ReWiFlow: Restricted Wildcard OpenFlow Rules

Sajad Shirali-Shahreza, Yashar Ganjali
Department of Computer Science
University of Toronto
Toronto, Canada
{shirali,yganjali}@cs.toronto.edu

ABSTRACT

The ability to manage individual flows is a major benefit of Software-Defined Networking. The overheads of this fine-grained control, *e.g.* initial flow setup delay, can overcome the benefits, for example when we have many time-sensitive short flows. Coarse-grained control of groups of flows, on the other hand, can be very complex: each packet may match multiple rules, which requires conflict resolution. In this paper, we present ReWiFlow, a restricted class of OpenFlow wildcard rules (the fundamental way to control groups of flows in OpenFlow), which allows managing groups of flows with flexibility and without loss of performance. We demonstrate how ReWiFlow can be used to implement applications such as dynamic proactive routing. We also present a generalization of ReWiFlow, called Multi-ReWiFlow, and show how it can be used to efficiently represent access control rules collected from Stanford's backbone network.

Categories and Subject Descriptors

C.2.3 [**Communication Networks**]: Network Operations – *Network monitoring*

General Terms

Management, Measurement, Performance.

Keywords

OpenFlow, Proactive Routing, SDN, Wildcard Rule.

1. INTRODUCTION

Software-Defined Networking (SDN) simplifies programmatic manipulation and management of network traffic using a centralized view of the network. The ability to easily access and manage individual flows is a significant advantage of SDN in general, and OpenFlow specifically. For example, the controller can detect heavy-hitter flows, also known as elephant flows, and route them separately for better throughput [4].

While managing heavy long-lived flows individually can be beneficial, fine-grained management of short-lived flows can lead to unacceptable side effects. In a typical data center, while there are some heavy flows (*e.g.* VM migrations), the majority of flows are short (*e.g.* 80% are less than 10KB [2]) and some of these short flows are latency sensitive. In web service requests for example, even a fraction of a second extra delay can have a considerable effect on user experience [6].

The flow setup delay is one of the main side effects of individually managing flows: the first packet of each flow is kept in the first switch until the controller receives the *Packet-In* message from that switch, processes it, and replies back with instructions on how to handle that packet. Although this delay may be negligible for large and long-lasting flows, it can be significant for short and latency sensitive flows.

Buffer size limitation is another factor that impacts both short and long flows. OpenFlow switches usually buffer received packets and only send a small part of each packet to the controller when the packet does not match any flow entry in the forwarding table. However, switch buffers are usually small. Even in software switches (which there is no cost or power issue), buffers are relatively small. For example in the Open vSwitch (http://openvswitch.org/) the default size of this buffer is only 256 packets. As a result, under heavy loads, this buffer could quickly become full, forcing the switch to send the entire packets to the controller, which increases both transmission delay and controller load and leads to longer overall delays.

The limited size of flow tables in switches could also create problems. When the flow table becomes full, the switch must evict an old entry before inserting a new one. If the entry that is removed represents an active flow in the network, then the switch has to send any subsequent packets from that flow to the controller, as they do not match any flow table entries. Not only this creates a huge sudden load on the controller, it is possible that at least one of these packets is dropped in the controller, resulting in a significant drop in flow throughput.

Coarse-grained handling of flows, *i.e.* dealing with flows as groups, is one way to overcome the aforementioned problems. OpenFlow provides this functionality by wildcard rules. In contrast to exact match rules that can be used to manage individual flows, a wildcard rule describes how a set of flows (*e.g.* all flows from host A) should be handled.

The OpenFlow specification seems to move in a direction that exact match rules will be rarely used: while a flow entry match in OpenFlow 1.0 specification is defined as a 12-tuple, the OpenFlow Extensible Match (OXM) (introduced in OpenFlow 1.2) allows switches to support a variety of header fields. For example in the latest version (1.5), 44 different types are defined. As a result, almost all flow entries would become wildcard rules, as they will not contain a value for all of defined header fields, because some of those header fields are only applicable to specific types of packets.

