Packet Traceback for Software-Defined Networks

Harvest Zhang
Princeton University
hlzhang@princeton.edu

Joshua Reich
Princeton University
jreich@cs.princeton.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

Packet traceback—determining how a packet could have arrived at a point of observation—is useful for network debugging, performance testing, and network forensics. However, existing mechanisms (e.g., NetSight) require modifications to switches and introduce additional network overhead. By providing a centralized representation of the network's packet-processing behavior as a *policy*, Software-Defined Networking (SDN) makes it possible to compute the transformations that could lead to the observed packet. Our work leverages higher-level SDN controller languages to perform packet traceback in a provably-correct manner *entirely on the controller*. Using the current policy as input, we precompute a compact symbolic representation of the *back policy*, which can then quickly produce all possible predecessors for any input packet. Our prototype is implemented in the Pyretic language; however, since any policy specified in low-level OpenFlow rules can be easily converted to a Pyretic representation, our method is completely general.

1. INTRODUCTION

The goal of packet traceback is to determine how a packet reached its current location, including the path through the network and any modifications of the packet en route. Packet traceback has various practical applications, ranging from network security to performance monitoring and network debugging. For example, a denial-of-service attack might be first detected, then traced back, and finally blocked at its entry points. Packet traceback may also help diagnose performance problems; if network operators discover that certain traffic flows have poor performance, a traceback can identify which links to check for congestion. Finally, knowing the potential paths taken by a packet is useful for debugging, by determining how behavior apparently deviating from operator intent (e.g., appearance of a packet at an unexpected location) could have arisen.

In traditional networks, distributed routing protocols and local packet-forwarding decisions make it difficult to determine the path a packet traverses. Active probing tools like traceroute can measure the forwarding path, but only with knowledge of—and control over—the sending host to launch the probes. Traceroute also only works correctly if the switches treat probes the same way as regular packets *and* the path has not changed in the meantime. Moreover, probe traffic adds load to an already taxed network, and may interfere with other network services (e.g., IDS, load balancers).

Alternately, the underlying switches could offer support for packet tracing. The IP "record route" option [1] instructs each node in the path to add its identifier to the packet, so the receiver can learn the end-to-end path. However, network operators almost always disable this feature to reduce overhead—and to prevent packets from becoming too large. In more recent work, NetSight [2] proactively collects history metadata for all packets and sends the information to a collector for analysis. However, NetSight requires changes to the switches (to generate reports) and incurs additional overhead to monitor each packet at each hop in its path.

In a Software-Defined Network (SDN), the logically centralized controller has complete knowledge of the current network policy. This enables a wide range of techniques that can analyze the policy at the controller, rather than monitoring in the data plane. Given a static policy, several existing tools can project how a packet at a given location would progress through the network, and check whether the policy violates certain network invariants (e.g., testing for loops or blackholes) [3, 4, 5]. While clearly very useful for tracing packets *forward* from a known starting point, these techniques are not efficient for tracing *backwards* to understand how a packet might have reached its current location.

A small traceback example.

Figure 1 shows three example policies that illustrate the challenges of packet traceback. In all three cases, we have a packet x whose source IP's first bit is 1, leaving switch a via port 1, and we want to trace its path back through the network.

Policy 1: Packets entering switch d on port 3 are forwarded to b if the first bit of the source IP address is 1, and to c otherwise. Switches b and c similarly forward these packets to a. For example, if c receives a packet

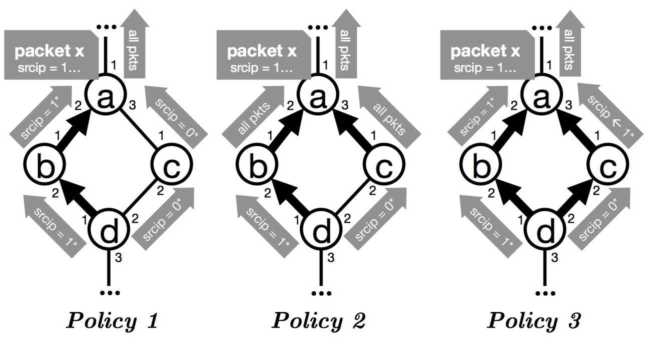

Figure 1: Three policies on the same small topology. Switches are labeled a-d, with links labeled with port numbers. Bold arrows indicate the links that packet traceback should explore when tracing packet x back from switch a, and the grey arrows indicate the policy.

with an IP whose first bit is 1, that packet would be dropped. Thus, we can determine that packet x could not have come from c, and must have traversed the path $d \to b \to a$.

