Diagnosing Missing Events in Distributed Systems with Negative Provenance

Yang Wu
University of Pennsylvania

Mingchen Zhao
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Wenchao Zhou
Georgetown University

Boon Thau Loo
University of Pennsylvania

ABSTRACT

When debugging a distributed system, it is sometimes necessary to explain the *absence* of an event – for instance, why a certain route is *not* available, or why a certain packet did *not* arrive. Existing debuggers offer some support for explaining the *presence* of events, usually by providing the equivalent of a backtrace in conventional debuggers, but they are not very good at answering "Why not?" questions: there is simply no starting point for a possible backtrace. In this paper, we show that the concept of *negative provenance* can be used to explain the absence of events in distributed systems. Negative provenance relies on counterfactual reasoning to identify the conditions under which the missing event *could have* occurred. We define a formal model of negative provenance for distributed systems, and we present the design of a system called Y! that tracks both positive and negative provenance and can use them to answer diagnostic queries. We describe how we have used Y! to debug several realistic problems in two application domains: software-defined networks and BGP interdomain routing. Results from our experimental evaluation show that the overhead of Y! is moderate.

Categories and Subject Descriptors

C.2.3 [**Network operations**]: Network management; D.2.5 [**Testing and debugging**]: Diagnostics

Keywords

Diagnostics, Debugging, Provenance

1. INTRODUCTION

Finding problems in complex distributed systems has always been challenging, as the substantial literature on network debugging and root-cause analysis tools [5, 9, 12, 13, 18] can attest. The advent of software-defined networking (SDN) has added a new dimension to the problem: forwarding can now be controlled by programs, and, like all other programs, these programs can have bugs. Finding such bugs can be difficult because, in a complex network of hosts, routers, switches, and middleboxes, they can manifest in subtle ways that have no obvious connection with the root cause. It would be useful to have a "network debugger" that can assist the network operator with this task, but existing techniques tend to be protocol-specific and cannot necessarily be applied to arbitrary distributed applications, or SDNs with arbitrary control programs. Hence, as others [8, 9] have observed, a more powerful debugger is needed.

Existing solutions, such as NetSight [9] or SNP [34], approach this problem by offering a kind of "backtrace", analogous to a stack trace in a conventional debugger, that links an observed effect of a bug to its root causes. For instance, suppose the administrator notices that a server is receiving requests that should be handled by another server. The administrator can then trace the requests to the last-hop switch, where she might find a faulty flow entry; she can trace the faulty entry to a statement in the SDN controller program that was triggered by a certain condition; she can trace the condition to a packet from another switch; and she can continue to recursively explain each effect by its direct causes until she reaches a set of root causes. The result is the desired "backtrace": a causal chain of events that explains how the observed event came to pass. We refer to this as the *provenance* [1] of the event.

Provenance can be a useful tool for debugging complex interactions, but there are cases that it cannot handle. For instance, suppose that the administrator observes that a certain server is *no longer receiving any requests* of a particular type. The key difference to the earlier scenario is that the observed symptom is not a positive event, such as the arrival of a packet, that could serve as a "lead" and point the administrator towards the root cause. Rather, the observed symptom is a negative event: the *absence* of packets of a certain type. Negative events can be difficult to debug: provenance does not help, and even a manual investigation can be difficult if the administrator does not know where the missing packets would normally come from, or how they would be generated.

Nevertheless, it is possible to construct a similar "backtrace" for negative events, using the concept of *negative provenance* [10, 32]. The key insight is to use counterfactual reasoning, that is, to examine all possible causes that *could have* produced the missing effect. For instance, it might be the case that the missing packets could only have reached the server through one of two upstream switches, and that one of them is missing a flow entry that would match the packets. Based on the controller program, we might then establish that the missing entry could only have been installed if a certain condition had been satisfied, and so on, until we either reach a positive event (such as the installation of a conflicting flow entry with higher priority) that can be traced with normal provenance, or a negative root cause (such as a missing entry in a configuration file).

Negative provenance could be a useful debugging tool for networks and distributed systems in general, but so far it has not been explored very much. A small number of papers from the database

A shorter version of this paper has appeared at ACM SIGCOMM 2014.
http://dx.doi.org/10.1145/2619239.2626335.

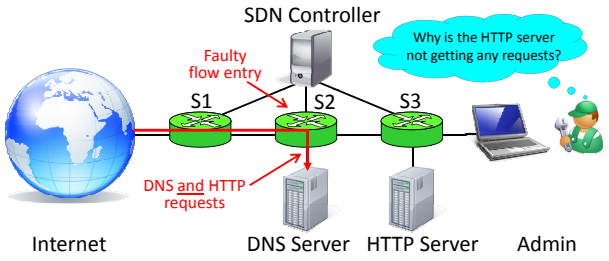


Figure 1: Negative event scenario: Web requests from the Internet are no longer reaching the web server because a faulty program on the controller has installed an overly general flow entry in the switch in the middle (S2).

community [3, 10, 19] have used negative provenance to explain why a given database query did *not* return a certain tuple, but, other than our earlier workshop paper that made a case for negative provenance [32], we are not aware of any previous applications in the networking domain.

In Section 2, we provide an overview of positive and negative provenance. We then make the following contributions:

- A formal model of positive and negative provenance in distributed systems, as well as a concrete algorithm for tracking such provenance (Section 3);
- A set of heuristics for simplifying the resulting provenance graphs and for making them more readable to a human investigator (Section 4);
- The design of Y! (pronounced "Why not?"), a system for tracking positive and negative provenance and for answering queries about it (Section 5);
- Two case studies of Y!, in the context of software-defined networks and BGP (Section 6); and
- An experimental evaluation of an Y! prototype, based on Mininet, Trema [29] and RapidNet [24] (Section 7).

We discuss related work in Section 8 and conclude the paper in Section 9.

2. OVERVIEW

In this section, we take a closer look at negative provenance, and we discuss some of the key challenges.

2.1 Scenario: Network debugging

Figure 1 shows a simple example scenario that illustrates the problem we are focusing on. A network administrator manages a small network that includes a DNS server, a web server, and a connection to the Internet. At some point, the administrator notices that the web server is no longer receiving any requests from the Internet. The administrator strongly suspects that the network is somehow misconfigured, but the only observable symptom is a *negative event* (the absence of web requests at the server), so there is no obvious starting point for an investigation.

Today, the only way to resolve such a situation is to manually inspect the network until some positive symptom (such as requests arriving at the wrong server) is found. In the very simple scenario in Figure 1, this is not too difficult, but in a data center or large corporate network, it can be a considerable challenge. It seems preferable for the administrator to directly ask the network for an explanation of the negative event, similar to a "backtrace" in a conventional debugger. This is the capability we seek to provide.

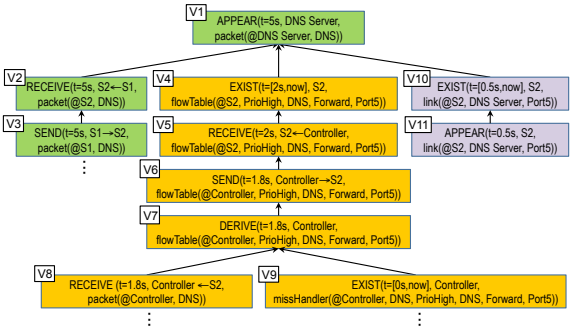


Figure 2: *Positive* provenance example, explaining how a DNS packet made its way to the DNS server.

2.2 Positive provenance

For *positive* events, there is a well-understood way to generate such a "backtrace": whenever an event occurs or some new state is generated, the system keeps track of its causes, and when an explanation is requested, the system recursively explains each event with its direct causes, until it reaches a set of "base events" (such as external inputs) that cannot be explained further. The result can be represented as a DAG, in which each vertex represents an event and each edge indicates a direct causal relationship. In the database literature, this is called the *provenance* [1] of the event; to distinguish it from negative provenance, we will refer to it as *positive provenance*.

Figure 2 shows an example that explains why a DNS request appeared at the DNS server at time $t = 5$ (V1). The DNS server had received the packet from switch S2, which in turn had received it from S1, and ultimately from the Internet (V2–V3); the switch was connected to the DNS server via port #5 (V10–V11) and had a flow entry that directed DNS packets to that port (V4). The flow entry had been installed at $t = 2$ (V5–V7) because the switch had forwarded an earlier DNS packet to the controller (V8), which had generated the flow entry based on its configuration (V9).

Positive provenance is an active research area, but some solutions for distributed systems are already available, such as ExSPAN [36] or SNP [34]. Provenance is often a useful debugging tool, just like the "backtraces" that many debuggers are offering.

2.3 Case study: Broken flow entry

We now return to the scenario in Figure 1. One possible reason for this situation is that the administrator has configured the controller to produce a generic, low-priority flow entry for DNS traffic and a specific, high-priority flow entry for HTTP traffic. If both entries are installed, the system works as expected, but if the low-priority entry is installed first, it matches HTTP packets as well; thus, these packets are not forwarded to the controller and cannot trigger the installation of the high-priority entry. This subtle race condition might manifest only at runtime, e.g., when both entries expire simultaneously during an occasional lull in traffic; thus, it could be quite difficult to find.

Positive provenance is not helpful here because, as long as requests are still arriving at the HTTP server, their provenance contains only the high-priority entry, and when the requests stop arriving, there is no longer anything to generate the provenance *of*!

2.4 How common are negative symptoms?

To get a sense of how common this situation is, we surveyed diagnostics-related posts on three mailing lists. We chose lists that cover a mix of different diagnostic situations, including NANOG-user and Outages [23] (for faults and misconfigurations), and

Mailing list	Posts related to diagnostics	Initial symptoms	
		Positive	Negative
NANOG-user	29/144	14 (48%)	15 (52%)
floodlight-dev	19/154	5 (26%)	14 (74%)
Outages [23]	46/60	8 (17%)	38 (83%)

Table 1: Survey of networking problems and their symptoms, as discussed on three mailing lists over a two-month period, starting on November 22, 2013.

floodlight-dev (for software bugs). To get a good sample size, we examined a two-month period for each list, starting on November 22, 2013. In each post, we looked for the description of the initial symptoms and classified them as either positive (something bad happened) or negative (something good failed to happen).

Table 1 shows our results. While the proportion of positive and negative symptoms varies somewhat between lists, we find that the negative symptoms are consistently *in the majority* – that is, it seems more common for problems to initially manifest as the absence of something (e.g., a route, or a response to a probe packet) than as the presence of something (e.g., high latencies on a path, or a DDoS attack).

Many of the problems we surveyed were eventually diagnosed, but we observe that the process seems comparatively harder: there were significantly more (and lengthier) email threads where negative symptoms resulted in inconclusive identification of root causes. Moreover, troubleshooting negative symptoms often required exploratory "guesswork" by the mailing list participants. Since this trial-and-error approach requires lots of time and effort, it seems useful to develop better tool support for this class of problems.

2.5 Negative provenance

Our approach towards such a tool is to extend provenance to negative events. Although these cannot be explained directly with positive provenance, there is a way to construct a similar "backtrace" for negative events: instead of explaining how an actual event *did* occur, as with positive provenance, we can simply find all the ways in which a missing event *could have* occurred, and then show, as a "root cause", the reason why each of them did not come to pass.

Intuitively, we can use a kind of counterfactual reasoning to recursively generate the explanations, not unlike positive provenance: for a web request to arrive at the web server, a request would have had to appear at the rightmost switch (S3), which did not happen. Such a request could only have come from the switch in the middle (S2), and, eventually, from the switch on the left (S1). But S2 would only have sent the request if there had been 1) an actual request, 2) a matching flow entry with a forward action to S3, and 3) no matching higher-priority flow entry. Conditions 1) and 2) were satisfied, but condition 3) was not (because of the DNS server's flow entry). We can then ask where the higher-priority flow entry came from, which can be answered with positive provenance. We refer to such a counterfactual explanation as *negative provenance*.

2.6 Challenges

To explain the key challenges, we consider two strawman solutions. First, it may seem that there is a simpler way to investigate the missing HTTP requests from Section 2.3: why not simply compare the system state before and after the requests stopped arriving, and return any differences as the likely cause? This approach may indeed work in some cases, but in general, there are way too many changes happening in a typical system: even if we could precisely pinpoint the time where the problem appeared, chances are that most of the state changes at that time would be unrelated. Moreover, if the problem was caused by a *chain* of events, this method would return only the last step in the chain. To identify the relevant events reliably, and to trace them back to the root cause, we must have a way to track *causality*, which is, in essence, what provenance represents.

Second, it may seem that, in order to track negative provenance, we can simply take an existing provenance system, like ExSPAN or SNP, and associate each positive provenance vertex with a negative "twin". However, the apparent similarity between positive and negative provenance does not go very deep. While positive provenance considers only *one specific* chain of events that led to an observed event, negative provenance must consider *all possible* chains of events that *could have* caused the observed event. This disparity between existential and universal quantifiers has profound consequences: for instance, negative provenance graphs are often infinite and cannot be materialized, and responses to negative queries tend to be a lot more complex, and thus need more sophisticated post-processing before they can be shown to a human user. These are some of the challenges we address in Y!.