Although wildcard rules resolve most of the limitations of exact match rules, they have their own set of challenges. Programming complexity is the main challenge of using wildcard rules: the inherent freedom and flexibility of choosing which header fields to match makes the problem combinatorially complex. We will review these challenges in more details in Section 2.

In this paper, we propose ReWiFlow, a restricted class of wildcard rules that make it easier to use wildcard rules, while improving performance. The basic idea is to trade the flexibility

of wildcard rules with the ease of dealing with them. In ReWiFlow, there is a defined ordering among all possible header fields that can appear in a wildcard rule. A header field can be present in the rule only if all prior fields (in the defined order) are present. For example, if destination IP is before destination port in the ordering, the destination port can appear (as an exact match) in the rule if the destination IP is also present.

We note that there is a partial ordering present in the current OpenFlow specification, known as flow match field prerequisite. However, it is defined for a small number of pairs. For example, if TCP Port is present, the IP Protocol should also be present and equal to TCP. What we propose here is a full ordering that, as we will show, has new advantages.

We prove the *prefix property* of ReWiFlow rules, which enables us to easily manage a collection of rules. We also present a generalization of ReWiFlow called Multi-ReWiFlow, which includes multiple groups of ReWiFlow rules with different orderings. This enables us to have the benefits of ReWiFlow in scenarios that no single ordering can accommodate all the required rules. We demonstrate how we can represent more than 1,600 access control rules (ACLs) of Stanford backbone network [20] with only 5 groups of ReWiFlow rules.

ReWiFlow mainly aims to reduce the programming complexity and overcome the hardware limitations of wildcard rules, but does not address loss of visibility that happens due to aggregation of flows. However, combining ReWiFlow with the recently proposed sampling extension FleXam [15] provides the best of both worlds: reduced complexity and improved efficiency without loss of visibility. We believe this combination leads to an elegant solution for some of the current challenges in. It can also open new doors to solve challenging problems such as anomaly and attack detection [14].

2. WILDCARD RULE CHALLENGES

2.1 Programming Complexity

The first challenge in dealing with wildcard rules is their programming complexity. To deal with network dynamics, we need to add new rules, delete rules, or split existing rules. There is an inherent complexity in choosing which header fields to select. Furthermore, different wildcard rules may interact with each other (*e.g.* a newly installed rule may match some of the packets that were handled by a previously installed rule), and analyzing such interactions is a complex and computationally intensive task [3].

Even managing Access Control Lists (ACLs), which are special wildcard rules that usually only contain source, destination, protocol and port number, is difficult for trained professionals. The existing solutions focus and are limited to visualization [10] and inconsistency checks [5]. Providing an automated solution to deal with wildcard rules in general is even more challenging.

In the case of OpenFlow wildcard rules, we need to automatically generate and update wildcard rules. For example in proactive routing, we start with a set of wildcard rules that can route all network traffic towards their destination. Since this is done a priori, the rules might not be optimal as network traffic changes over time. So we need to update the rules installed on switches by splitting existing rules, merging them, or adding new rules.

Rule Priorities. Each packet can match at most one exact match rule. However, a packet may match multiple wildcard rules. Therefore, when using wildcard rules, we should consider rule overlaps, and assign priorities to ensure packets match the desired rule. Finding rule overlaps is not the only source of complexity. Assume we have rules A, B, and C that match a given set of packets, and the integer priority we assign to them is such that $P_A > P_B > P_C$ (*i.e.* A has higher priority than B, etc.) If we want to break rule B into two new rules B1 and B2 with $P_{B1} > P_{B2}$, then we need to have $P_A > 1+P_C$. Otherwise, we have to change the priority of A and C, and potentially a series of other rules. We might even reach a state that it is not possible to assign priorities anymore.

Rule Deletion. Similar to the priority assignment problem during new flow installation, handling flow removals (either because it is expired, evicted due to table becoming full, or as requested by the controller) is not simple in the presence of wildcard rules. When a flow entry is removed, the packets that were matched to it may now match (potentially several) other rules. In addition to the complexity of identifying such rules, this may require a large number of new flow installations or updates to resolve conflicts.

Large Search Space. The third problem arises when the controller needs to break down a wildcard rule. The search space is large since there are different header fields that we can add to the match list, leading to reduced efficiency of the process.