Policy 2: Instead of matching on IP prefixes, switches b and c forward *all* packets to a. Thus, when tracing x back from a we must consider the possibility that x could have come from either b or c. Only when we trace back another hop do we discover that the path must be $d \to b \to a$, since x could not have reached c from d.

Policy 3: With packet modification, we must make sure to reverse any changes to the packet as we trace back. Policy 3 is identical to policy 1, except that switch c now changes the first bit of each packet's source IP to 1 before forwarding them to a. As a result, we are left with two possibilities: if the packet originally entered d with the first source IP bit equal to 0, then the path was $d \to c \to a$ with modification at c, but if the packet entered d with a source IP whose first bit was 1, the path was $d \to b \to a$. Only by tracing further back can we potentially disambiguate between these possibilities[1].

These examples illustrate the challenges of performing traceback on arbitrary policies. We must allow for *multiple* possible predecessors for a packet, may need to trace back multiple hops to resolve ambiguities, and must consider the effects of packet modifications. Performing these operations from scratch on each observed packet would be computationally expensive. Instead, given a policy, we want to compute a *symbolic* representation of the *back policy* so we can quickly perform a traceback for any concrete packet. In addition, a symbolic representation would enable a wide range of analysis of the back policy (e.g., to check certain invariants).

Computing the back policy requires an effective way to "invert" the original policy. In this paper, we present an efficient algorithm for computing the back policy. We start with OpenFlow policies written in the NetCore [6, 7] domain-specific language—a high-level language that specifies policies using boolean predicates and simple composition operators instead of low-level OpenFlow rules. A runtime system (e.g., Pyretic [7]) can compile these higher-level policies into OpenFlow rules to install on the switches. Similarly, given any collection of OpenFlow rules, a trivial linear-time algorithm can generate an equivalent NetCore policy, making NetCore an appealing starting point for our traceback work.

After a brief overview of NetCore in Section 2, we show how to transform a policy into a provably correct back policy in Section 3. This back policy, when applied to a packet at a given location, would produce the set containing all possible immediate predecessor packets. Applied iteratively, we can obtain all predecessor sequences that could have resulted in the observed packet. Section 4 describes our implementation of packet traceback in Pyretic. While a computed back policy is correct, its representation may not be *compact*. Our implementation applies simplification techniques to transform a back policy into an equivalent—yet smaller—representation that enables more efficient traceback. Preliminary experiences with our prototype demonstrate that efficient packet traceback is a valuable tool for SDN debugging and management.

2. BACKGROUND ON NETCORE

A NetCore policy is a function from a located packet (i.e., a packet and its current switch and port) to a *set* of located packets [6]. An empty set corresponds to a dropped packet, a single packet at a new location corresponds to unicast forwarding, and multiple packets corresponds to multicast. Figure 2 summarizes the key elements of NetCore [6, 7], following the syntax of [8].

Fields	f	$::=$	$f_1 \mid \cdots \mid f_k$	
Filters	A, B	$::=$	$[1]$	*Identity*
		$\mid$	$[0]$	*Drop*
		$\mid$	$[f = v]$	*Match*
		$\mid$	$A \cdot B$	*And*
		$\mid$	$A + B$	*Or*
		$\mid$	$\neg A$	*Negation*
Policies	M, N	$::=$	A	*Filter*
		$\mid$	$(f \leftarrow v)$	*Modify*
		$\mid$	$M \cdot N$	*Sequential*
		$\mid$	$M + N$	*Parallel*

Figure 2: NetCore syntax.

The building blocks in Figure 2 can be used to construct sophisticated network policies [7, 8, 9]. We briefly describe the key elements:

[1]E.g., if switch d can only receive packets whose first source IP bit is 0, then we know the path is $d \to c \to a$.

Filter: A policy $[q]$ passes through all packets that match a predicate q and drops all other packets. For example, drop (which always outputs the empty set) is simply $[0]$, while $[f = v]$ passes through only packets whose field f has value v, with the packet unchanged.

