries that would be likely to arise in pratice.

Methodology. We used a cluster of five Dell r620 servers, each with two eight-core 2.60 GHz Xeon CPU E5-2650 processors and 64 GB of RAM running Ubuntu 14.04.1 LTS. Running time was obtained in OCaml using the Jane Street `Time` library. Times reported include the time for compiling the query and generating the predicate set for the compiled query. Time for parsing, generating, and installing the policies is excluded.

Results and Analysis. The results of our experiments on Topology Zoo can be seen in Figure 10, Figure 12, and Figure 11. The scatter plots in Figure 10 compare the total term size for various topologies plotted against the time to run a given query over these topologies. Note that both axes on the scatter plots are logarithmic. The bar plots compare the time to run various queries over several representative topologies. In all plots, time is reported in seconds.

Topology	**Term Size**	**Switches**	DROP	5-HOP	1-ALL	SW4OR5	LONG-PATH
Goodnet	5949	17	0.16 / 0	0.58 / 0	0.39 / 289	1.13 / 82	0.88 / 0
Karen	18549	25	0.35 / 0	1.39 / 98	1.03 / 625	3.05 / 346	2.03 / 0
Canerie	32913	32	0.85 / 0	2.37 / 130	2.12 / 1024	6.51 / 582	4.25 / 110
Intranetwork	75585	39	1.63 / 0	5.19 / 138	5.53 / 1521	13.79 / 346	12.25 / 0
Internode	138123	66	3.91 / 0	12.75 / 840	18.62 / 4356	29.28 / 958	25.22 / 0
Columbus	278585	70	9.23 / 0	22.97 / 482	34.57 / 4900	43.23 / 1584	30.13 / 0
Oteglobe	358595	93	12.46 / 0	35.07 / 776	56.20 / 6906	35.87 / 1344	34.02 / 0

Figure 12: Experimental results on representative Topology Zoo Networks. The column for each query gives results of the form (t/p), where t is the amount of time needed to run the Felix compiler in seconds, and p is the number of predicates generated.

Overall, our implementation runs in less than a second on topologies with a small number of terms and scales linearly with term size. For example, with the trivial DROP query, we see terms in the 1000s taking .01 seconds, terms in the 10000s taking .1 seconds, and so on. For the 1-ALL query, we see terms in the 1000s taking .1 seconds, terms in the 10000s taking 1 second, and so on. The table and bar graphs in Figure 12 and Figure 11 depict detailed running times for selected topologies. A noticeable pattern is that queries with a star in them take longer than queries without a star.

Discussion. Overall, we believe these experiments show that our initial prototype performs well enough to usable across a variety of real-world topologies, configurations, and queries. We intend to explore optimizations that improve the performance of the compiler in future work.

8. RELATED WORK

There is an extensive literature on systems and abstractions for network measurement. We briefly review the work most closely related work to Felix.

The standard approach to network measurement is to sample traffic on certain links to collect repositories of flow records (or full packets) for offline analysis using standard formats such as sFlow and NetFlow. This approach effectively decouples forwarding from measurement since it uses separate mechanisms to implement each. However, unlike Felix, it does not allow operators to directly specify rich network-wide queries based on regular paths.

GigaScope pioneered the use of declarative query languages for network measurement [7]. The system offered a streaming SQL-like query language—i.e., evaluation was formulated in terms of sliding windows over streams of packets. It also provided the ability to use regular expressions to inspect packet payloads. The GigaScope compiler translated high-level queries to efficient code for a collection of heterogeneous devices.

A variety of measurement approaches have been explored in the context of SDN. Frenetic proposed a high-level language for monitoring network traffic using declarative query constructs [8]. Another early paper by Jose et al. exploited the capabilities provided by SDN controllers and switches to build a dynamic system that performed continuous monitoring of "heavy hitters" [12]. Work by Narayana et al. has investigated the problem of compiling regular path queries to forwarding tables [20]. All of these systems use forwarding tables to implement measurement functionality. Hence, they suffer from many of the limitations discussed in the early sections of this paper.

A notable exception is the HONE system, which proposed flexible abstractions for joint management of hosts and switches within a unified framework [24]. Like Felix, HONE uses end hosts to implement certain measurement tasks. However, HONE lacks abstractions for expressing, analyzing, and partitioning network-wide queries based on regular expressions. An interesting direction for future work would be to build a unified system that combines the features of HONE and Felix.

Network debugging is closely related to network measurement. An influential early paper by Handigol et al. on the `ndb` system proposed the idea of "network breakpoints" and "packet backtraces" to assist SDN programmers in developing correct programs [11]. To implement these features, `ndb` proposed implementing switches to generate packet digests that could be sent to a central repository for analysis. Like Felix, `ndb` can be used to obtain global visibility but the mechanisms are largely different.