2.2 Loss of Visibility

The second challenge of using wildcard rules is losing visibility: the controller sees the packets matching any wildcard rule as one flow, and thus has limited (or no) knowledge of the individual flows that are aggregated. In reactive routing, each new flow in the network will trigger a *Packet-In* message, which informs the controller of its existence. Furthermore, when a flow is finished, the associated flow entry in the switch will expire after the defined timeout, which triggers another *Flow-Removed* message to the controller. As a result, the controller is aware of active flows in the network, which is essential for many network monitoring and management applications. The controller visibility over active flows is lost when it uses wildcard rules.

2.3 Hardware Challenges

It is easier, cheaper, and faster to implement exact match rules in comparison to wildcard rules. Dealing with wildcard rules requires specific and complex hardware, such as TCAM, whereas exact match rules can be implemented through hash tables [1]. As a result, the number of wildcard rules that can be handled by a switch is usually much smaller than the number of exact match rules.

A remedy that is used is to focus on a specific class of wildcard rules. For example, if we limit the rules to only specify the destination MAC address, we can implement them in the MAC lookup hardware table of a layer two switch (instead of TCAM). Similarly, we can restrict rules to source/destination IP address and netmask and deploy them in longest prefix matching hardware tables of layer three switches. However, it is not clear how one can mix and match such solutions with unrestricted wildcard rules, due to complications of dealing with priorities.

3. PROPOSED SOLUTIONS

3.1 Increase Visibility

Two different approaches have been proposed to increase the controller visibility. FleXam [15] is a flexible sampling extension

for OpenFlow that provides selective sampling. It enables the controller to sample packets (stochastically, or deterministically) and to define where sampled packets should be sent. In the context of wildcard rules, the controller can gain significant visibility by appropriately sampling packets from wildcard rules. FlowInsight [19] proposes another approach where switches handle different individual flows matching a wildcard rule separately, allowing the controller to query and receive information about them when needed.

3.2 Reduce Programming Complexity

Handling wildcard rules is a complex task for applications. The general idea behind proposed solutions in this area is to describe and implement control applications in a higher level of abstraction, and then to automatically generate the necessary rules.

In one group of these proposals, control applications are implemented in a high level language and abstraction and then compiled into low level flow entries. A well-known example is NetCore [11], which is a high-level, declarative language to describe packet forwarding. There are other proposed programming languages built on top of NetCore such as Pyretic [12] and Flowlog [13].

Another group of solutions create and maintain an abstract state of the network. Control applications can access the network state, and request changes to the state using a well-defined API that hides the lower-level boilerplates. A well-known example of this approach is Onix [8].

Our proposal in this paper is orthogonal to these attempts. Unlike prior works that try to completely hide wildcard rules from the user, ReWiFlow provides a subset of wildcard rules that are easier to use and manage. ReWiFlow aims to complement those approaches and provide a more comprehensive suite of tools for using wildcard rules. For example, it can be used for security applications (e.g. port scan detection [14]), which are usually difficult to implement in systems that do not allow access to packet level information [8].

Furthermore, restricting wildcard rules will make it easier to create, merge, or split rules automatically, which eases compiler development and enables more efficient rule generation (*e.g.* reduce the number of generated rules, which is now done through a series of heuristics after rule generation [11]). The prefix property of ReWiFlow (which we will define and prove later on) can also be used to make automatic handling of rule deletion easier. This is an essential addition to languages such as NetCore as it gives them the ability to generate temporary wildcard rules (*e.g.* block a host for an hour due to suspicious activity) as opposed to permanent rules, which is what they support currently.

3.3 Increase Matching Speed

The last group of proposals focus on how to increase the speed of finding the rule matching an incoming packet. Directly comparing with wildcard rules requires the use of TCAMs, which are both expensive and power hungry. This limits the number of wildcard rules that can be stored in a switch. On the other hand, it is more efficient to search among exact match entries, and BCAMs or SRAMs can be used instead of TCAMs [18]. So the general idea to speed up matching is creating a list of exact match rule instance of a wildcard rule each time a new flow matching that rule is observed, and always search in the exact match field first. This idea is used for both hardware switches [18] as well as software switches such as Open vSwitch. However, this approach introduces new complexities in switch implementation: after each change such as a new rule addition or deletion, the exact match table should be checked and updated for any possible conflict.