Modify: A policy $f \leftarrow v$ changes the value of header field f to value v by outputting a set containing a copy of the input packet whose field f has value v. NetCore represents the packet's location as logical header fields, so packet forwarding is equivalent to modifying the switch and/or port fields.

Sequential composition: $A \cdot B$ takes policies A and B, producing a new policy that first applies policy A to the input packet and then applies B to each located packet in the set output by A, and finally takes the union of these outputs. For example, we might sequentially compose a match and a modify, $[f_1 = v_1] \cdot (f_2 \leftarrow v_2)$, which outputs all packets that originally had a value v_1 for field f_1, but with the value of f_2 changed to v_2.

Parallel composition: $M + N$ takes two policies M and N, producing a new policy that applies M and N to the same input packet, and outputs the set union of their individual outputs. For example, we might compose in parallel two match policies, $[f = v_1] + [f = v_2]$, which outputs any packets for which field f has a value of either v_1 or v_2.

2.1 Network topology as a policy

The network topology can also be represented as a NetCore policy [8], capturing the forwarding of a packet across a link. For example, we can encode the link going from switch b to a in Figure 1 as

$$L_{ba} := [s = b] \cdot [o = 1] \cdot (s \leftarrow a) \cdot (i \leftarrow 2) \cdot (o \leftarrow \varnothing) \qquad (1)$$

so that a located packet x at switch b and outport 1 will have its switch updated to a, its inport updated to 2, and its outport cleared.

A topology T can then be defined as as the parallel composition of all (a, b) in the set of links S, denoted by the summation symbol:

$$T := \sum_{(a,b) \in S} L_{ab} \qquad (2)$$

For a network with policy P and topology T, we can represent the forwarding of packets as the application of P followed by T (i.e., $P \cdot T$) [8].

2.2 Simplifying NetCore policies

NetCore policies satisfy a family of axioms [8] that enable provably correct syntactic transformations. For example, consider the policy $(f \leftarrow v) \cdot [f = v]$, which assigns the packet header field f to the value v and then tests whether the value of f matches v. Since this match is always true, the following equivalence holds

$$(f \leftarrow v) \cdot [f = v] \triangleq (f \leftarrow v)$$

Likewise, a policy that first modifies field f_1 and then immediately modifies a *different* header field f_2 is equal to the policy that makes the same modifications in the opposite order:

$$(f_1 \leftarrow v_1) \cdot (f_2 \leftarrow v_2) \triangleq (f_2 \leftarrow v_2) \cdot (f_1 \leftarrow v_1)$$

In Section 4, we use these kinds of transformations to reduce the complexity of the back policy.

3. THE BACK POLICY

In a slight abuse of notation, we denote policy P's *back policy* as P^{-1}. This notation highlights the intuition behind what a back policy is: P takes a packet and produces an output set, so P^{-1} should take members of that output set and map them back to a set containing that original input packet. Applied iteratively, we can thus generate the predecessors of an observed packet, their predecessors' predecessors, and so on until all possible traces that could have generated the observed packet have been produced.

3.1 Definition of the back policy

Like any other NetCore policy, a back policy maps a located packet to a set of located packets. However, the back policy is *not* intended to be compiled to a set of rules to install on switches. Instead the back policy is evaluated on the controller to determine the set of traces that could have generated an observed packet. Thus, while in a normal policy an output set with multiple packets corresponds to multicast (i.e., a single packet forwarded to multiple locations), the set of output packets in a back policy instead expresses uncertainty (i.e., a packet could have come from several possible locations).

Given a network policy P and arbitrary packets x and y, we define P^{-1} formally as the policy for which

$$y \in P(x) \Leftrightarrow x \in P^{-1}(y) \qquad (3)$$

In words, for any packet y in the set of packets output when policy P is applied to input packet x, it must hold that x is also in the set of packets output when P^{-1} is applied to packet y. The reverse also holds true: for any packet x in the output set of P^{-1} applied to y, the packet y must be in the output set of P applied to x.