Another active line of research is investigating data plane measurement primitives. Work on OpenSketch proposed a simple three-stage pipeline based on hashing, filtering, and counting, and demonstrated it could be implemented efficiently and used to express a variety of measurement tasks [27]. P4 offers a rich collection of primitives including stateful memory, hashing, etc. [4]. Compared to Felix, these systems focus mostly on measurement primitives at the data plane level. It would be interesting to explore using OpenSketch and P4 as platforms for implementing Felix's end-host monitor.

A large number of languages for SDN programming have been proposed in recent years. Languages such as Frenetic [8], NetCore [18], Pyretic [19], Maple [25], and NetKAT [1, 10, 23] have introduced high-level abstractions and semantics that enable programmers to reason precisely about the behavior of networks. Several different network programming languages based on logic programming have also been proposed including NDLog [17] and FlowLog [21].

There is also a growing body of work investigating applications of formal methods to SDN. NICE [5] uses a model checker and symbolic execution to find bugs in network programs written in Python. Automatic Test Packet Generation [28] constructs a set of packets that provide coverage for a given network-wide configuration. VeriCon [2] uses first-order logic and a notion of admissible topologies to automatically check network-wide properties. Several different systems have proposed techniques for checking network reachability properties including seminal work by Xie et al. [26], Header Space Analysis [13], VeriFlow[14], and the NetKAT verifier [10]. The program analysis used in Felix builds on the foundation provided by these tools.

9. CONCLUSION

This paper presents the design and implementation of Felix, a new SDN measurement system. Unlike most previous work, Felix uses a high-level language to express measurement queries and a program analysis to compute predicates that implement those queries at the edge. Cleanly separating forwarding and measurement in this way reduces the size and complexity of configurations, which has a number of important operational benefits. However, it does mean that query results are computed against an idealized model of the network.

There are several possible directions for future work. An obvious next step is to develop optimizations that improve the performance of our NetKAT analysis tool. Our current implementation is simple, but the NetKAT semantics provides a solid foundation for exploring optimizations. Another exciting direction involves hybrid approaches where measurement is not necessarily pushed to the edge, but merely pushed to certain devices. For example, we might perform measurement at the interface between mutually-distrusting islands but use Felix within each island. The same ideas might be useful for extending our techniques to handle congestion and bugs. Finally, we are interested in exploring extensions to handle stateful data planes as well as probabilistic network models [9].

Acknowledgments. The authors wish to thank the SOSR reviewers for helpful feedback. Our work is supported in part by NSF under grants CNS-1111698, CCF-1253165, CNS-1413972, and ACI-1440744; ONR under grant N00014-15-1-2177; a Google Faculty Research Award; and gifts from Cisco, Facebook, and Fujitsu.

10. REFERENCES

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *POPL*, pages 113–126, January 2014.

[2] Thomas Ball, Nikolaj Bjorner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. In *PLDI*, pages 282–293, June 2014.

[3] Philippe Biondi. Scapy. Available at http://www.secdev.org/projects/scapy/demo.html.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3):87–95, July 2014.

[5] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *NSDI*, April 2012.

[6] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, April 2005.

[7] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *ACM SIGMOD*, pages 647–651, 2003.

[8] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM ICFP*, 2011.

[9] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic NetKAT. In *ESOP*, 2016. To appear.

[10] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *POPL*, pages 343–355. ACM, 2015.

[11] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the debugger for my software-defined network? In *ACM HotSDN*, pages 55–60, 2012.

[12] Lavanya Jose, Minlan Yu, and Jennifer Rexford. Online measurement of large traffic aggregates on commodity switches. In *USENIX HotICE*, pages 13–13, 2011.

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

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

[15] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Selected Areas in Communications*, 29(9):1765–1775, October 2011.

[16] Linux. Iptables. Available at http://linux.die.net/man/8/iptables.

[17] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, August 2005.

[18] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *POPL*, pages 217–230, January 2012.

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

[20] Srinivas Narayana, Jennifer Rexford, and David Walker. Compiling path queries. In *NSDI*, 2016. To appear.

[21] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, April 2014.

[22] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *SIGCOMM*, 2012.

[23] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for NetKAT. In *ACM ICFP*, 2015.

[24] Peng Sun, Minlan Yu, Michael J. Freedman, and Jennifer Rexford. Hone: Joint host-network traffic management in software-defined network. *Journal of Network and System Management*, July 2014.

[25] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*, pages 87–98, August 2013.

[26] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *INFOCOM*, March 2005.

[27] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, pages 29–42, 2013.

[28] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CoNEXT*, pages 241–252, December 2012.