MegaFlows is another similar concept introduced in Open vSwitch 1.11. The main goal of MegaFlows is supporting wildcard matching in the kernel module (in contrast to only performing exact matches). The kernel module has the power to restrict wildcard rule by changing some of the wildcard fields to the value of the packet that it matches. This adds more complexity to the switch implementation: the user space program controlling the kernel module is expected to ensure that each incoming packet matches at most one exact or wildcard rule. Furthermore, the user space program should also keep track of how the kernel module restricts installed wildcard rules. These clearly add more programming complexity, even though they might lead to an increase in speed.

4. REWIFLOW

ReWiFlow is a restricted class of OpenFlow wildcard rules, where there is a defined ordering among all possible header fields that can appear in a wildcard rule. This restriction does not require any changes to the OpenFlow specification: ReWiFlow rules are valid OpenFlow rules that can be accepted by any OpenFlow compatible switch and controller.

At the same time, this restriction makes it easier to design algorithms using wildcard rules, without significantly limiting the flexibility. It can also be implemented more efficiently, as we will discuss later on.

4.1 Definition

Before delving into ReWiFlow, let us formally define OpenFlow rules. Assume that H is the set of all header fields $h_1, h_2, \ldots, h_n$ that can appear in a flow rule (h_1 can be the source IP, h_2 the source port, etc.). A flow match rule f is represented as:

$$f = \{(h_j, v_j) \mid h_j \in H\},$$

where each pair (h_j, v_j) means that header field h_j should be equal to the value of v_j (we will discuss mask support later).

The main idea behind ReWiFlow is to have a strict ordering function among all possible header fields. Assume that m is such an ordering. Each of numbers $1..n$ are assigned exactly to one h_j:

$$m: H \rightarrow [n] \mid m(h_i) = m(h_j) \leftrightarrow h_i = h_j.$$

We say a wildcard rule f_i is a ReWiFlow rule with mapping m if:

$$\forall\ (h_j, v_j) \in f\ \&\ \forall\ h_i \in \mathrm{H} \mid m(h_i) < m(h_j) \rightarrow \exists v_i \mid (h_i, v_i) \in \mathrm{f}.$$

In other words, f is a ReWiFlow rule with mapping m if for every header field h_j that is present in f, all header fields that are before h_j in the ordering of m are also present.

Assume that the ordering is $m(h_i)=i$, *i.e.* h_i is the i-th header field in the ordering. If h_i is present, then $h_1, h_2, \ldots, h_{i-1}$ will also be present, and we can represent f as:

$$f = [(h_1,v_1), (h_2,v_2), \ldots, (h_i,v_i)].$$

In other words, a ReWiFlow rule will only have header field h_1, $h_2, \ldots, h_i$ for a known i. This property is very similar to prefix

property of netmasks in IP addresses, *i.e.* a predefined set of all possible header fields can appear in a ReWiFlow rule.

Example. Assume that we have 7 different header fields: VLAN ID, Source IP, Destination IP, Source TCP Port, Destination TCP Port, Source UDP Port, and Destination UDP Port. A possible ordering is: VLAN ID, Destination IP, Destination TCP Port, Destination UDP Port, Source IP, Source TCP Port, and Source UDP Port. In this ordering, VLAN ID must be present in all ReWiFlows, because it comes before all other fields. The simplest rules will match traffic with a specific VLAN tag. We can restrict such rules by limiting them to those designated to a specific IP. We can continue this by requiring a specific destination port (either TCP or UDP), and so on.

Masked Header Fields. OpenFlow allows some header fields to be masked. In ReWiFlow we can break a field into subfields to allow masks. For example, to support CIDR masks for the IP destination field, we can split that field into multiple parts such that each valid mask that we want to support will set the first few parts and leave the rest to be wildcard. For instance, if we want to support netmasks /8, /20 and /32, we can split the destination IP field into 3 parts of length 8, 12 and 12. The /8 mask will have the first part as exact match and the rest as wildcard and /24 will have the first two parts as exact match and the fourth part as wildcard. These sub-fields will be treated like any other field, *i.e.* there is no need that all of these fields be adjacent in the ordering.