3. BASIC NEGATIVE PROVENANCE

In this section, we show how to derive a simple provenance graph for both positive and negative events. For ease of exposition, we will assume that the distributed system is written in Network Datalog (NDlog) [16], since this representation makes provenance particularly easy to see. However, our approach is not specific to NDlog, or even to declarative languages; indeed, our case studies in Section 6.1 apply it to Pyretic [21], an existing imperative programming language for SDNs, as well as to BGP debugging.

3.1 Background: Network Datalog

We begin by briefly reviewing the features of NDlog that are relevant here. In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*. Each table contains a number of *tuples*. For instance, an SDN switch might contain a table called `flowTable`, and each tuple in this table might represent a flow entry, or a SDN controller might have a table called `packetIn` that contains the packets it has received from the switches. Tuples can be manually inserted, or they can be programmatically derived from other tuples; the former are called *base tuples*, and the latter are referred to as *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from each other. For instance, the rule `A(@X,P):-B(@X,Q),Q=2*P` says that a tuple `A(@X,P)` should be derived on node X whenever there is also a `B(@X,Q)` tuple on that node, and `Q=2*P`. Here, P and Q are variables that must be instantiated with values when the rule is applied; for instance, a `B(@X,10)` tuple would create an `A(@X,5)` tuple. The `@` operator specifies the node on which the tuple resides. (NDlog supports other operators – e.g., arithmetic or aggregation operators – as well as user-defined functions, but we do not consider these here.) A key advantage of a declarative formulation is that causality is very easy to see: if a tuple `A(@X,5)` was derived using the rule above, then `A(@X,5)` exists simply because `B(@X,10)` exists, and because `10=2*5`.

Rules may include tuples from different nodes; for instance, `C(@X,P):- C(@Y,P)` says that tuples in the C-table on node Y should be sent to node X and inserted into the C-table there. We write $+\tau$ to denote a message saying that the recipient should insert a tuple τ, and $-\tau$ to denote a message saying that the recipient should remove τ. To avoid redundancy, we write $\pm\tau$ when a statement applies to both types of messages. When a tuple can be derived in more than one way, $+\tau$ is only sent when the first derivation appears, and $-\tau$ only when the last derivation disappears.

3.2 Goals

Before we define our provenance graph, we first state, somewhat informally, the properties we would like to achieve. One way to describe what "actually happened" in an execution of the system is by means of a *trace*: a sequence of message transmissions and arrivals, as well as base tuple insertions and deletions. (Other events, such as derivations, follow deterministically from these.) Following [35], we can then think of the provenance $G(e, \mathcal{E})$ of an event e in a trace $\mathcal{E}$ as describing a series of trace properties, which, in combination, cause e to appear – or, in the case of a negative event, prevent e from appearing. We demand the following properties:

- **Soundness:** $G(e, \mathcal{E})$ must be consistent with $\mathcal{E}$;
- **Completeness:** There must not be another execution $\mathcal{E}'$ that is also consistent with $G(e, \mathcal{E})$ but does not contain the event e; and
- **Minimality:** There must not be a subset of $G(e, \mathcal{E})$ that is also sound and complete.

Informally, soundness means that $G(e, \mathcal{E})$ must describe events that actually happened in $\mathcal{E}$ – we cannot explain the absence of a tuple with the presence of a message that was never actually sent. Completeness means that $G(e, \mathcal{E})$ must be sufficient to explain e, and minimality means that all events in $G(e, \mathcal{E})$ must actually be relevant (though there could be more than one provenance that is minimal in this sense). We will state these properties formally in Section 3.8.

3.3 The provenance graph

Provenance can be represented as a DAG in which the vertices are events and the edges indicate direct causal relationships. Thanks to NDlog's simplicity, it is possible to define a very simple provenance graph for it, with only ten types of event vertices (based on [34]):

- EXIST$([t_1, t_2], N, \tau)$: Tuple τ existed on node N from time t_1 to t_2;
- INSERT(t, N, τ), DELETE(t, N, τ): Base tuple τ was inserted (deleted) on node N at time t;
- DERIVE(t, N, τ), UNDERIVE(t, N, τ): Derived tuple τ acquired (lost) support on N at time t;
- APPEAR(t, N, τ), DISAPPEAR(t, N, τ): Tuple τ appeared or disappeared on node N at time t;
- SEND$(t, N \rightarrow N', \pm\tau)$, RECEIVE$(t, N \leftarrow N', \pm\tau)$: $\pm\tau$ was sent (received) by node N to/from N' at t; and
- DELAY$(t, N \rightarrow N', \pm\tau, d)$: $\pm\tau$, sent from node N to N' at time t, took time d to arrive.

The edges between the vertices correspond to their intuitive causal connections: tuples can appear on a node because they a) were inserted as base tuples, b) were derived from other tuples, or c) were received in a message from another node (for cross-node rules). Messages are received because they were sent, and tuples exist because they appeared. Note that vertices are annotated with the node on which they occur, as well as with the relevant time; the latter will be important for negative provenance because we will often need to reason about past events.

This model can be extended to support negative provenance by associating each vertex with a negative "twin" [32]:

- NEXIST$([t_1, t_2], N, \tau)$: Tuple τ never existed on node N in time interval $[t_1, t_2]$;
- NINSERT$([t_1, t_2], N, \tau)$, NDELETE$([t_1, t_2], N, \tau)$: Tuple τ was never inserted (removed) on N in $[t_1, t_2]$;
- NDERIVE$([t_1, t_2], N, \tau)$, NUNDERIVE$([t_1, t_2], N, \tau)$: τ was never derived (underived) on N in $[t_1, t_2]$;
- NAPPEAR$([t_1,t_2],N, \tau)$, NDISAPPEAR$([t_1,t_2],N, \tau)$: Tuple τ never (dis)appeared on N in $[t_1, t_2]$;
- NSEND$([t_1, t_2], N, \tau)$, NRECEIVE$([t_1, t_2], N, \tau)$: τ was never sent (received) by node N in $[t_1, t_2]$; and
- NARRIVE$([t_1, t_2], N_1 \rightarrow N_2, t_3, \tau)$: τ was sent from N_1 to N_2 at t_3 but did not arrive within $[t_1, t_2]$.

Again, the causal connections are the intuitive ones: tuples never existed because they never appeared, they never appeared because they were never inserted, derived, or received, etc. However, note that, unlike their positive counterparts, *all* negative vertices are annotated with time intervals: unlike positive provenance, which can refer to specific events at specific times, negative provenance must explain the *absence* of events in certain intervals.

3.4 Handling multiple explanations

Sometimes the absence of an event can have more than one cause. For instance, suppose there is a rule `A:-B,C,D` and, at some time t, none of the tuples B, C, or D exist. How should we explain the absence of A in this case? One possible approach would be to include the absence of all three tuples; this would be useful, for instance, if our goal was recovery – i.e., if we wanted to find a way make A appear. However, for diagnostic purposes, the resulting provenance is somewhat verbose, since the absence of each individual tuple is already sufficient to explain the absence of A. For this reason, we opt to include only a *sufficient reason* in our provenance trees.

In cases where there is more than one sufficient reason, the question arises which one we should choose. Since we aim for compact provenance trees, *we try to find the reason that can be explained with the fewest vertices.* For instance, if B and D are derived tuples whose absence is due to a complex sequence of events on several other nodes, whereas C is a base tuple that simply was never inserted, we would choose the explanation that is based on C, which only requires a single NINSERT vertex. In practice, it is not always easy to see which explanation is simplest (at least not without fully expanding the corresponding subtree), but we can use heuristics to find a good approximation, e.g., based on a look-ahead of a few levels down the tree.

3.5 Graph construction

Provenance systems like ExSPAN [36] rely on a materialized provenance graph: while the distributed system is executing, they build some representation of the vertices and edges in the graph, and they respond to queries by projecting out the relevant subtree. This approach does not work for negative provenance because the provenance graph is typically infinite: for instance, it contains NEXIST vertices for every tuple that could *potentially* exist, and, for each vertex with that contains a time interval I, it also contains vertices with intervals $I' \subseteq I$.

For this reason, we adopt a top-down procedure for constructing the provenance of a given (positive or negative) event "on demand", without materializing the entire graph. We define a function QUERY(v) that, when called on a vertex v in the provenance graph, returns the immediate children of v. Thus, the provenance of a negative event e can be found by constructing a vertex v_e that describes e (e.g., a NEXIST vertex for an absent tuple) and then calling QUERY recursively on v_e until leaf vertices are reached. The

function QUERY(EXIST([t_1, t_2],N,τ))
  S $\leftarrow \emptyset$
  for each (+τ,N,t,r,c) $\in$ Log: $t_1 \leq t \leq t_2$
    S $\leftarrow$ S $\cup$ { APPEAR(t,N,τ,r,c) }
  for each (−τ,N,t,r,c) $\in$ Log: $t_1 \leq t \leq t_2$
    S $\leftarrow$ S $\cup$ { DISAPPEAR(t,N,τ,r,c) }
  RETURN S
function QUERY(APPEAR(t,N,τ,r,c))
  if BaseTuple(τ) **then**
    RETURN { INSERT(t,N,τ) }
  else if LocalTuple(N,τ) **then**
    RETURN { DERIVE(t,N,τ,r) }
  else RETURN{RECEIVE(t,$N \leftarrow r.N$,τ)}
function QUERY(INSERT(t,N,τ))
  RETURN $\emptyset$
function QUERY(DERIVE(t,N,τ,τ:-τ_1,τ_2...))
  S $\leftarrow \emptyset$
  for each τ_i: **if** (+τ_i,N,t,r,c) $\in$ Log:
    S $\leftarrow$ S $\cup$ { APPEAR(t,N,τ_i,c) }
  else
    $t_x \leftarrow$ max $t'<t$: (+τ,N,t',r,1) $\in$ Log
    S $\leftarrow$ S $\cup$ { EXIST([t_x,t],N,τ_i,c) }
  RETURN S
function QUERY(RECEIVE(t,$N_1 \leftarrow N_2$,+τ))
  $t_s \leftarrow$ max $t'<t$: (+τ,N_2,t',r,1) $\in$ Log
  RETURN { SEND(t_s,$N_1 \rightarrow N_2$,+τ),
    DELAY(t_s,$N_2 \rightarrow N_1$,+τ,$t - t_s$) }
function QUERY(SEND(t,$N \rightarrow N'$,+τ))
  FIND (+τ,N,t,r,c) $\in$ Log
  RETURN { APPEAR(t,N',τ,r) }
function QUERY(NEXIST([t_1,t_2],N,τ))
  if $\exists t < t_1$: (-τ,N,t,r,1) $\in$ Log **then**
    $t_x \leftarrow$ max $t<t_1$: (-τ,N,t,r,1) $\in$ Log
    RETURN { DISAPPEAR(t_x,N,τ),
      NAPPEAR((t_x,t_2],N,τ) }
  else RETURN { NAPPEAR([0,t_2],N,τ) }
function QUERY(NDERIVE([t_1,t_2],N,τ,r))
  S $\leftarrow \emptyset$
  for (τ_i, I_i) $\in$ PARTITION([t_1,t_2],N,τ,r)
    S $\leftarrow$ S $\cup$ { NEXIST(I_i,N,τ_i) }
  RETURN S
function QUERY(NSEND([t_1,t_2],N,+τ))
  if $\exists t_1 < t < t_2$: (-τ,N,t,r,1) $\in$ Log **then**
    RETURN { EXIST([t_1,t],N,τ),
      NAPPEAR((t,t_2],N,τ) }
  else RETURN { NAPPEAR([t_1,t_2],N,τ) }
function QUERY(NAPPEAR([t_1,t_2],N,τ))
  if BaseTuple(τ) **then**
    RETURN { NINSERT([t_1,t_2],N,τ) }
  else if LocalTuple(N,τ) **then**
    RETURN $\bigcup_{r \in \text{Rules(N):Head}(r)=\tau}$
      { NDERIVE([t_1,t_2],N,τ,r) }
  else RETURN {NRECEIVE([t_1,t_2],N,+τ)}
function QUERY(NRECEIVE([t_1,t_2],N,+τ))
  S $\leftarrow \emptyset$, $t_0 \leftarrow t_1 - \Delta$max
  for each $N' \in$ SENDERS(τ,N):
    X$\leftarrow$ {$t_0 \leq t \leq t_2$|(+τ,N',t,r,1)$\in$Log}
    $t_x \leftarrow t_0$
    for (i=0; i< $|X|$; i++)
      S$\leftarrow$S$\cup${NSEND((t_x,X_i),N',+τ),
      NARRIVE((t_1,t_2),$N' \rightarrow N$,X_i,+τ)}
      $t_x \leftarrow X_i$
    S $\leftarrow$ S $\cup$ { NSEND([t_x,t_2],N',+τ) }
  RETURN S
function Q(NARRIVE([t_1,t_2],$N_1 \rightarrow N_2$,t_0,+τ))
  FIND (+τ,N_2,t_3,(N_1,t_0),1) $\in$ Log
  RETURN { SEND(t_0,$N_1 \rightarrow N_2$,+τ),
    DELAY(t_0,$N_1 \rightarrow N_2$,+τ,$t_3 - t_0$) }

Figure 3: Graph construction algorithm. Some rules have been omitted; for instance, the handling of $+\tau$ and $-\tau$ messages is analogous, and the rules for INSERT/DELETE, APPEAR/DISAPPEAR, and DERIVE/UNDERIVE are symmetric.

interval in v_e can simply be some interval in which e was observed; it does not need to cover the entire duration of e, and it does not need to contain the root cause(s).