Intuitively, the forward direction ($y \in P(x) \Rightarrow x \in P^{-1}(y)$) says that the output set of the back policy is not "too small": the output set of the back policy will contain every packet that, when used as input to the original policy, could have generated the back policy's initial input. The reverse direction ($x \in P^{-1}(y) \Rightarrow y \in P(x)$) says that the output set of back policy is not "too large": there is no packet generated by the back policy that, when used as input to the original policy, will not generate a set containing the back policy's initial input.

3.2 Examples of back policies

To illustrate the back policy, we return to the examples in Figure 1. The first policy in Figure 1 consists of multiple forwarding policies composed in parallel:[2]

$$\begin{aligned} P_1 := &([s=a] \cdot ([i=2] + [i=3]) \cdot (o \leftarrow 1)) && (4)\\ &+ ([s=b] \cdot [srcip=1*] \cdot (o \leftarrow 1)) && (5)\\ &+ ([s=c] \cdot [srcip=0*] \cdot (o \leftarrow 1)) && (6)\\ &+ ([s=d] \cdot [srcip=1*] \cdot (o \leftarrow 1)) && (7)\\ &+ ([s=d] \cdot [srcip=0*] \cdot (o \leftarrow 2)) && (8) \end{aligned}$$

The link policy L_{ba} from b to a is given by Eqn. 1. Thus, a packet x coming into b would be sequentially evaluated first by P_1, which would set x's outport to 1 if the first bit of x's source IP was 1 (and drop x otherwise), and then by L_{ba}, which forwards x out port 1 to switch a.

We have a packet to trace back, x, whose first source IP bit is 1 and which has just left switch a out of port 1. Since we don't know the port on which x entered a, we must trace back through all three ports: i.e., we begin with three copies of x, one at each possible inport.

The back of P_1 is given by

$$\begin{aligned} {P_1}^{-1} \triangleq &([s=a] \cdot ([i=2] + [i=3]) \cdot [o=1] \cdot (o \leftarrow \varnothing)) && (9)\\ &+([s=b] \cdot [srcip=1*] \cdot [o=1] \cdot (o \leftarrow \varnothing)) && (10)\\ &+([s=c] \cdot [srcip=0*] \cdot [o=1] \cdot (o \leftarrow \varnothing)) && (11)\\ &+([s=d] \cdot [srcip=1*] \cdot [o=1] \cdot (o \leftarrow \varnothing)) && (12)\\ &+([s=d] \cdot [srcip=0*] \cdot [o=2] \cdot (o \leftarrow \varnothing)) && (13) \end{aligned}$$

Note that applying P_1^{-1} keeps the two copies of x at ports 2 and 3 but drops the copy of x at port 1 immediately.

We first look at the link (b, a), whose back policy

$${L_{ba}}^{-1} \triangleq [s=a] \cdot [i=2] \cdot (s \leftarrow b) \cdot (o \leftarrow 1) \cdot ((i \leftarrow 1) + (i \leftarrow 2))$$

sends x from a's inport 2 back to switch b's outport 1. Note that L_{ba}^{-1} produces two packets at inports 1 and 2, since this link policy has insufficient information to determine the port on which x entered b. Similarly, L_{ca}^{-1} sends x from a's inport 3 back to c's outport 1, producing two packets with inports 1 and 2.

In the next iteration, we apply ${P_1}^{-1}$ to the two copies of x at inports 1 and 2 of b, where both match (and have their outports stripped); we then apply topology links ${L_{db}}^{-1}$ and ${L_{ab}}^{-1}$, which produces a set of three packets at d's outport 1 and a similar set of three packets at a's outport 2.

However, when we apply ${P_1}^{-1}$ to the two copies of x at inports 1 and 2 of c, the match $[srcip=0*]$ fails since the first bit of x's source IP is 1, and both packets are dropped.

[2]For illustrative purposes, we use a simple policy that drops packets entering switch a on port 1.

Name	**Policy** (P)	**Back policy** (P^{-1})
Filter	$[q]$	$[q]$
Modify	$(f \leftarrow v)$	$[f=v] \cdot \sum_{u \in type(f)} (f \leftarrow u)$
Sequential	$M \cdot N$	$N^{-1} \cdot M^{-1}$
Parallel	$M + N$	$M^{-1} + N^{-1}$

Table 1: NetCore primitives and their back policies. M,N are policies; q is an abstract predicate.