4.2 Advantages

To show how ReWiFlow can simplify programming using wildcard rules, let us prove a special characteristic of ReWiFlow rules first: the *Prefix Property*.

Theorem 1. If a packet matches two ReWiFlow rules f_i and f_j, then one of these rules is a subset of the other one, *i.e.* one of them only matches a subset of flows that are matched by the other one. Considering the set definition for rules, this can be represented as $f_i \subset f_j$ or $f_j \subset f_i$.

Proof. Without loss of generality and to simplify the proof, let us assume that $m(h_i) = i$, *i.e.* the ordering among header fields is $h_1 < h_2 < \ldots < h_n$. Also, assume that if a flow rule f requires header field h to be v, we represent that as $f(h) = v$.

Assume that $f_i \not\subset f_j$ and $f_j \not\subset f_i$, and $a = |f_i| \leq |f_j| = b$, *i.e.* f_i is the smaller flow based on the number of header fields that are present. Based on the definition of ReWiFlow rules, both rules f_i and f_j check the first a header fields $h_1, h_2, \ldots, h_a$. If $f_i(h_l) = f_j(h_l)$ for all of these a fields, then f_j will only match a subset of packets that match f_i, because every packet that match f_j will also match f_i. So $f_i \subset f_j$ which contradicts the assumption of $f_i \not\subset f_j$.

If there is a mismatch between the two rules among these header fields, then there is a header field l that $f_i(h_l) \neq f_j(h_l)$. This is a contradiction since the field h_l in the packet that matches both of these rules is v_l, or $f_i(h_l) = v_l = f_j(h_l)$. This completes the proof. ■

4.2.1 Rule Priorities

The priority of ReWiFlow rules can be easily set to their length, *i.e.* the number of header fields in them. Based on Theorem 1, if two rules f_i and f_j are overlapping, then one of them is matching a subset of the other and will be longer, hence it should have a higher priority (otherwise the longer rule with lower priority will match no packet). This is similar to longest prefix matching in IP network routing: the longest matching rule will be selected.

4.2.2 Removed Rules

It is easy to find which of the remaining rules will match packets of a deleted rule. All we need to do is to sort the present header fields based on their ReWiFlow order, and remove the header fields from the highest one in the order one by one. We continue until we reach another rule that is present. This is the rule that will match packets belonging to the deleted rule. If the set becomes empty, there is no match.

It is clear that in each step, we are generalizing the rule (matching one less field), so the final rule will match all such packets. Let us assume that with this process, we create a rule f_j from the removed rule f_i, but the rule that will match those packets is f_k. Based on Theorem 1, we can see that $f_k \subset f_i$ and $f_j \subset f_i$, since f_i should have higher priority than the other two, which means that it is longer than both of them. Theorem 1 also shows that either $f_k \subset f_j$ or $f_j \subset f_k$. We cannot have $f_j \subset f_k$ because in this case we should have stopped when the rule set was equal to f_k, contradicting that f_j was the first time that we reach one of the remaining rules. It also cannot be the case that $f_k \subset f_j$ since in this case f_j will have a higher priority (because it is longer), contradicting that f_k is the new matching rule.

We also note that in ReWiFlow all packets of a removed flow will match at most one of the remaining rules. The prefix property allows the controller to store all installed rules in a trie data structure, which provides fast search to find the new rule that will match the given packet. It also enables a quick way to find which currently installed rules will be affected when a new rule is installed (*e.g.* to check whether the new routing optimization rule may interfere with a currently installed access control rule).

4.2.3 Simple Search Space

Rule breakdown, which is equivalent to searching for new sub-rules, is very simple in ReWiFlow: the pre-specified ordering of header fields defines which new field should be added to the wildcard rule, and only its matching value should be searched.