QUERY needs access to a log of the system's execution to date. We assume that the log is a sequence of tuples $(\pm\tau, N, t, r, c)$, which indicate that τ was derived ($+\tau$) or underived ($-\tau$) on node N at time t via rule r. Since some tuples can be derived in more than one way, we include a derivation counter c, which is 1 when a tuple first appears, and is increased by one for each further derivation. For tuples that node N received from another node N', we set $r = N'$, and for base tuples, we set $r = \perp$ and $c = 1$.

Figure 3 shows part of the algorithm we use to construct positive and negative provenance. There are several points worth noting. First, the algorithm uses functions BaseTuple(τ) and LocalTuple(N,τ) to decide whether a missing tuple τ is a base tuple that was not inserted, a local tuple on node N that was not derived, or a remote tuple that was not received. The necessary information is a byproduct of the compilation of any NDlog program and is thus easily obtained. Second, to account for propagation delays, the algorithm uses a constant Δ_{max} that denotes the maximum time a message can spend in the network and still be accepted by the recipient; this is used to narrow down the time interval during which a missing message could have been sent. Third, the algorithm can produce the same vertex more than once, or semantically identical vertices with adjacent or overlapping intervals; in these cases, it is necessary to coalesce the vertices using the union of their intervals in order to preserve minimality. Finally, the algorithm uses two functions PARTITION and SENDERS, which we explain next.

3.6 Explaining nonderivation

The PARTITION function encodes a heuristic for choosing among several possible explanations of a missing derivation. When explaining why a rule with multiple preconditions did not derive a certain tuple, we must consider a potentially complex parameter space. For instance, if `A(@X,p):-B(@X,p,q,r),C(@X,p,q)` did not derive `A(@X,10)`, we can explain this with the absence of `B(@X,10,q,r)`, `C(@X,10,q,r)`, or a combination of both – e.g., by dividing the possible q and r values between the two preconditions. Different choices can result in explanations of dramatically different sizes once the preconditions themselves have been explained; hence, we would prefer a partition of the parameter space (here, $Q \times R$) that results in an explanation that is as small as possible. In general, finding the optimal partition is at least as hard as the SETCOVER problem, which is NP-hard; hence the need for a heuristic. In our experiments, we use a simple greedy heuristic that always picks the largest available subspace; if there are multiple subspaces of the same size, it explores both for a few steps and then picks the one with the simplest subgraph.

3.7 Missing messages

The SENDERS($\pm\tau$,N) function is used to narrow down the set of nodes that could have sent a specific missing message $\pm\tau$ to node N. One valid choice is to simply return the set of all nodes in the system that have a rule for deriving τ; however, the resulting provenance can be complex, since it must explain why *each* of these nodes did not send $\pm\tau$. Hence, it is useful to enhance SENDERS with other information that may be available. For instance, in a routing protocol, communication is restricted by the network topology, and messages can come only from direct neighbors.

In some cases, further nodes can be ruled out based on the specific message that is missing: for instance, a BGP message whose AS path starts with 7 should come from a router in AS 7. We do not pursue this approach here, but we hypothesize that static analysis of the NDlog program could be used for inferences of this type.

3.8 Formal properties

We now briefly present the key definitions from our formal model (see Appendix A). An *event* $d@n = (m, r, t, c, m')$ represents that rule r was triggered by message m and generated a set of (local or remote) messages m' at time t, given the precondition c (a set of tuples that existed on node n at time t). Specifically, we write $d@n_{recv} = (m@n_{send}, -, t, 1, m@n_{recv})$ to denote a message m (from n_{send} is delivered at n_{recv} at t). A *trace* $\mathcal{E}$ of a system execution is an ordered sequence of events from an initial state $\mathcal{S}_0$, $\mathcal{S}_0 \xrightarrow{d_1@n_1} \mathcal{S}_1 \xrightarrow{d_2@n_2} \ldots \xrightarrow{d_x@n_x} \mathcal{S}_x$. We say a trace $\mathcal{E}$ is *valid*, if (a) for all $\tau_k \in c_i$, $\tau_k \in \mathcal{S}_{i-1}$, and (b) for all

$d_i@n_i = (m_i, r_i, t_i, c_i, m'_i)$, either m_i is a base message from an external source, or there exists $d_j@n_j = (m_j, r_j, t_j, c_j, m'_j)$ that precedes $d_i@n_i$ and $m_i \in m'_j$. We say that $\mathcal{E}'$ is a *subtrace* of $\mathcal{E}$ (written as $\mathcal{E}' \subseteq \mathcal{E}$) if $\mathcal{E}'$ consists of a subset of the events in $\mathcal{E}$ in the same order. In particular, we write $\mathcal{E}|n$ to denote the subtrace that consists of all the events on node n in $\mathcal{E}$. We say that $\mathcal{E}'$ and $\mathcal{E}$ are equivalent (written as $\mathcal{E}' \sim \mathcal{E}$) if, for all n, $\mathcal{E}'|n = \mathcal{E}|n$.

To properly define minimality, we use the concept of a *reduction*: given negative provenance $G(e, \mathcal{E})$, if there exist vertices $v_1, v_2 \in V(G)$, where the time interval of v_1 and v_2 ($t(v_1)$ and $t(v_2)$ respectively) are adjacent, and v_1 and v_2 have the same dependencies, then G can be reduced to G' by combining v_1 and v_2 into $v \in V(G')$, where $t(v) = t(v_1) \cup t(v_2)$. Given two negative provenance $G(e, \mathcal{E})$ and $G'(e, \mathcal{E})$, we say G' is *simpler* than G (written as $G' < G$), if any of the following three hold: (1) G' is a subgraph of G; (2) G' is reduced from G (by combining v_1 and v_2); or (3) there exists G'', such that $G' < G''$ and $G'' < G$.

Using these definitions, we can formally state the three properties from Section 3.2 as follows:

Property (Soundness): *Negative provenance $G(e, \mathcal{E})$ is sound iff (a) it is possible to extract a valid subtrace $\mathcal{E}_{sub} \subseteq \mathcal{E}'$, such that $\mathcal{E}' \sim \mathcal{E}$ and (b) for all vertices in $G(e, \mathcal{E})$, their corresponding predicates hold in $\mathcal{E}$.*

Property (Completeness): *Negative provenance $G(e, \mathcal{E})$ is complete iff there exists no trace $\mathcal{E}'$ such that a) $\mathcal{E}'$ assumes the same external inputs as $G(e, \mathcal{E})$, and b) e exists in $\mathcal{E}'$.*

Property (Minimality): *Negative provenance $G(e, \mathcal{E})$ is minimal, if no $G' < G$ is sound and complete.*

We have proven that our provenance graph has all three properties. The proofs are available in Appendix A.

4. ENHANCING READABILITY

So far, we have explained how to generate a "raw" provenance graph. This representation is correct and complete, but it is also extremely detailed: for instance, simple and common events, such as message exchanges between nodes, are represented with many different vertices. This "clutter" can make the provenance difficult to read. Next, we describe a post-processing technique that can often simplify the provenance considerably, by pruning unhelpful branches, and by summarizing common patterns into higher-level vertices.

4.1 Pruning unhelpful branches

Logical inconsistencies: Some explanations contain logical inconsistencies: for instance, the absence of a tuple τ_1 with parameter space S_1 might be explained by the absence of a tuple τ_2 with parameter space $S_2 \subseteq S_1$. If we can recognize such inconsistencies early, there is no need to continue generating the provenance until a set of base tuples is reached – the precondition is clearly unsatisfiable. Thus, we can safely truncate the corresponding branch of the provenance tree.

Failed assertions: Some branches explain the absence of events that the programmer has already ruled out. For instance, if a branch contains a vertex NEXIST($[t_1, t_2]$,N,P(5)) and it is known that P can only contain values between 0 and 4, the subtree below this vertex is redundant and can be removed. We use programmer-specified assertions (where available) to recognize situations of this type. The assertions do not have to be provenance-specific – they can be the ones that a good programmer would write anyway.

Branch coalescing: A naïve execution of the algorithm in Figure 3 would result in a provenance *tree*, but this tree would contain many duplicate vertices because many events have more than one effect. To avoid redundancy, we combine redundant vertices whenever possible, which turns the provenance tree into a DAG. If two vertices have overlapping time intervals but are otherwise identical, we use the union of the two intervals. (Note that a smart implementation of PARTITION could take the multiplicity of shared subtrees into account.)

Application-specific invariants: Some explanations may be irrelevant for the particular SDN that is being debugged. For instance, certain data – such as constants, topology information, or state from a configuration file – changes rarely or never, so the absence of changes, or the presence of a specific value, do not usually need to be explained. One simple way to identify constant tables is by the absence of derivation rules in which the table appears at the head. Optionally, the programmer can use a special keyword to designate additional tables as constant.

4.2 Different levels of detail

Another way to make negative provenance graphs more useful for the human investigator is to display the provenance at different levels of detail. For instance, if a message fails to appear at node N_1 but could only have originated at node N_2 several hops away, the basic provenance tree would show, for each node on the path from N_1 to N_2, that the message was not sent from there, because it failed to appear there, because it was not received from the next-hop node, etc. We can improve readability by summarizing these (thematically related) vertices into a single *super-vertex*. When the graph is first shown to the human investigator, we include as many super-vertices as possible, but the human investigator has the option to expand each super-vertex into the corresponding fine-grained vertices if necessary.[1]

We have identified three situations where this summarization can be applied. The first is a chain of transient events that originates at one node and terminates at another, as in the above example; we replace such chains by a single super-vertex. The second is the (common) sequence NEXIST$([t_1, t_2], N, \tau) \leftarrow$ NAPPEAR$([t_1, t_2], N, \tau) \leftarrow$ NDERIVE$([t_1, t_2], N, \tau)$, which basically says that a tuple was never derived; we replace this with a single ABSENCE$([t_1, t_2], N, \tau)$ super-vertex; its positive counterpart EXISTENCE$([t_1, t_2], N, \tau)$ is used to replace a positive sequence. The third situation is a derivation that depends on a small set of triggers – e.g., flow entries can only be generated when a packet p is forwarded to the controller C. In this case, the basic provenance will contain a long series of NAPPEAR$([t_i, t_{i+1}], C, p)$ vertices that explain the *common* case where the trigger packet p *does not* exist; we replace these with a single super-vertex ONLY-EXIST$(\{t_1, t_2, \ldots\}$ *in* $[t_{\text{start}}, t_{\text{end}}], C, p)$ that initially focuses attention on the *rare* cases where the trigger *does* exist.

5. THE Y! SYSTEM

In this section, we describe the design of Y! (for "Why not?"), a system for capturing, storing, and querying both positive and negative provenance.

5.1 Overview

Like any debugger, Y! is meant to be used in conjunction with some other application that the user wishes to diagnose; we refer to this as the *target application*. Y! consists of four main components: The *provenance extractor* (Section 5.2) monitors the target

[1]More generally, visualization and interactive exploration are useful strategies for working with large provenance graphs [17].

application and extracts relevant events, such as state changes or message transmissions. These events are passed to the *provenance storage* (Section 5.3), which appends them to an event log and also maintains a pair of indices to enable efficient lookups of negative events. When the user issues a provenance query, the *query processor* uses the stored information to construct the relevant subtree of the provenance graph, simplifies the subtree using the heuristics from Section 4, and then sends the result to the *frontend*, so that the user can view, and interactively explore, the provenance. We now explain some key components in more detail.

5.2 Provenance extractor

Recall from Section 3 that the input to the graph construction algorithm is a sequence of entries $(\pm\tau, N, t, r, c)$, which indicate that the c.th derivation of tuple τ appeared or disappeared on node N at time t, and that the reason was r, i.e., a derivation rule or an incoming message. The purpose of the provenance extractor is to capture this information from the target application. This functionality is needed for all provenance systems (not just for negative provenance), and it should be possible to use any of the several approaches that have been described in the literature. For instance, the target application can be annotated with calls to a special library whenever a relevant event occurs [22], the runtime that executes the target application (e.g., an NDlog engine or a virtual machine) can report the relevant events [36], or a special proxy can reconstruct the events from the sequence of messages that each node sends and receives [34]. Note that the latter two approaches can be applied even to legacy software and unmodified binaries.

5.3 Provenance storage

The provenance storage records the extracted events in an append-only log and makes this log available to the query processor. A key challenge is efficiency: with positive provenance, it is possible to annotate each event with pointers to the events that directly caused it, and, since there is a fairly direct correspondence between events and positive vertices in the provenance graph, these pointers can then be used to quickly navigate the graph. With negative provenance, however, it is frequently necessary to evaluate range queries over the time domain ("Did tuple τ ever exist during interval $[\tau_1, \tau_2]$"). Moreover, our PARTITION heuristic requires range queries over other domains, e.g., particular subspaces of a given table ("Are there any `X(a,b,c)` tuples on this node with $5 \leq b \leq 20$?") to decide which of several possible explanations might be the simplest. If Y! evaluated such range queries by scanning the relevant part of the log, performance would suffer greatly.