Looking ahead to the next iteration, all the packets at a will be dropped, but the packets at d are not, so we continue tracing back from d. Continuing to iterate, we eventually determine that the only path that x could have traveled was $d \rightarrow b \rightarrow a$.

Packet modification.

The impact of packet modification is illustrated by policy 3 in Figure 1 at switch c, which can be represented as a policy by

$$P_3 := [s=c] \cdot [srcip=0*] \cdot (srcip \leftarrow 1*) \cdot (o \leftarrow 1) \ldots$$

When we compute its back policy P_3^{-1}, we make sure to reverse the modification:

$${P_3}^{-1} \triangleq [s=c] \cdot [srcip = 1*] \cdot [o=1] \cdot (srcip \leftarrow 0) \cdot (o \leftarrow \varnothing) \ldots$$

This ensures that the correct field values for x are being restored as we trace back; otherwise, if we do not undo the change, an incorrect version of x would get dropped at switch d and we would miss a potential path.

3.3 Efficient calculation of the back policy

The compositional structure of NetCore ensures that the back policy as defined above can always be calculated via a straightforward syntactic transformation. Specifically, since any given NetCore policy is recursively built from the four basic components shown in Table 1, we can apply the back for each of these rules recursively to obtain a representation of the corresponding back policy.

In the rest of this section, we prove and explain each case in Table 1. Then, induction over the structure of the syntax of policies proves that our computed back policy satisfies Eqn. 3 for any NetCore policy.

3.3.1 Filter

Informally, the back of any filter is itself. Formally:

$$[q]^{-1} \triangleq [q] \qquad (14)$$

We start our proof with the forward direction of Eqn. 3, replacing both P and P^{-1} with $[q]$, as per Eqn. 14:

$$y \in [q](x) \Rightarrow x \in [q](y) \qquad (15)$$

In other words, for any output packet y produced by $[q](x)$, it must hold that x is produced by $[q](y)$.

We observe that this is always true, since either $[q](x)$ outputs the empty set, in which case there is no packet y and Eqn. 15 is trivially true; or $[q](x)$ outputs the set containing only x, in which case $y = x$ and thus $[q](y)$ also outputs the set containing only x.

The proof for the reverse direction is symmetric. $\square$

3.3.2 Modify

Informally, the back of modifying field f to value v is the policy that first tests that field f contains value v and then generates all possible previous assignments of f. Formally:

$$(f \leftarrow v)^{-1} \triangleq [f = v] \cdot \sum_{u \in type(f)} (f \leftarrow u) \tag{16}$$

For any packet output by $(f \leftarrow v)$, that packet's field f must have value v; the back of modify thus first ensures $([f = v])$. However, without additional knowledge, we do not know what value field f had before, so the back of modify must subsequently produce one packet for every valid value that field f could take (e.g., if f is i, one packet for each inport existing on the switch).

Again, we start our proof with the forward direction of Eqn. 3, substituting for P and P^{-1}

$$y \in (f \leftarrow v)(x) \Rightarrow x \in \left([f = v] \cdot \sum_{u \in type(f)} (f \leftarrow u) \right)(y) \tag{17}$$

If $y \in (f \leftarrow v)(x)$, then we know that field f of y must equal v and that this is the *only* potential way in which y differs from x. Thus we also know x must be in the set of packets identical to y with respect to every field, except possibly f. This is precisely the set output by applying $[f = v] \cdot \sum_{u \in type(f)} (f \leftarrow u)$ to packet y.

In the reverse direction,

$$x \in \left([f = v] \cdot \sum_{u \in type(f)} (f \leftarrow u) \right)(y) \Rightarrow y \in (f \leftarrow v)(x) \tag{18}$$

either y's field f equals v or it does not. In the latter case, Eqn. 18 trivially holds, since there are no packets in the output set. Thus we only need consider when y's field f has value v. In this case, x will be equal to y with respect to all fields, except possibly f. By definition, $(f \leftarrow v)(x)$ will output the single packet equal to y with respect to all fields other than f, and also equal to y on field f: that is, y. $\square$