4.2.4 Hardware Implementation

ReWiFlow converts the general wildcard search into a simple prefix search: if we reorder the header fields based on the ReWiFlow ordering, then finding the matching rule is equivalent to finding the longest prefix matching in this reordered header. Considering that IP routing and longest prefix matching is a well-studied topic, it enables us to use a wide array of techniques and algorithms such as special Bloom filters [9] and tries data structures [16] that could significantly improve the performance of hardware versus using plain TCAM for general wildcard matching.

The strict ordering in ReWiFlow may enforce matching of header fields that are not present in a rule. For example, if UDP port is after TCP port, then to match UDP port we should also have TCP port present in the rule. A simple solution to this problem is to set the value of non-present header fields to 0 when the switch has to match the field in a rule. In the TCP and UDP port example, we can have the TCP port equal to 0 and UDP port equal to 53 in the rule. This solution does not add any limitations to the system: packets that should be matched are still matched, and those packets that should not be matched will not. This simple

technique allows us to take advantage of the prefix property: a switch can create a fixed size bit string from each packet header during the parsing phase (by using the ordering of *m*, values of present fields, and zero for absent fields).

Note that, if the controller needs to install ReWiFlow rules in current OpenFlow switches, it can simply remove the header field parts that cannot appear in any of targeted packets (e.g. destination TCP port in UDP packets).

Another advantage of the strict fixed ordering to specify header field values is that if a considerable number of flows have a specific length, then searching among them would be similar to exact match if we only consider those header fields. As a result, a hash table can be used to search among them. In other words, we may have a number of hash tables that are used to search for rules of different common length. If no matching rules are found, then a general table with varying length (*i.e.* priority) is searched for matching rules[1]. This idea is similar to tuple space search classification (using hash value of header fields to find matching wildcard rule) [17] that is currently used in Open vSwitch, which is not possible to do with general wildcard rules in hardware.

4.3 Multi-ReWiFlow

Although the prefix property will ease working with wildcard rules, the necessity of selecting matching value for previous header fields in order to match a specific header field may inflate the number of rules that should be installed, or make it impossible to do that proactively. For example, if a group of rules only needs to match source or destination ports (*e.g.* to block SMTP traffic), while another group only needs to match source or destination IP (*e.g.* block a specific target machine), then no ordering can accommodate both groups of rules.

To solve this problem, we propose Multi-ReWiFlow, a generalization of ReWiFlow where we can have multiple groups of ReWiFlow rules each with a different orderings. The intuition behind Multi-ReWiFlow is that while we may not be able to use a single ordering to match all rules, we can match the majority of network rules with only a few groups of ReWiFlow rules. For example, we can have a group of rules ordered based on destination IP, and destination port, and another group of rules based on source IP, and source port. There is an optional group of rules that have no explicit ordering (which we call *exceptional set*), *i.e.* rules that do not fit in of the ordered sets. This can reduce the number of ReWiFlow sets in cases that we need multiple new orderings to match a few rules only.

To find the matching rule in the Multi-ReWiFlow, we first find matching rules in each of ReWiFlow rule sets (and possibly in the exceptional set), and then select the rule with highest priority among them. Finding the match in those groups can be done in parallel, which makes the search time independent of the number of rule sets. Assuming the exceptional set of rules is also small, we can have the simplicity and speed of ReWiFlow, without the loss of generality.

[1] Note that there are some implementation details that we skip, such as searching among those tables from longer prefix to shorter to obey priorities; and only put a rule in those exact match tables if it has no sub-rule or all of its sub-rules are also in some exact match table.

The tuple search space [17] approach that is currently used in Open vSwitch can be seen as a special case of Multi-ReWiFlow where we create a set for each combination of determined header fields. For example, if a group of rules matches on source IP only, and another set matches on source IP and destination IP, then Open vSwitch will create two tuple groups, one for source IP only and another one for source IP and destination IP. This can be done in Multi-ReWiFlow by creating two sets of rules with identical ordering of source IP and then destination IP, and by putting rules with both source IP and destination IP filled only in the second set (even though they could be inserted in the first set as well). As a result, the equivalent Multi-ReWiFlow will have fewer number of sets in comparison to the equivalent tuple search approach, as it can merge tuples that are a subset of each other together. This can reduce the number of different tables that we need to implement.