Instead, Y! uses R-trees [7] to efficiently access the log. R-trees are tree data structures for indexing multi-dimensional data; briefly, the key idea is to group nearby objects and to represent each group by its minimum bounding rectangle at the next-higher level of the tree. Their key advantage in our setting is that they can efficiently support multidimensional range queries.

On each node, Y! maintains two different R-trees for each table on that node. The first, the *current tree*, contains the tuples that currently exist in the table; when tuples appear or disappear, they are also added or removed from the current tree. The second, the *historical tree*, contains the tuples that have existed in the past. State tuples are added to the historical tree when they are removed from the current tree; event tuples, which appear only for an instant, are added directly to the historical tree.

The reason for having two separate trees is efficiency. It is known that the performance of R-trees degrades when elements are frequently inserted and removed because the bounding rectangles will no longer be optimal and will increasingly overlap. By separating the historical tuples (where deletions can no longer happen) from the current tuples, we can obtain a more compact tree for the former and confine fragmentation to the latter, whose tree is much smaller. As an additional benefit, since tuples are appended to the historical tree in timestamp order, splits in that tree will typically occur along the time dimension; this creates a kind of "time index" that works very well for our queries.

5.4 Pruning the historical tree

Since the historical tree is append-only, it would eventually consume all available storage. To avoid this, Y! can reclaim storage by deleting the oldest tuples from the tree. For instance, Y! can maintain a cut-off time T_{cut}; whenever the tree exceeds a certain pre-defined size limit, Y! can slowly advance the cut-off time and keep removing any tuples that existed before that time until enough space has been freed. To enable the user to distinguish between tuples that were absent at runtime and tuples that have been deleted from the tree, the graph construction algorithm can, whenever it accesses information beyond T_{cut}, annotate the corresponding vertex as potentially incomplete.

5.5 Limitations

Like other provenance systems, Y!'s explanations are limited by the information that is available in the provenance graph. For instance, Y! could trace a misconfiguration to the relevant setting, but not to the person who changed the setting (unless that information were added to the provenance graph). Y! also has no notion of a program's *intended* semantics: for instance, if a program has a concurrency bug that causes a negative event, a query for that event will yield a detailed explanation of how the given program produced that event. Only the operator can determine that the program was supposed to do something different.

6. CASE STUDIES

In this section, we describe how we have applied Y! to two application domains: software-defined networks (SDN) and BGP routing. We chose these domains partly because they yield interesting debugging challenges, and partly because they do not already involve declarative code (applying Y! to NDlog applications is straightforward!). We illustrate two different implementation strategies: automatically extracting declarative rules from existing code (for SDN) and writing a declarative description of an existing implementation (for BGP). We report results from several specific debugging scenarios in Section 7.

6.1 SDN debugging

Our first case study is SDN debugging: as others [8] have pointed out, better debugging support for SDNs is urgently needed. This scenario is challenging for Y! because SDNs can have almost arbitrary control programs, and because these programs are typically written in non-declarative languages. Provenance can be extracted directly from imperative programs [22], but switching to a different programming model would require some adjustments to our provenance graph. Hence, we use automated program transformation to extract declarative rules from existing SDN programs.

Language: Pyretic We chose to focus on the Pyretic language [21]. We begin by briefly reviewing some key features of Pyretic that are relevant here. For details, please see [21].

Pyretic programs can define a mix of *static* policies, which are immutable, and *dynamic* policies, which can change at runtime based on system events. Figure 4 shows a summary of the relevant syntax. A static policy consists of *actions*, e.g., for forwarding packets to a specific port (`fwd(port)`), and *predicates* that

Primitive actions:

```
A ::= drop | passthrough | fwd(port) | flood |
      push(h=v) | pop(h) | move(h1=h2)
```

Predicates:

```
P ::= all_packets | no_packets | match(h=v) |
      ingress | egress | P & P | (P | P) | ~P
```

Query policies:

```
Q ::= packets(limit,[h]) | counts(every,[h])
```

Policies:

```
C ::= A | Q | P[C] | (C|C) | C>>C | if_(P,C,C)
```

Figure 4: Static Pyretic syntax (from [21])

```
def learn(self):
  def update(pkt):
    self.P = if_(match(dstmac=pkt['srcmac']),
    switch=pkt['switch']),
      fwd(pkt['inport']), self.P)
    q = packets(1,['srcmac','switch'])
    q.when(update)
    self.P = flood | q
def main():
  return dynamic(learn)()
```

Figure 5: Self-learning switch in Pyretic (from [21])

restrict these actions to certain types of packets, e.g., to packets with certain header values (`match(h=v)`). Two policies `a` and `b` can be combined through *parallel composition* (`a|b`), meaning that `a` and `b` should be applied to separate copies of each packet, and/or through *sequential composition* (`a>>b`), meaning that `a` should be applied to incoming first, and `b` should then be applied to any packet(s) that `a` may produce. For instance, `match(inport=1)>>(fwd(2)|fwd(3))` says that packets that arrive on port 1 should be forwarded to both ports 2 and 3.

Dynamic policies are based on *queries*. A query describes packets or statistics of interest – for instance, packets for which no policy has been defined yet. When a query returns new data, a callback function is invoked that can modify the policy. Figure 5 (taken from [21]) shows a simple self-learning switch that queries for packets with unknown MAC addresses; when a packet with a new source MAC m is observed on port p, the policy is updated to forward future packets with destination MAC m to port p.

Pyretic has other features besides these, and providing comprehensive support for them is beyond the scope of our case study. Here, our goal is to support an interesting subset, to demonstrate that our approach is feasible.

Translation to NDlog: Our Pyretic frontend transforms all static policies into a "normal form" that consists of groups of parallel "atoms" (with a sequence of matches and a single action) that are arranged sequentially. This form easily translates to OpenFlow wildcard entries: we can give the highest priority to the atoms in the first group, and assign further priorities to the following groups in descending order. To match Pyretic's behavior, we do not install the wildcard entries in the switch directly, but rather keep them as base tuples in a special `MacroRule` table in the controller. A second stage then matches incoming packets from the switches against this table, and generates the corresponding microflow entries (without wildcards), which are then sent to the switch.

For each query policy, the frontend creates a separate table and a rule that sends incoming packets to this table if they match the query. The trigger is evaluated using NDlog aggregations; for instance, waiting for a certain number of packets is implemented with NDlog's `count<>` operator.

Our frontend supports one type of dynamic policies: policies that append new logic in response to external events. These are essentially translated to a single NDlog rule that is triggered by the relevant external event (specified as a query policy) and computes

```
MacroRule(@C,sw,inPort0,dstMAC0,act,Prio0) :-
  UpdateEvent(@C,sw,srcMac), HighestP(@C,Prio),
  PktIn(@sw,inPort1,srcMAC,dstMAC1), inPort0=*,
  dstMAC0=srcMAC, act=fwd(inPort1),
  Prio0=Prio1+10
```

Figure 6: NDlog translation of the self-learning switch.

and installs the new entry. The self-learning switch from Figure 5 is an example of such a policy; Figure 6 shows the rule that it is translated to. The rule directly corresponds to the `if-then` part in Figure 5, which forwards packets to newly observed MAC addresses to the correct port, and otherwise (in the `else` branch) falls back on the existing policy. Once translated in this way, it is easy to see the provenance of a dynamic policy change: it is simply the packet that triggered the change.

6.2 BGP debugging

Our second case study focuses on BGP. There is a rich literature on BGP root-cause analysis, and a variety of complex real-world problems have been documented. Here, we focus exclusively on the question whether Y! can be used to diagnose BGP problems with negative symptoms, and we ignore many other interesting questions, e.g., about incentives and incremental deployment. (Briefly, we believe that the required infrastructure and the privacy implications would be roughly comparable to those of [27]; in a partial deployment, some queries would return partial answers that are truncated at the first vertex from a network outside the deployment.)

To apply Y!, we follow the approach from [34] and write a simple declarative program that describes how the BGP control plane makes routing decisions. Our implementation is based on an NDlog encoding of a general path vector protocol provided by the authors of [30]; since this code was generic and contained no specific policies, we extended it by adding simple routing policies that respect the Gao-Rexford guidelines and import/export filters that implements the valley-free constraint. This yielded essentially a declarative specification of an ISP's routing behavior. With this, we could capture BGP message traces from unmodified routers, as described in [34], and infer the provenance of the routing decisions by replaying the messages to our program.

7. EVALUATION

In this section, we report results from our experimental evaluation of Y! in the context of SDNs and BGP. Our experiments are designed to answer two high-level questions: 1) is negative provenance useful for debugging realistic problems? and 2) what is the cost for maintaining and querying negative provenance?

We ran our experiments on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM. The OS was Ubuntu 13.04, and the kernel version was 3.8.0.

7.1 Prototype implementation

For our experiments, we built a prototype of Y! based on the RapidNet declarative networking engine [24]. We instrumented RapidNet to capture provenance for the NDlog programs it executes, and we added provenance storage based on the R-tree implementation from [14]. To experiment with SDNs, we set up networks in Mininet [20]. Since NDlog is not among the supported controllers, we wrote a simple proxy for Trema [29] that translates controller messages to NDlog tuples and vice versa. To capture the provenance of packets flowing through the network, we set up port mirroring on the virtual switches and used `libpcap` to record packet traces on the mirrored ports. Since Y!'s provenance graphs only use the packet headers, we capture only the first 96 bytes of header and its timestamp.

Q1	SDN1	NAPPEAR($[t_1,t_2]$, packet(@D, PROTO=HTTP))
Q2	SDN2	NAPPEAR($[t_1,t_2]$, packet(@D, PROTO=ICMP))
Q3	SDN3	NAPPEAR($[t_1,t_2]$, packet(@D, PROTO=SQL))
Q4	SDN2	APPEAR (t_3, packet(@D, PROTO=ICMP))
Q5	SDN3	APPEAR (t_3, packet(@D, PROTO=SQL))
Q6	BGP1	NAPPEAR($[t_1,t_2]$, bestroute(@$AS2$, TO=AS7))
Q7	BGP2	NAPPEAR($[t_1,t_2]$, packet(@$AS7$, SRC=AS2))
Q8	BGP3	NAPPEAR($[t_1,t_2]$, bestroute(@$AS2$, TO=AS7))
Q9	BGP4	NAPPEAR($[t_1,t_2]$, bestroute(@$AS2$, TO=AS7))

Table 2: Queries we used in our experiments

To demonstrate that our approach does not substantially affect throughput and latency, we also built a special Trema extension that can capture the provenance directly, without involving RapidNet. This extension was used for some of experiments in Section 7.5, as noted there. Other than that, we focused more on functionality and complexity than on optimizing performance; others have already shown that provenance can be captured at scale [15], and the information Y! records is not substantially different from theirs – Y! merely uses it in a different way.

7.2 Usability: SDN debugging

For our SDN experiments, we used Mininet to create the following three representative SDN scenarios:

- **SDN1: Broken flow entry.** A server receives no requests because an overly general flow entry redirects them to a different server (taken from Section 2.3).
- **SDN2: MAC spoofing.** A host, which is connected to the self-learning switch from Figure 5, receives no responses to its DNS lookups because another machine has spoofed its MAC address.
- **SDN3: Incorrect ACL.** A firewall, intended to allow Internet users to access a web server W and internal users a database D, is misconfigured: Internet users can access only D, and internal users only W.

Each scenario consists of four hosts and three switches. For all three scenarios, we use Pyretic programs that have been translated to NDlog rules (Section 6.1) and are executed on RapidNet; however, we verified that each problem also occurs with the original Pyretic runtime. Note that, in all three scenarios, positive provenance cannot be used to diagnose the problem because there is no state whose provenance could be queried.

The first three queries we ask are the natural ones in these scenarios: in SDN1, we ask why the web server is not receiving any requests (Q1); in SDN2, we ask why there are no responses to the DNS lookups (Q2); and in SDN3, we ask why the internal users cannot get responses from the database (Q3). To exercise Y!'s support for positive provenance, we also ask two positive queries: why a host in SDN2 *did* receive a certain ICMP packet (Q4), and why the internal database is receiving connections from the Internet (Q5). To get a sense of how useful negative provenance would be for debugging realistic problems in SDNs, we ran diagnostic queries in our three scenarios and examined the resulting provenance. The first five rows in Table 2 show the queries we used. The full responses are in Appendix B; here, we focus on Q1 from scenario SDN1, which asks why HTTP requests are no longer appearing at the web server.