3.3.3 Sequential composition

The back of the sequential composition of two policies is the reverse sequential composition of their individual backs. Formally:

$$(M \cdot N)^{-1} \triangleq N^{-1} \cdot M^{-1} \tag{19}$$

Intuitively, this mirrors the logic of true mathematical inverses[3] (i.e., for any two invertible functions g and h from integers to integers, $(g \cdot h)^{-1} = h^{-1} \cdot g^{-1}$). The proof of Eqn. 19 is similar in structure and relies on the formal definition of sequential composition

$$(M \cdot N)(x) := \bigcup_{z \in M(x)} N(z) \tag{20}$$

which allows us to prove the forward direction

$$y \in (M \cdot N)(x) \Rightarrow x \in (N^{-1} \cdot M^{-1})(y) \tag{21}$$

If y is in the output set of $(M \cdot N)(x)$, by Eqn. 20 there exists a packet z such that z is in $M(x)$ and y is in $N(z)$. By Eqn. 3, this means z must also be in $N^{-1}(y)$, and similarly x must be in $M^{-1}(z)$, which is precisely the definition of $N^{-1} \cdot M^{-1}$: as seen by substituting N^{-1} for M, M^{-1} for N, and y for x into Eqn. 20

$$(N^{-1} \cdot M^{-1})(y) := \bigcup_{z \in N^{-1}(y)} M^{-1}(z)$$

The proof for the reverse direction is symmetric. $\square$

3.3.4 Parallel composition

Finally, the back of two parallel composed policies is just the parallel composition of their individual backs. Formally:

$$(M + N)^{-1} \triangleq M^{-1} + N^{-1} \tag{22}$$

Intuitively, any packet produced by $M + N$ must have been produced by either M or N. Therefore, we can be certain that the initial packet must be in either the set produced by M^{-1} or N^{-1}; i.e., their parallel composition. Using the definition of parallel composition

$$(M + N)(x) := M(x) \cup N(x) \tag{23}$$

we prove the forward direction

$$y \in (M + N)(x) \Rightarrow y \in (M^{-1} + N^{-1})(y) \tag{24}$$

by noting that if y is in the output set of $(M + N)(x)$, then y must either be in the output set of $M(x)$ or $N(x)$. By the definition of back policy (Eqn. 3), x must either be generated by $M^{-1}(y)$ or $N^{-1}(y)$ to y, so x must also be in the set of packets generated by $(M^{-1} + N^{-1})(y)$. The proof for the reverse direction is symmetric. $\square$

4. PROTOTYPE IMPLEMENTATION

Our implementation of packet traceback runs on the Pyretic platform [7, 9]. The majority of the logic is

[3]To see why back and inverse are not always equivalent, consider the filter [0], which has a well-defined back policy (itself) but no mathematical inverse: i.e., there is no policy f such that $([0] \cdot f)(x) = \{x\}$.

implemented on top of Pyretic, using Pyretic's existing primitives, though we extended the Pyretic runtime system to support (i) representing the topology as a policy and (ii) using summations for parallel composition over a range of values the field being summed over may take. We used Mininet [10] to test packet traceback on various policies and topologies.

The general algorithm for computing a packet traceback consists of two steps. First, given the policy P, we compute the back policy P^{-1} as described in Section 3, once for each time the policy or topology changes. Second, to perform traceback on any concrete located packet x, we iteratively apply the back policy. We first apply P^{-1} to obtain a set of located packets $X = P^{-1}(x)$, which consists of all packets that could lead to x in one step. Then, we apply $P^{-1}(x')$ for each packet $x' \in X$, and so on. Eventually, for each packet, we reach a point where the back policy either (i) drops the packet or (ii) reaches an ingress link at the edge of the network[4]. At termination, we are left with a set of located packets at ingress points. To see how these packets flow through the network, we can iteratively apply P on each packet and record the results.