5. EXAMPLE APPLICATIONS

In this section, we demonstrate how we can use ReWiFlow and Multi-ReWiFlow with two example applications. First, we present how ReWiFlow can be used to implement a dynamic proactive routing scheme. A system like this can reap the benefits of both proactive and reactive routing, *i.e.* low setup delay, high visibility and fine-grained control. Then, we show a preliminary set of results that show how Muti-ReWiFlow can be used to represent the access control rules of an operational network [20].

5.1 Proactive Routing

We consider a network that starts with a minimal set of rules to route packets among hosts. Over time, the controller monitors switch port loads, and whenever there is a significant imbalance, it tries to either move some wildcard rules from one port to another, or break down a wildcard rule by creating a sub-flow from that rule and move that to the other port. To create a sub-flow from a wildcard rule, we simply add the next header field based on the selected ReWiFlow ordering, and select the best value for that field to balance the load. The controller receives sampled packets from the switches and buffers them for a period of 1 second to be able to break down wildcard rules. This is an example of applications that are not a good match for previous solutions to reduce programming complexity, because state-based solutions such as Onix are not designed to react to rapid changes in traffic characteristics [8] and high-level languages like NetCore are well suited for permanent changes and invariance [11].

For this experiment, we used the Mininet network simulator, with a modified Open vSwitch and NOX controller that supports FleXam [14]. We also used the common fat-tree topology (similar to what used in DevoFlow [4] and FleXam [14]). Due to the limitations of Mininet (e.g. 2Gbps total bandwidth), we performed our simulation for a network with 16 hosts and 20 switches. The workload was created based on flow characteristics of data center networks reported by Kandula et al. [7] and MGEN toolkit (http://cs.itd.nrl.navy.mil/work/mgen/) was used for traffic generation. We simulated different network loads by multiplying all flow inter-arrival times by a constant factor. We used four different ReWiFlow orderings: (src ip, dst port, dst ip, src port), (dst ip, src ip, dst port, src port), (src port, dst ip, src ip, dst port), and (dst port, src ip, dst ip, src port).

Our preliminary results here are very promising: using wildcard ReWiFlow rules eliminates the flow setup time, which translates to reduced flow completion time (*e.g.* 75% reduction for small flows (<10KB) in high load scenario). It also reduces the table

occupancy and controller load (*e.g.* ~1K installed rules with ReWiFlow vs. ~20K installed rules in reactive routing).

The difference between different ordering of header fields is more prominent in higher loads. For example, the number of rules installed by first ordering is less than 1/3 of third ordering in extreme load scenario. The length of installed ReWiFlow rules also depends on the selected ordering. E.g., more than 95% of installed rules in third and fourth ordering have length 1, because the first header field is source or destination port in those ordering. On the other hand, more than 85% of installed rules for the first ordering have length 2 under high load scenario.

5.2 Multi-ReWiFlow for Access Control Rules

As an example of how Multi-ReWiFlow can be used, we demonstrate how access control rules (*i.e.* ACLs) of Stanford backbone networks [20] can be represented by Multi-ReWiFlow. There are a total of 1636 configured ACLs in Stanford backbone routers. In each ACL, there is an action (permit/deny) and 5 header fields that can be determined to match packets: source IP, destination IP, protocol, source port, and destination port. Different rules define different subset of fields. For example, 160 rules defined the protocol and destination port, 136 rules only defined the /32 source IP, and 156 rules defined the /32 source IP and /27 destination IP.

To store these rules as a Multi-ReWiFlow, we tried various orderings and found a group of 4 ReWiFlow rule sets that cover 85% of rules: The first ordering is [source IP (/0, /8, /12/, /14, /16, /21, /22, /23, /24, /25, /26, /27, /29, /30, /32)[2], destination IP (/0, /16, /24, /27, /31, /32), protocol, destination port, source port], which contains 708 ACLs (43% of all ACLs). The second ordering is [destination IP, protocol, destination port, source IP, source port] which contains 310 of remaining ACLs (19% of all ACLs). The third ordering is [protocol, destination port, source IP, destination IP, source port] which contains 289 of remaining ACLs (18% of all ACLs). The fourth ordering is [source IP (/22), destination IP (/23, /32), source IP (/23, /32), protocol, source port, destination IP, destination port] which contains 81 of remaining ACLs (5% of all ACLs). A fifth exceptional set will cover the remaining 249 rules (15% of all ACLs).