Figure 7 shows the provenance generated by Y! for Q1. The explanation reads as follows: HTTP requests did not arrive at the HTTP server (V1) because there was no suitable flow entry at the switch (V2). Such an entry could only have been installed if a

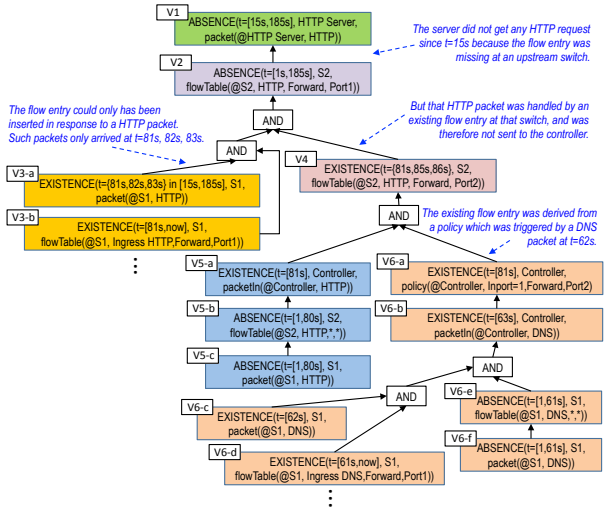


Figure 7: Answer to Q1, as returned by Y!

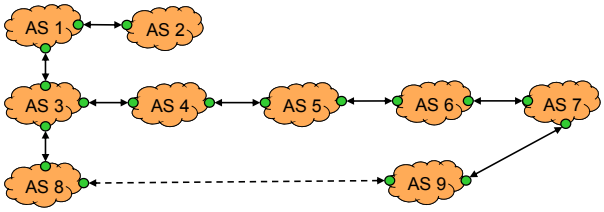


Figure 8: Topology for the BGP1 scenario.

HTTP packet had arrived (V3a+b) and caused a table miss, but the latter did not happen because there already was an entry – the low-priority entry (V4) – that was forwarding HTTP packets to a different port (V5a-c), and that entry had been installed in response to an earlier DNS packet (V6a-f). We believe that "backtraces" of this kind would be useful in debugging complex problems.

7.3 Usability: BGP debugging

For our BGP experiments, we picked four real BGP failure scenarios from our survey (Section 2.4):

- **BGP1: Off-path change.** In the topology from Figure 8, AS 2 initially has a route to AS 7 via AS 1,3,4,5,6, but loses that route when a new link is added between AS 8 and AS 9 (neither of which is on the path). This is a variant of a scenario from [27].
- **BGP2: Black hole.** A buggy router advertises a spurious `/32` route to a certain host, creating a "black hole" and preventing that host from responding to queries.
- **BGP3: Link failure.** An ISP temporarily loses connectivity, due to a link failure at one of its upstream ASes.
- **BGP4: Bogon list.** A network cannot reach a number of local and federal government sites from its newly acquired IP prefix because that prefix was on the bogon list earlier.

We set up small BGP topologies, using between 4 and 18 simulated routers, to recreate each scenario. In each scenario, we then asked one query: why AS 2 in scenario BGP1 has no route to AS 7 (Q6), why a host in scenario BGP2 cannot reach the black-holed host (Q7), why the ISP in scenario BGP3 cannot reach a certain AS (Q8), and why the network in scenario BGP4 cannot connect

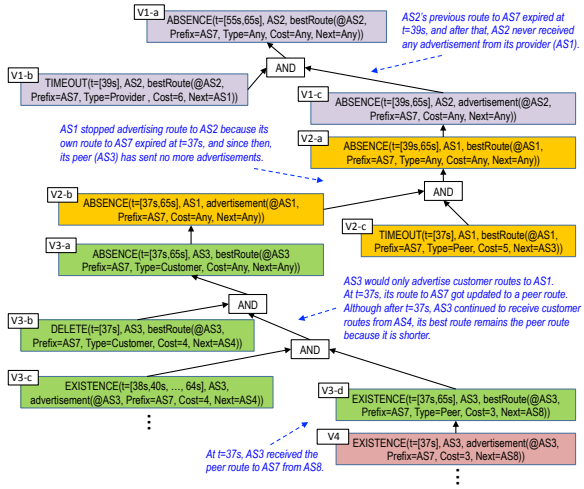


Figure 9: Answer to Q6, as returned by Y!

Figure 10: Size of the provenance with some or all heuristics disabled.

to a particular site (Q9). Table 2 shows the specific queries. As expected, Y! generated the correct response in all four scenarios; here, we focus on one specific query (Q6/BGP1) due to lack of space. The other results (available in Appendix C) are qualitatively similar.

Figure 9 shows the provenance generated by Y! for query Q6. The explanation reads as follows: AS 2 has no route to AS 7 (V1-a) because its previous route expired (V1-b) and it has not received any new advertisements from its provider AS 1 (V1-c). This is because AS 1 itself has no suitable route: its peer AS 3 stopped advertising routes to AS 7 (V2a-c) because AS 3 only advertises customer routes to AS 7 due to the valley-free constraint (V3-a). AS 3 previously had a customer route but it disappeared (V3-b). Although AS 3 continues to receive the customer route from AS 4 (V3-c), the peer route through AS 8 (V4) is preferred because it has a shorter AS path (V3-d). The provenance of the peer route could be further explored by following the graph beyond V4.

7.4 Complexity

Recall from Section 4 that Y! uses a number of heuristics to simplify the provenance before it is shown to the user. To quantify how well these heuristics work, we re-ran the queries in Table 2 with different subsets of the heuristics disabled, and we measured the size of the corresponding provenance graphs.

Figure 10 shows our results. Without heuristics, the provenance contained between 55 and 386 vertices, which would be difficult for a human user to interpret. The pruning heuristics from Section 4.1 generally remove about half the vertices, but the size of the

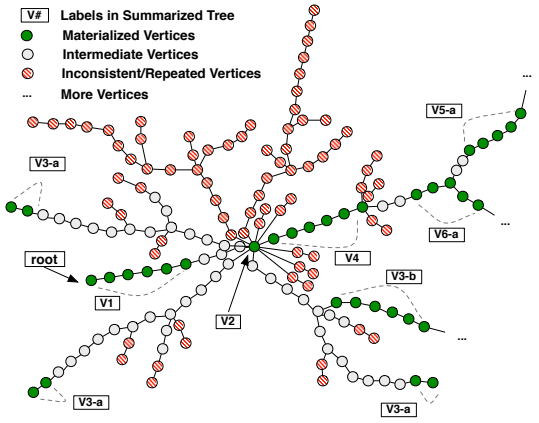


Figure 11: Raw provenance for query Q1 before post-processing.

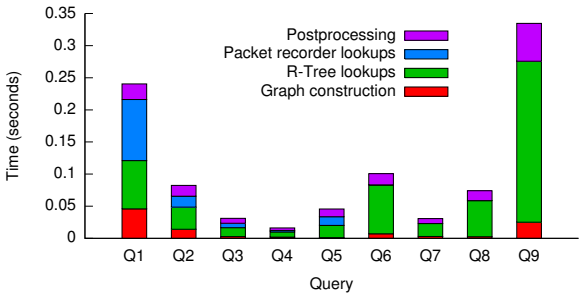


Figure 12: Turnaround time for the queries in Table 2.

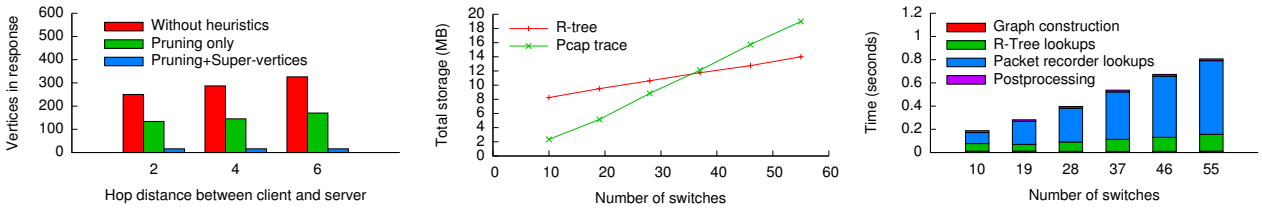
provenance remains substantial. However, the super-vertices from Section 4.2 are able to shrink the provenance considerably, to between 4 and 24 vertices, which should be much easier to interpret.

To explain where the large reduction comes from, we show the raw provenance tree (without the heuristics) for Q1 in Figure 11. The structure of this tree is typical of the ones we have generated: a "skeleton" of long causal chains, which typically correspond to messages and events propagating across several nodes, and a large number of small branches. The pruning heuristics remove most of the smaller branches, while the super-vertices "collapse" the long chains in the skeleton. In combination, this yields the much-simplified tree from Figure 7.

7.5 Run-time overhead

Disk storage: Y! maintains two data structures on disk: the packet traces and the historical R-tree. The size of the former depends on the number of captured packets; each packet consumes 120 bytes of storage. To estimate the size of the latter, we ran a program that randomly inserted and removed `flowEntry` tuples, and we measured the number of bytes per update. We found that, for trees with 10^3 to 10^6 updates, each update consumed about 450 byte of storage on average.

These numbers allow us to estimate the storage requirements in a production network. We assume that there are 400 switches that each handle 45 packets per second, and that the SDN controller generates 1,200 flow entries per second. Under these assumptions, a commodity hard disk with 1TB capacity could easily hold the provenance for the most recent 36 hours. If necessary, the storage cost could easily be reduced further, e.g., by compressing the data, by storing only a subset of the header fields, and/or by removing redundant copies of the headers in each flow.



(a) Size of query results (b) Use of storage space (c) Query turnaround time

Figure 13: Scalability results

Latency and throughput: Maintaining provenance requires some additional processing on the SDN controller, which increases the latency of responses and decreases throughput. We first measured this effect in our prototype by using `Cbench` to send streams of PacketIn messages, which is a current standard in evaluating OpenFlow controllers [4]. We found that the 95th percentile latency increased by 29%, from 48.38 ms to 62.63 ms, when Y! was enabled; throughput dropped by 14%, from 56.0 to 48.4 requests per second.

However, these results are difficult to generalize because RapidNet's performance as an SDN controller is not competitive with state-of-the-art controllers, even without Y!. We therefore repeated the experiment with our Y! extension for native Trema (Section 7.1); here, adding Y! increased the average latency by only 1.6%, to 33 microseconds, and decreased the average throughput by 8.9%, to 100,540 PacketIn messages per second. We note that this comparison is slightly unfair because we manually instrumented a specific Trema program to work with our extension, whereas the RapidNet prototype can work with any program. However, adding instrumentation to Trema programs is not difficult and could be automated. More generally, our results suggest that capturing provenance is not inherently expensive, and that an optimized RapidNet could potentially do a lot better.

7.6 Query processing speed

When the user issues a provenance query, Y! must recursively construct the response using the process from Section 3.5 and then post-process it using the heuristics from Section 4. Since debugging is an interactive process, a quick response is important. To see whether Y! can meet this requirement, we measured the turnaround time for the queries in Table 2, as well as the fraction of time consumed by Y!'s major components.

Figure 12 shows our results. We make two high-level observations. First, the turnaround time is dominated by R-tree and packet recorder lookups. This is expected because the graph construction algorithm itself is not very complex. Second, although the queries vary in complexity and thus their turnaround times are difficult to compare, we observe that none of them took more than one second; the most expensive query was Q9, which took 0.33 seconds to complete.

7.7 Scalability

We do not yet have experience with Y!, or negative provenance, in a large-scale deployment. However, we have done a number of experiments to get an initial impression of its scalability. Due to lack of space, we report only a subset of our results here.

Complexity: In our first experiment, we tested whether the complexity of the provenance increases with the number of possible traffic sources. We simulated a four-layer fat-tree topology with 15 switches, and we placed the client and the server on different leaves, to vary the hop distance between them from 2 to 6. Our results for running the learning-switch query (Q1) are shown in Figure 13a (the bars are analogous to Figure 10). As expected, the size of the raw provenance for Q1 grew substantially – from 250 to 386 vertices – because 1) there number of possible sources for the missing traffic increased, because each additional hop brings additional branches on the backtrace path and 2) each additional hop required extra vertices to be represented in the provenance. But the first effect was mitigated by our pruning heuristics, since the extra sources were inconsistent with the network state, and the second effect was addressed by the summarization, which merged the vertices along the propagation path into a single super-vertex. Once all the heuristics had been applied, the size of the provenance was 16 vertices, independent of the hop count.

Storage: In our second experiment, we simulated three-layer fat-tree topologies of different sizes (i.e., with different node degrees); each edge switch was connected to a fixed number of active hosts that were constantly sending HTTP requests to the server. Figure 13b shows how Y!'s storage requirements grew with the number of switches in the network. As expected, the size of both the `pcap` trace and the R-tree was roughly proportional to the size of the network; this is expected because a) each new switch added a fixed number of hosts, and b) the depth of the tree, and thus the hop count between the server and its clients, remained constant. Generally, the storage requirement depends on the rate at which events of interest (packet transmissions, routing changes, etc.) are captured, as well as on the time for which these records are retained.

Query speed: Our third experiment is analogous to the second, except that we issued a query at the end and measured its turnaround time. Figure 13c shows our results. The dominant cost was the time it took to find packets in the `pcap` trace; the R-tree lookups were much faster, and the time needed to construct and post-process the graph was so small that it is difficult to see in the figure. Overall, the lookup time was below one second even for the largest network we tried.