We compute the back policy using a simple recursive traversal through the policy. Similarly, applying the back policy to a packet is relatively simple, using existing Pyretic mechanisms for policy evaluation. However, evaluating P^{-1} can be extremely slow when the back policy includes a summation over a field with a large range of values. We can reduce computation time substantially by reducing P^{-1} to a simpler form. For example, consider the policy $P := [f = v] \cdot f \leftarrow w$. Recalling from Table 1 that the back policy of $M \cdot N$ is $N^{-1} \cdot M^{-1}$, we we see

$$P^{-1} \triangleq [f = w] \cdot \sum_{u \in type(f)} f \leftarrow u \cdot [f = v]$$

Using the following three axioms [8]:

$$\begin{aligned} (f \leftarrow v) \cdot [f = v] &\triangleq (f \leftarrow v) \\ (f \leftarrow v') \cdot [f = v] &\triangleq [0] \\ M + [0] &\triangleq M \end{aligned}$$

we can simplify this expression to use a single modify

$$\begin{aligned} P^{-1} &\triangleq [f = w] \cdot ((f \leftarrow v) \cdot [f = v] + \sum_{u \neq v} (f \leftarrow u) \cdot [f = v]) \\ &\triangleq [f = w] \cdot ((f \leftarrow v) + \sum_{u \neq v} [0]) \\ &\triangleq [f = w] \cdot (f \leftarrow v) \end{aligned}$$

This makes sense: the original policy matches packets where field f has value v and then changes f to w, whereas the back policy matches packets where field f has value w and then changes f back to v.

[4]If P has a forwarding loop, this process might not terminate; a safe implementation must either use loop-detection or require a loop-free policy.

This simplification easily extends to few-to-one mappings, such as

$$([f = u] + [f = v]) \cdot (f \leftarrow w)$$

whose simplified back policy is

$$[f = w] \cdot ((f \leftarrow u) + (f \leftarrow v))$$

As expected, if several different values of f are both mapped to value w, the back policy will output a set of packets containing a packet with each possible predecessor value of f in order to capture this ambiguity. These and other simplifications typically allow us to produce a simpler—yet equivalent—representation of P^{-1}, to reduce the computational overhead of applying the back policy to concrete packets.

5. CONCLUSION

In this paper, we design and implement a solution for packet traceback computed entirely by the controller—without introducing any data-plane overhead. The core of this algorithm is the computation of a "back policy" using a provably correct set of syntactic transformations on a given policy. This back policy can then be iteratively applied to packets of interest to determine all possible paths the packets could have taken through the network. Furthermore, this algorithm is generalizable to any SDN implementation, since any lower-level policy (such as OpenFlow rule tables) can be converted into a NetCore policy.

In our ongoing work, we are exploring three main issues. First, we are designing a general way to resolve ambiguous tracebacks—where multiple traces could lead to the observed packet. When the traceback would be ambiguous, we can extend the policy to tag packets (e.g., using a VLAN header) to ensure the traceback algorithm can disambiguate between multiple traces. Second, while the back policy applies to symbolic packets, the iterative application of the back policy relies on having a *concrete* packet. We are exploring techniques for *symbolic* traceback so we can precompute the iterative application of the back policy. Third, we are investigating extensions of our traceback techniques to work across administrative domains, by designing a protocol where one autonomous system could "hand off" a partially complete traceback to the next upstream autonomous system(s). Together, these extensions would offer a scalable, end-to-end solution for packet traceback.

6. REFERENCES

[1] "Internet protocol: DARPA Internet program protocol specification," Sept. 1981. RFC 791.

[2] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown, “I know what your packet did last hop: using packet histories to troubleshoot networks,” in *NSDI*, Apr. 2014.

[3] P. Kazemian, G. Varghese, and N. McKeown, “Header space analysis: Static checking for networks,” in *NSDI*, 2012.

[4] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real time network policy checking using header space analysis,” in *NSDI*, 2013.

[5] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, “VeriFlow: Verifying network-wide invariants in real time,” Apr. 2013.

[6] C. Monsanto, N. Foster, R. Harrison, and D. Walker, “A compiler and run-time system for network programming languages,” in *Principles of Programming Languages*, vol. 47, pp. 217–230, Jan. 2012.

[7] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing software defined networks,” in *NSDI*, Apr. 2013.

[8] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, “NetKAT: Semantic foundations for networks,” in *Principles of Programming Languages*, pp. 113–126, Jan. 2014.

[9] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular SDN Programming with Pyretic,” *USENIX ;login*, vol. 38, pp. 40–47, October 2013.

[10] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, “Reproducible network experiments using container-based emulation,” in *ACM CoNEXT*, pp. 253–264, Dec. 2012.