The interesting observation about these ordering is that they also provide insight about what rules in each category do: the first set focuses on the communication between specific pairs of machines, the second set is about restricting access to specific machines, and the third one on specific types of traffic. As a comparison, if we wanted to use the tuple search space method, we need to have 13 different tables because there are 13 different combinations of header fields in these rules, which is four times higher than the number of sets for Multi-ReWiFlow.

6. CONCLUSION

In this paper, we proposed ReWiFlow as a restricted class of OpenFlow wildcard rules that can reduce the programming complexity of wildcard rules. We also presented Multi-ReWiFlow, as a more flexible generalization of ReWiFlow. Our preliminary experiments show that we can use ReWiFlow to implement a dynamic proactive routing to eliminate the flow setup time and improve the overall flow completion time. We have also promising results showing how Multi-ReWiFlow can be used to represent ACLs of an operational network. An important part of using ReWiFlow and Multi-ReWiFlow is selecting header field orderings. We plan to develop automatic methods to extract such orderings based on application needs, network characteristics, and traffic.

[2] The numbers inside parenthesis represent how this field is broken down into multiple parts to support masks.

7. REFERENCES

[1] M. Appelman. 2012. Performance Analysis of OpenFlow Hardware. M.Sc. Thesis. University of Amsterdam.

[2] T. Benson, et al. 2010. Network traffic characteristics of data centers in the wild. In IMC '10. 267-280.

[3] R. Bifulco, and F. Schneider. 2013. OpenFlow rules interactions: definition and detection. In SDN4FNS 2013.

[4] A.R. Curtis, et al. 2011. DevoFlow: scaling flow management for high-performance networks. In SIGCOMM'11. 254-265.

[5] H. Hamed and E. Al-Shaer. 2006. Taxonomy of conflicts in network security policies. Comm. Mag. 44(3). 134-141.

[6] C.Y. Hong, et al. 2012. Finishing flows quickly with preemptive scheduling. In SIGCOMM '12. 127-138.

[7] S. Kandula, S., et al. 2009. The nature of data center traffic: measurements & analysis. In IMC'09. 202-208.

[8] T. Koponen, et al. 2010. Onix: a distributed control platform for large-scale production networks. In OSDI'10.

[9] H. Lim, et al. 2014. On Adding Bloom Filters to Longest Prefix Matching Algorithms. IEEE Trans. Compu. 63(2). 411-423.

[10] F. Mansmann, et al. 2012. Visual analysis of complex firewall configurations. In VizSec '12.

[11] C. Monsanto, et al. 2012. A compiler and run-time system for network programming languages. In POPL '12. 217-230.

[12] C. Monsanto, et al. 2013. Composing software-defined networks. In NSDI'13.

[13] T. Nelson, et al. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In NSDI'14. 519-531.

[14] S. Shirali-Shahreza and Y. Ganjali. 2013. Efficient Implementation of Security Applications in OpenFlow Controller with FleXam. In HotI 2013. 49-54.

[15] S. Shirali-Shahreza and Y. Ganjali. 2013. FleXam: Flexible Sampling Extension for Monitoring and Security Applications in OpenFlow. In HotSDN 2013.

[16] H. Song, et al. 2012. Efficient trie braiding in scalable virtual routers. IEEE/ACM Trans. Netw. 20(5). 1489-1500.

[17] V. Srinivasan, et al. 1999. Packet classification using tuple space search. SIGCOMM Comput. Commun. Rev. 29(4). 135-146.

[18] H. Yamanaka, et al. 2014. OpenFlow Networks with Limited L2 Functionality. In ICN2014. 221-229.

[19] G Yao, et al. 2014. FlowInsight: Separating Visibility and Operability in SDN Data Plane. In ONS 2014.

[20] H. Zeng, et al. 2012. Automatic Test Packet Generation. In CoNEXT 2012.