Possible optimizations: Since our implementation has not been optimized, some of the costs could grow quickly in a large-scale deployment. For instance, in a data center with 400 switches that handle 1 Gbps of traffic each, our simple approach of recording `pcap` traces at each switch would consume approximately 30 GB of storage *per second* for the date center, or about 75 MB for each switch. Packet recorder lookups, which compromise a major portion of query latency, in such a large trace would be limited by disk read throughput, and could take minutes. However, we note that there are several ways to reduce these costs; for instance, techniques from the database literature – e.g., a simple time index – could be used to speed up the lookups, and the storage cost could be reduced by applying filters.

7.8 Summary

Our results show that Y! – and, more generally, negative provenance – can be a useful tool for diagnosing problems in networks: the provenance of the issues we looked at was compact and read-

able, and Y! was able to find it in less than a second in each case. Our results also show that the readability is aided considerably by Y!'s post-processing heuristics, which reduced the number of vertices by more than an order of magnitude. Y!'s main run-time cost is the storage it needs to maintain a history of the system's past states, but a commodity hard-drive should be more than sufficient to keep this history for more than a day.

8. RELATED WORK

Network debugging: Many tools and techniques for network debugging and root cause analysis have been proposed, e.g., [5, 9, 18, 26], but most focus on explaining positive events. These tools can be used to (indirectly) troubleshoot negative symptoms, but the results from our survey in Section 2.4 suggest that the lack of direct support for negative queries makes this significantly more difficult. Hubble [12] uses probing to find AS-level reachability problems but is protocol-specific; header space analysis [13] provides finer-grain results but relies on static analysis and thus cannot explain complex, dynamic interactions like the ones we consider here. ATPG [33] tests for liveness, reachability, and performance, but cannot handle dynamic nodes like the SDN controller. NICE [2] uses model checking to test whether a given SDN program has specific correctness properties; this approach is complementary to ours, which focuses on diagnosing unforeseen problems at runtime. We are not aware of any protocol-independent systems that can explain negative events in a dynamic distributed system.

Negative provenance: There is a substantial literature on tracking provenance in databases [1, 6, 11, 25, 31] and in networks [34, 36], but only a few papers have considered negative provenance. Huang et al. [10] and Meliou et al. [19] focus on instance-based explanations for missing answers, that is, how to obtain the missing answers by making modifications to the value of base instances (tuples); Why-Not [3] and ConQueR [28] provide query-based explanations for SQL queries, which reveal over-constrained conditions in the queries and suggest modifications to them. None of these papers considers distributed environments and networks, as we do here. In the networking literature, there is some prior work on *positive* provenance, including our own work on ExSPAN [36], SNP [34], and DTaP [35], but none of these systems can answer (or even formulate) negative queries. To our knowledge, the only existing work that does support such queries is our own workshop paper [32], on which this paper is based.

9. CONCLUSION

In this paper, we have argued that debuggers for distributed systems should not only be able to explain why an unexpected event *did* occur, but also why an expected event *did not* occur. We have shown how this can be accomplished with the concept of *negative provenance*, which so far has received relatively little attention. We have defined a formal model of negative provenance, we have presented an algorithm generating such provenance, and we have introduced Y!, a practical system that can maintain both positive and negative provenance in a distributed system and answer queries about it. Our evaluation in the context of software-defined networks and BGP suggests that negative provenance can be a useful tool for diagnosing complex problems in distributed systems.

Acknowledgments: We thank our shepherd Nate Foster and the anonymous reviewers for their comments and suggestions. We also thank Behnaz Arzani for helpful comments on earlier drafts of this paper. This work was supported by NSF grants CNS-1065130, CNS-1054229, CNS-1040672, and CNS-0845552, as well as DARPA contract FA8650-11-C-7189.

10. REFERENCES

[1] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, Jan. 2001.
[2] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test OpenFlow applications. In *Proc. NSDI*, Apr. 2012.
[3] A. Chapman and H. V. Jagadish. Why not? In *Proc. SIGMOD*, June 2009.
[4] D. Erickson. The Beacon OpenFlow controller. In *Proc. HotSDN*, Aug. 2013.
[5] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. SIGCOMM*, Aug. 2004.
[6] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *Proc. SIGMOD*, June 2007.
[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD*, June 1984.
[8] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *Proc. HotSDN*, Aug. 2012.
[9] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. NSDI*, Apr. 2014.
[10] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, 1(1):736–747, Aug. 2008.
[11] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *Proc. CIDR*, Jan. 2011.
[12] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proc. NSDI*, Apr. 2008.
[13] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, Apr. 2012.
[14] libspatialindex. `http://libspatialindex.github.io/`.
[15] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging DISC analysis. Technical Report CSE2012-0990, UCSD.
[16] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, Nov. 2009.
[17] P. Macko and M. Seltzer. Provenance Map Orbiter: Interactive exploration of large provenance graphs. In *Proc. TaPP*, June 2011.
[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. SIGCOMM*, Aug. 2011.
[19] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *Proc. SIGMOD*, May 2012.
[20] Mininet. `http://mininet.org/`.
[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, Apr. 2013.
[22] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, May 2006.
[23] Outages mailing list. `http://wiki.outages.org/index.php/Main_Page#Outages_Mailing_Lists`.
[24] RapidNet. `http://netdb.cis.upenn.edu/rapidnet/`.
[25] C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. ICDE*, Apr. 2007.
[26] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *Proc. EuroSys*, Apr. 2006.
[27] R. Teixeira and J. Rexford. A measurement framework for pin-pointing routing changes. In *Proc. Network Troubleshooting workshop (NetTS)*, Sept. 2004.
[28] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *Proc. SIGMOD*, June 2010.
[29] Trema. `http://trema.github.io/trema/`.
[30] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. L. Talcott. FSR: Formal analysis and implementation toolkit for safe inter-domain routing. *IEEE/ACM ToN*, 20(6):1814–1827, Dec. 2012.
[31] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. CIDR*, Jan. 2005.
[32] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering Why-Not queries in software-defined networks with negative provenance. In *Proc. HotNets*, 2013.
[33] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, Dec. 2012.
[34] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, Oct. 2011.
[35] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proc. VLDB*, Aug. 2013.
[36] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD*, 2010.

APPENDIX

A. FORMAL MODEL

A.1 Background: Execution Traces

To set stage for the discussion, we introduce some necessary definitions of system execution. The execution of an NDlog program can be characterized by the sequence of events that take place; we refer to this sequence as an *execution trace*. More formally, an execution trace is defined as follows.

Definition (Event)*: An event $d@n = (m, r, t, c, m')$ represents that rule r was triggered by message m and generated a set of (local or remote) messages m' at time t, given the precondition c. The precondition c is a set of tuples that existed on n at time t.*

Definition (Trace)*: A trace $\mathcal{E}$ of a system execution is an ordered sequence of events from an initial state $\mathcal{S}_0$, $\mathcal{S}_0 \xrightarrow{d_1@n_1} \mathcal{S}_1 \xrightarrow{d_2@n_2} ... \xrightarrow{d_k@n_k} \mathcal{S}_k$.*

Definition (Subtrace): *We say that $\mathcal{E}'$ is a subtrace of $\mathcal{E}$ (written as $\mathcal{E}' \subseteq \mathcal{E}$) if $\mathcal{E}'$ consists of a subset of the events in $\mathcal{E}$ in the same order. In particular, we write $\mathcal{E}|n$ to denote the subtrace that consists of all the events on node n in $\mathcal{E}$.*

Definition (Equivalence): *We say that two traces $\mathcal{E}'$ and $\mathcal{E}$ are equivalent (written as $\mathcal{E}' \sim \mathcal{E}$), if, for all n, $\mathcal{E}'|n = \mathcal{E}|n$.*

A.2 Properties

Given the negative provenance of $G(e, \mathcal{E})$ of the absence of an event e in execution trace $\mathcal{E}$, we formally define the following properties.

Definition (Validity)*: We say a trace $\mathcal{E}$ is valid, if (a) for all $\tau_k \in c_i$, $\tau_k \in S_{i-1}$, and (b) for all $d_i@n_i = (m_i, r_i, t_i, c_i, m_i')$, either m_i is a base message from an external source, or there exists $d_j@n_j = (m_j, r_j, t_j, c_j, m_j')$ that proceeds $d_i@n_i$ and $m_i \in m_j'$.*

Property (Soundness): *Negative provenance $G(e, \mathcal{E})$ is sound iff (a) it is possible to extract a valid subtrace $\mathcal{E}_{sub} \subseteq \mathcal{E}'$, such that $\mathcal{E}' \sim \mathcal{E}$, and (b) for all vertices in $G(e, \mathcal{E})$, their corresponding predicates hold in $\mathcal{E}$.*

Property (Completeness): *Negative provenance $G(e, \mathcal{E})$ is complete iff there exists no trace $\mathcal{E}'$ such that a) $\mathcal{E}'$ assumes same external inputs as $G(e, \mathcal{E})$, that is, $(\Delta\tau@n, -, t, -, -) \in \mathcal{E}$ iff* INSERT/DELETE$(\tau, n, t) \in V(G)$*, and b) e exists in the $\mathcal{E}'$.*

Definition (Reduction): *Given negative provenance $G(e, \mathcal{E})$, if there exists $v_1, v_2 \in V(G)$, where the time interval of v_1 and v_2 ($t(v_1)$ and $t(v_2)$ respectively) are adjacent, and v_1 and v_2 have the same dependencies, then G can be reduced to G' by combining v_1 and v_2 into $v \in V(G')$, where $t(v) = t(v_1) \cup t(v_2)$.*

Definition (Simplicity): *Given two negative provenance $G(e, \mathcal{E})$ and $G'(e, \mathcal{E})$, we say G' is simpler than G (written as $G' < G$), if any of the following three hold:*
(1) G' is a subgraph of G;
(2) G' is reduced from G (by combining v_1 and v_2);
(3) There exists G'', such that $G' < G''$ and $G'' < G$.

Property (Minimality): *Negative provenance $G(e, \mathcal{E})$ is minimal, if no $G' < G$ is sound and complete.*

A.3 Proofs

Definition (Trace Extraction): *Given a negative provenance $G(e, \mathcal{E})$, the trace is extracted by running Algorithm 1 based on topological sort.*

Algorithm 1 converts *positive* vertices in the provenance graph to events and thus uses a topological ordering to assemble the events into a trace. In particular, Line 9-19 implements the construction of one individual event, where the information of a rule evaluation (such as triggering event, conditions, and action) is extracted from the corresponding vertices in $G(e, \mathcal{E})$

Algorithm 1 Extracting traces from provenance

```
1: proc ExtractTrace(G = (V, E))
2: // calculate the out-degree of all vertices in G
3: for ∀ e = (v, v') ∈ E do degree(v)++
4: // generate the subtrace based on topological sort
5: NodeToProcess = V
6: while NodeToProcess ≠ φ do
7:     select v ∈ NodeToProcess : degree(v) = 0, and ∄v' that is located
       on the same node and has a larger timestamp
8:     NodeToProcess.remove(v)
9:     if typeof(v) = DERIVE or UNDERIVE then
10:        find e = (v, v') ∈ E, output ← v'
11:        for ∀ e = (v', v) ∈ E do
12:            if typeof(v') = INSERT or DELETE then
13:                trigger = v' // v' is the triggering event
14:            else
15:                condition.add(v') // v' is a condition
16:        ruleName ← v.ruleName, time ← v.time
17:        event ← (trigger, ruleName, time, condition, output)
18:        trace.push_front(event)
19:    else if typeof(v) = RECEIVE then
20:        find e = (v', v) ∈ E // v' must be a SEND vertex
21:        event ← (v', −, v.time, 1, v)
22:        trace.push_font(event)
23:    find e = (v', v) ∈ E, degree(v') ← degree(v') − 1
24: return trace
```

LEMMA 1. *For any vertex $v \in V(G)$, v's corresponding predicate p_v holds in $\mathcal{E}$.*

Proof. By construction (see Figure 3), any positive vertex $v \in V(G)$ is generated only if the corresponding event entry exists in the log (which is deterministically determined by $\mathcal{E}$. Therefore, predicate p_v naturally holds in $\mathcal{E}$. Similarly, the predicate p_w (for negative vertex w) also hold in $\mathcal{E}$. □

LEMMA 2. *The extracted trace $\mathcal{E}_{sub}$ is a valid trace.*

Proof. We need to show that in the generated trace $\mathcal{S}_0 \xrightarrow{d_1@n_1} \mathcal{S}_1 \xrightarrow{d_2@n_2} ... \xrightarrow{d_k@n_k} \mathcal{S}_k$, (a) for all $\tau \in c_i$, $\tau \in S_{i-1}$, and (b) for all $d_i@n_i = (m_i, r_i, t_i, c_i, m_i')$, either m_i is a base message from an external source, or there exists $d_j@n_j = (m_j, r_j, t_j, c_j, m_j')$ that precedes $d_i@n_i$ and $m_i \in m_j'$.

It has been shown in [35] that, for a positive event e' in $\mathcal{E}$, the extraction algorithm yields a valid trace for the provenance of e' (which is the subgraph G' of G that is rooted by e'). Note that executing Algorithm 1 on G can be considered as combining traces generated from multiple such G's, $\{d_{i,1}@n_{i,1}\}, \{d_{i,2}@n_{i,2}\}, ..., \{d_{i,k}@n_{i,k}\}$, where each trace $\{d_{i,p}@n_{i,p}\}$ is valid.

Condition a. We know that, for any event $d_{i,p}@n_{i,p}$, the conditions for the event $c_{i,p}$ hold (i.e., the corresponding tuples exist when the rule is triggered). It can be shown that, in the combined trace, any event $d@n$ that contributes to $c_{i,p}$ precedes $d_{i,p}@n_{i,p}$;

this is because $d@n$ has to be a (transitive) children of $d_{i,p}@n_{i,p}$ in G, and, therefore, should also be in trace p. We can then conclude that, in the combined trace $d_i@n_i$, the conditions for the event c_i hold when the event is triggered (i.e., condition b holds).

Condition b. Because any trace $\{d_{i,p}@n_{i,p}\}$ is valid, we know that, for any event $d_{i,p}@n_{i,p} = (m_{i,p}, r_{i,p}, t_{i,p}, c_{i,p}, m'_{i,p})$, the trigger event $m_{i,p}$ was indeed generated by a preceding event $d_{j,p}@n_{j,p}$. This also holds in the combined trace (i.e., condition a holds), as the topological relationship between $d_{i,p}@n_{i,p}$ and $d_{j,p}@n_{j,p}$ remains. □

LEMMA 3. *The extracted trace $\mathcal{E}_{sub} \subseteq \mathcal{E}'$, where $\mathcal{E}' \sim \mathcal{E}$.*

Proof. We need to show that a) all the events in $\mathcal{E}_{sub}$ also appear in $\mathcal{E}$ (and thus in any $\mathcal{E}' \sim \mathcal{E}$), and b) the local event ordering pertains on each node.

Condition a. An event $d_i@n_i$ is generated and included in $\mathcal{E}_{sub}$ for each DERIVE (or UNDERIVE) vertex (and its direct parent and children) in $G(\Delta\tau, \mathcal{E})$. On the other hand, by construction (Figure 3), each DERIVE (or UNDERIVE) vertex v corresponds to an event $d_j@n_j$, and it is not difficult to see that $d_i@n_i$ is identical to $d_j@n_j$. Therefore, all the events in $\mathcal{E}_{sub}$ do appear in $\mathcal{E}$ as well.

Condition b. According to Algorithm 1 (specifically, Line 7), $d_1@n$ precedes $d_2@n$, iff $d_2@n$ has a larger timestamp than $d_1@n$ (that is, $d_1@n$ precedes $d_2@n$ in the actually execution $\mathcal{E}$. Note that events on different nodes may be reordered, but this is captured by the equivalence ($\sim$) relation. □

THEOREM 4. *Negative provenance $G(\Delta\tau, \mathcal{E})$ is sound.*

Proof. Directly from Lemma 1, 2, 3. □

THEOREM 5. *Negative provenance $G(\Delta\tau, \mathcal{E})$ is complete.*

Proof. We prove the completeness property by induction on the depth of the provenance graph.

Base case. In the base case, the negative provenance $G(e, \mathcal{E})$ is a single-vertex graph (i.e., NINSERT$([t1, t2], N, \tau)$ or NDELETE$([t1, t2], N, \tau)$). In this case, e is an update of a base tuple τ, and it is obvious that there exists no trace $\mathcal{E}'$ that has the same external input and contains $\Delta\tau$.

Induction case. Suppose there exists $\mathcal{E}'$, such that $\mathcal{E}'$ assumes the same external input as $G(e, \mathcal{E})$, and e exists in $\mathcal{E}'$. We perform a case analysis by considering the type of the root vertex of $G(e, \mathcal{E})$.

- NEXIST$([t1, t2], N, \tau)$. Suppose τ existed (partly) between $[t1', t2']$ in $\mathcal{E}'$, where $[t1, t2] \cap [t1', t2'] = [t3, t4] \neq \phi$. By construction (in Figure 3), NAPPEAR$([t0, t2], N, \tau)$ is in G, where $t0 \leq t1$. We can conclude, based on induction hypothesis, that there is no execution that assumes same external input as G but has τ appeared between $[t0, t2]$. This contradicts the existence of τ in $\mathcal{E}'$.
- NAPPEAR$([t1, t2], N, \tau)$. Suppose τ appeared at $t \in [t1, t2]$ in $\mathcal{E}'$. τ cannot be a base tuple, otherwise, G and $\mathcal{E}'$ would have conflicting external inputs. If τ is a locally derived tuple, τ was derived by some rule execution in $\mathcal{E}'$, however, for any rule r that might derive τ, NDERIVE$([t1, t2], N, \tau, r)$ is in G; a contradiction is reached by applying the induction hypothesis. If τ is received from communication, message $+\tau$ was received at time t in $\mathcal{E}'$, whereas NRECEIVE$([t1, t2], N, +\tau)$ is in G; a contradiction is reached by applying the induction hypothesis.
- NDERIVE$([t1, t2], N, \tau, r)$. Suppose τ was derived at $t \in [t1, t2]$ by rule r in $\mathcal{E}'$. It must be the case that all the condition held at time t (that is, their corresponding tuples $c_1 ... c_k$ existed at t). On the other hand, for each rule (including rule r) that might derive τ, the graph construction algorithm considers the parameter space in which τ would not be derived, and includes a set of negative events that cover the whole parameter space.

  Since G contains the cover set, there exists c_i such that NEXIST$([t1', t2'], N, c_i)$ exists in G, where $t \in [t1', t2']$. Based on induction hypothesis, there is no execution that assumes same external input as G but has c_i existed time t. This contradicts the rule evaluation that derived τ at time t in $\mathcal{E}'$.
- NSEND$([t1, t2], N, \Delta\tau)$. Suppose $\Delta\tau$ was sent at $t \in [t1, t2]$ in $\mathcal{E}'$. It must be the case that τ appeared (or disappeared) at time t in $\mathcal{E}'$. However, by construction, NAPPEAR$([t1, t2], N, \tau)$ (or NDISAPPEAR$([t1, t2], N, \tau)$) is in G, which, based on induction hypothesis, indicates that there is no execution that assumes same external input as G but has τ appeared (or disappeared) between $[t1, t2]$. This contradicts the appearance (or disappearance) of τ in $\mathcal{E}'$.
- NRECEIVE$([t1, t2], N, \Delta\tau)$. Suppose message $\Delta\tau$ was received at time $t \in [t1, t2]$ in $\mathcal{E}'$. It must be the case that $\Delta\tau$ was sent by some node M at time t_{send}, and arrived N at time $t_{recv} \in [t1, t2]$. On the other hand, the graph construction algorithm considers all nodes (including M) that could have sent $\Delta\tau$ to N. Specifically, for node M, NSEND$([s, t], M, \Delta\tau)$ is constructed for any time interval $[s, t]$ in which M did not send $\Delta\tau$ in E.

  If $t_{send} \in [s, t]$, a contradiction is reached by applying the induction hypothesis. Otherwise (M did send $\Delta\tau$ at t_{send} in E), NARRIVE$([t1, t2], M \rightarrow N, t_{send}, \Delta\tau)$ is in G. Based on the induction hypothesis, there exists no execution that assumes same external input as G but has $\Delta\tau$ received at N between $[t1, t2]$. Contradiction.
- NARRIVE$([t1, t2], N_1 \rightarrow N_2, t0, \Delta\tau)$. Since execution trace $\mathcal{E}'$ agrees with G on all external inputs (including how the network affects the message delivery), they must agree on the delivery time t of message event $(\Delta\tau@N_1, -, t, 1, \Delta\tau@N_2)$. Therefore, it cannot be the case that message $\Delta\tau$ arrived within $[t1, t2]$ in $\mathcal{E}'$ but not in $\mathcal{E}$.

The case analysis of vertices NDISAPPEAR and NUNDERIVE is analogous to the those of NAPPEAR and NDERIVE respectively; the case analysis for positive vertices have been shown in [35]. □

THEOREM 6. *Negative provenance $G(e, \mathcal{E})$ is minimal.*

Proof. By construction, $G(e, \mathcal{E})$ should not be further reducible; this is because the graph construction algorithm (described in Section 3.5) considers semantically identical vertices with adjacent or overlapping interval, and coalesces these vertices (i.e., apply reduction on G). We next prove, by induction on the depth of $G(e, \mathcal{E})$, that there is no strict subgraph of $G(e, \mathcal{E})$ that is sound and complete.

Base case. In the base case, the negative provenance $G(e, \mathcal{E})$ is a single-vertex graph (i.e., NINSERT$([t1, t2], N, \tau)$ or NDELETE$([t1, t2], N, \tau)$). In this case, a strict subgraph of G (an empty graph) is not complete.

Induction case. Assume the root vertex v of $G(e, \mathcal{E})$ has children vertices $v_1, v_2, ..., v_k$. We write G_{v_i} to denote the subgraphs rooted by children vertex v_i. Based on the induction hypothesis, any strict subgraph of G_{v_i} is either not sound or not complete. Therefore, any sound and complete provenance that contains v_i must contain the complete G_{v_i}. We perform a case analysis by considering the type of the root vertex v.

- NEXIST$([t1, t2], N, \tau)$. NEXIST has children vertices NAPPEAR and (potentially) DISAPPEAR (if τ existed previously and disappeared at time $t < t1$). NAPPEAR$([t, t2], N, \tau)$ is essential for the completeness of G; otherwise, an execution $\mathcal{E}'$ in which τ appeared in $[t, t2]$ would have τ existed during this time interval (that is, the subgraph without NAPPEAR is not complete). The DISAPPEAR vertex is also essential; this is because τ could have already existed before $t1$, in which case, τ would still exist in $[t1, t2]$ though it did not appear again.
- NAPPEAR$([t1, t2], N, \tau)$. If τ is a base tuple, NAPPEAR has child vertex NINSERT, which is essential for the completeness of G; otherwise, execution $\mathcal{E}'$ can have τ inserted, and therefore appear, at $t \in [t1, t2]$. Similarly, tuples that are locally derived or received from network, we can also show NAPPEAR's child vertex NDERIVE (or NRECEIVE) are essential for the completeness.
- NDERIVE$([t1, t2], N, \tau)$. NDERIVE has a set of NEXIST as its children vertices. These NEXIST vertices consist of a cover set S for the parameter space in which τ would not be derived. Recall from Section 3.6 that the PARTITION algorithm that generates S ensures the minimality of S, that is, any strict subset of S is not a cover set. Therefore, each of the NEXIST vertices is essential for completeness.
- NSEND$([t1, t2], N, +\tau)$. NSEND has children vertices NAPPEAR and (potentially) EXIST (if τ existed partly in $[t1, t2]$ and disappeared at time $t \in [t1, t2]$). NAPPEAR$([t, t2], N, \tau)$ is essential for the completeness of G; otherwise, an execution $\mathcal{E}'$ in which τ appeared in $[t, t2]$ would have sent $+\tau$ during this time interval. The EXIST$([t1, t], N, \tau)$ vertex is also essential; this is because there exists execution E' in which $+\tau$ was derived in $[t1, t]$, which could have resulted in a state change, and, therefore, a message sent from N (it didn't, in $\mathcal{E}$, only because τ was already existed).
- NRECEIVE$([t1, t2], N, \Delta\tau)$. NRECEIVE has children vertices NSEND and NARRIVE, both of which are essential for completeness; this is because there exists execution E' in which some node N' has sent $\Delta\tau$ that arrived at node N within $[t1, t2]$.
- NARRIVE$([t1, t2], N_1 \rightarrow N_2, t0, \Delta\tau)$. NARRIVE has children vertices SEND and DELAY, both of which are necessary for the completeness of G; otherwise, there exists an execution $\mathcal{E}'$ in which $\Delta\tau$ was never sent by N_1 or $\Delta\tau$ has been delivered to N_2 between $[t1, t2]$.

The case analysis of vertices NDISAPPEAR and NUNDERIVE is analogous to the those of NAPPEAR and NDERIVE respectively. □

B. RESPONSES

Q2. Q2 from scenario SDN2 asks why a host cannot receive ICMP relies from the DNS server.

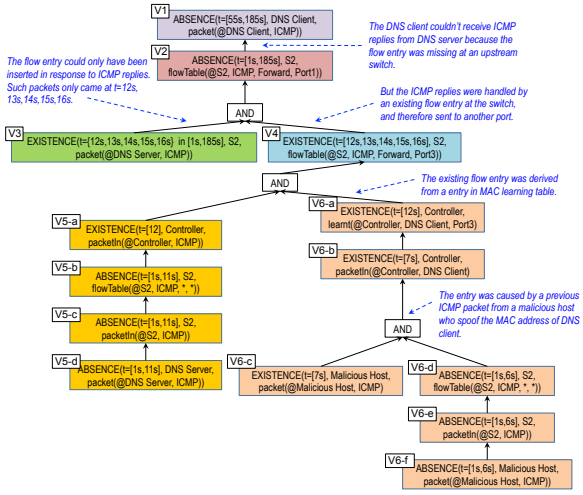


Figure 14: Answer to Q2, as returned by Y!

Figure 14 shows the provenance generated by Y!. The explanation is: ICMP replies did not return to the DNS client (V1) because a suitable flow entry was missing at an upstream switch (V2). The flow entry could only have been triggered by a ICMP reply packet, which did arrive several times (V3). But these replies were handled by an existing flow entry which forwarded the replies to another port (V4). This flow entry was derived from the MAC learning table (V6a) when packetIn arrived (V5a-d). The false MAC learning table was there because a malicious host with spoofed MAC address sent an ICMP packet earlier (V6c).

Q3. Q3 from scenario SDN3 asks why an internal user cannot access the database.

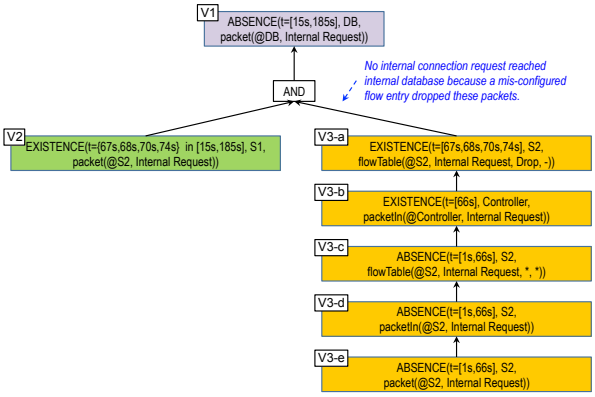


Figure 15: Answer to Q3, as returned by Y!

As shown in Figure 15, Y! explains the problem (V1) by showing that an existing flow entry (V3a-e) had been dropping internal database requests (V2).

Q4. Q4 from scenario SDN2 focuses on the situation when a false MAC learning table entry caused a host to receive ICMP relies without issuing ICMP requests.

Figure 16 depicts how this had happened: The false MAC-port binding (V1) was learned from a previous packetIn (V2), which in turn was triggered because the host send a packet with spoofed MAC address (V3), and there had been no matching flow entry for this packet (V4a-d).

Q5. Q5 from scenario SND3 asks why Internet requests made it to a internal database.

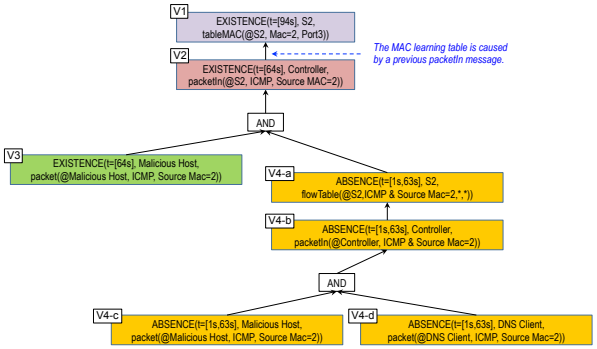


Figure 16: Answer to Q4, as returned by Y!

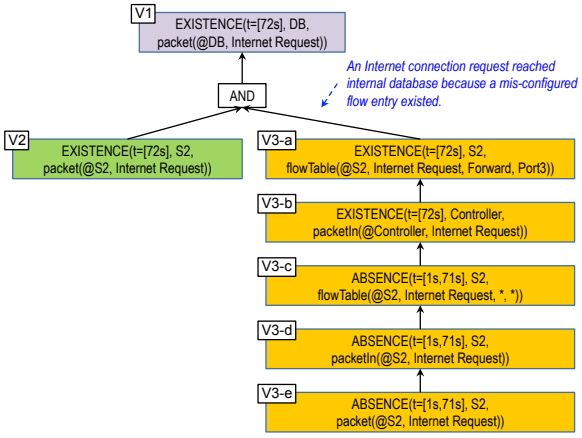


Figure 17: Answer to Q5, as returned by Y!

The problem is explained in Figure 17: An Internet request were seen at the internal database (V1) because such a request did arrive (V2), and there was a misconfigured flow entry (V3a-e) that allowed this packet to pass.

Q7. Q7 from scenario BGP2 asks why a host cannot reach the black-holed host.

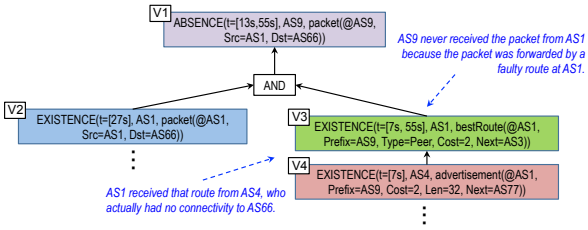


Figure 18: Answer to Q7, as returned by Y!

The problem is explained in Figure 18: When the packet was sent (V2), it was handled by an existing route at AS 1 (V3). That route was advertised by AS 4 (V4), who had no connectivity to the destination of the packet.

Q8. Q8 from scenario BGP3 asks why the ISP cannot reach a certain AS.

As shown in Figure 19, the route timeout at AS 4 (V1-b) and since then the provider AS 1 never sent any advertisements (V1-c). AS 1 did not sent advertisements because it lost its own route (V2-b) and its peer AS 2 stopped advertising routes (V2-c). AS 2 stopped advertising because AS 2 itself lost its route (V3-b). And the route was never recovered because its customer AS 5 never sent the route again (V3-c). AS 5 never sent advertisements again because the route at AS 5 timeout (V4-b) and AS 5 never receive any

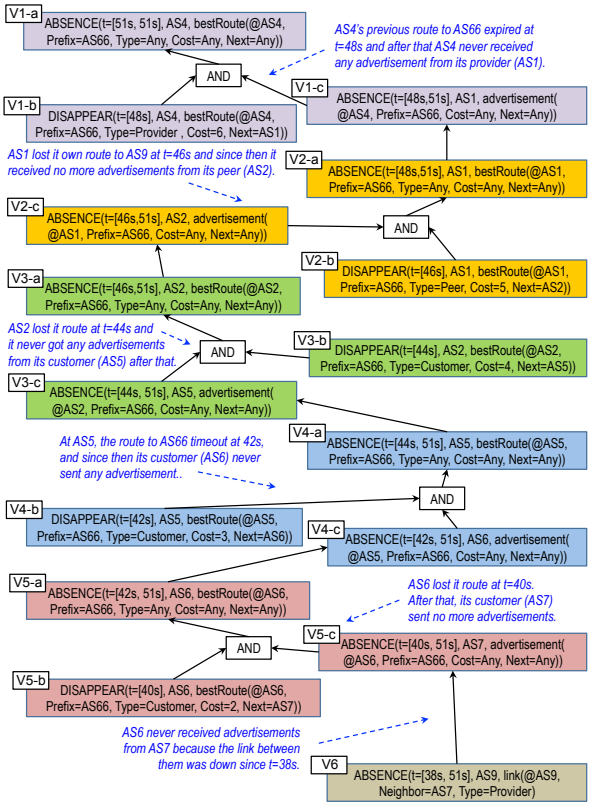


Figure 19: Answer to Q8, as returned by Y!

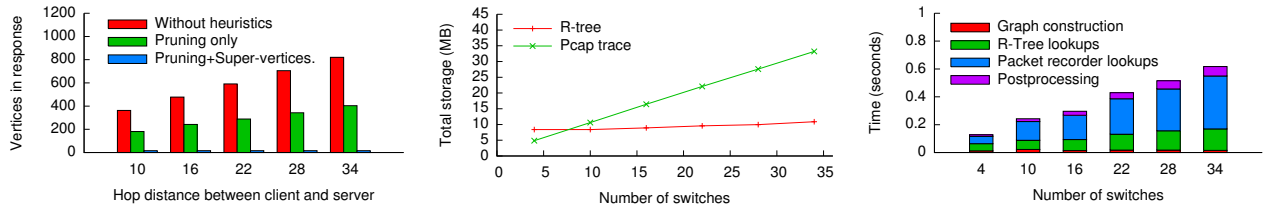
advertisements since then (V4-c). AS 6 could have sent advertisements to AS 5. It did not because its link to AS 7, where the route could have come from, was down since t=38s (V7).

Q9. Q9 from scenario BGP4 asks why the network cannot connect to a particular site.

Figure 21 shows the provance for Q9. The explanation goes as follows: the route timeout at AS 4 (V1-b) and since then the provider AS 1 never sent any advertisements (V1-c). AS 1 did not sent advertisements because it lost its own route (V2-b) and its peer AS 2 stopped advertising routes (V2-c). AS 2 stopped advertising because AS 2 itself lost its route (V3-b). And the route was never recovered because its customer AS 5 never sent the route again (V3-c). AS 5 never sent advertisements again because the route at AS 5 timeout (V4-b) and AS 5 never receive any advertisements since then (V4-c). AS 6 could have sent advertisements to AS 5, but it did not. Because although it kept receiving advertisements from AS 7 (V6-c), there was an updated entry in the BOGON list which caused AS 6 to ignore the advertisements (V6-d).

C. SCALABILITY

To get a sense of how well Y! scales with network size, we ran the learning switch query (Q1) on networks of different sizes, and with different amounts of traffic. We measured the complexity of explanation, storage overhead and query speed. We discussed the result on fat-tree topologies in Section 7.7. In fat-tree topologies, the network diameter is bounded by tree height. For example, the longest hop distance in Figure 13(a) is 6. To get a sense of how Y! scales in networks with even larger diameters, we focused on a linear topology. In each experiment, we changed the number of hops between the HTTP server and client. The other settings remain the



(a) Size of query results (b) Use of storage space (c) Query turnaround time

Figure 20: Additional scalability results

same. For example, the number of packets traveling through the network is kept stable.

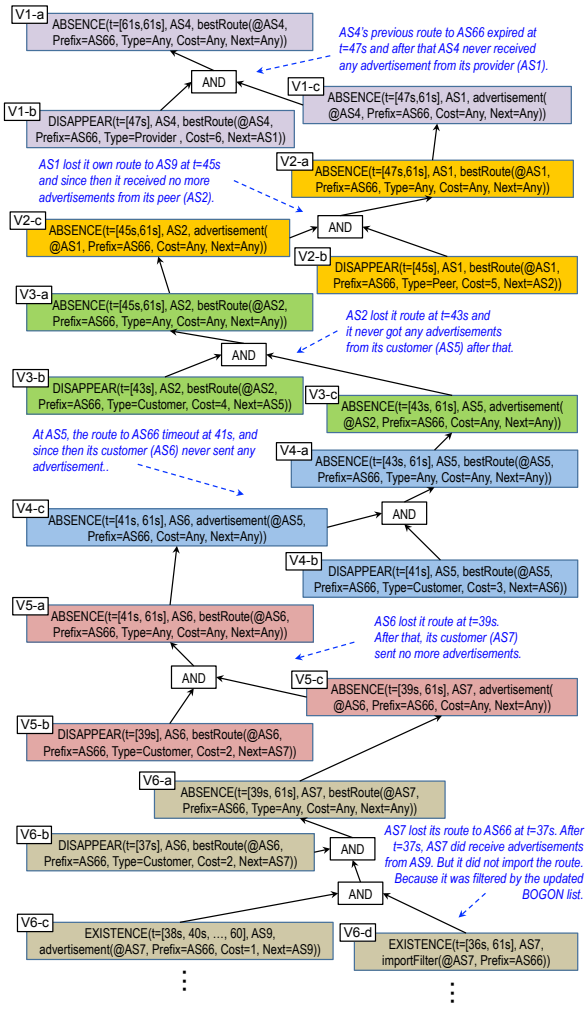


Figure 21: Answer to Q9, as returned by Y!

Complexity: We tested whether the complexity of the provenance (number of vertices) increases with the hop count between the server and the client. Our results are shown in Figure 20(a). As expected, the size of the raw provenance for Q1 grew substantially from 363 to 820 vertices. However once all the heuristics had been applied, the size of the provenance was 16 vertices, independent of the hop count. The reason is similar to what we described in Section 7.7.

Storage: Figure 20(b) shows how Y!'s storage requirements grew with the number of switches in the network. As expected, the size of both the `pcap` trace and the R-tree was roughly proportional to the size of the network. The `pcap` trace grows because when network grows, each packet will traverse more switches, and each switch will append the event to its trace. The R-tree grows because handling packets at each additional switch will generate state changes, which are recorded in R-trees.

Query speed: We issued a query Q2 at the end and measured its turn-around time. Figure 20(c) shows our results: the query speed scales linearly with the number of switches. The dominant cost was the time it took to find packets in the `pcap` trace; the R-tree lookups were much faster, and the time needed to construct and post-process the graph was so small that it is difficult to see in the figure. This is expected because the `pcap` trace query is not optimized. As we mentioned in Section 7.7, an additional time index should reduce the time significantly.

The experiments show that our previous conclusion in Section 7.8 still holds. The complexity of explanation is reduced considerably by Y!'s post-processing heuristics, which reduced the number of vertices by more than an order of magnitude. Y!'s main run-time cost is the storage it needs to maintain a history of the Y!'s past states.