Differential Provenance: Better Network Diagnostics with Reference Events

Ang Chen
University of Pennsylvania

Yang Wu
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Wenchao Zhou
Georgetown University

Boon Thau Loo
University of Pennsylvania

ABSTRACT

In this paper, we propose a new approach to diagnosing problems in complex networks. Our approach is based on the insight that many of the trickiest problems are anomalies – they affect only a small fraction of the traffic (e.g., perhaps a certain subnet), or they only manifest infrequently. Thus, it is quite common for the network operator to have "examples" of both working and non-working traffic readily available – perhaps a packet that was misrouted, and a similar packet that was routed correctly. In this case, the cause of the problem is likely to be wherever the two packets were treated differently by the network.

We sketch the design of a network debugger that can leverage this information using a novel concept that we call *differential provenance*. Like classical provenance, differential provenance tracks the causal connections between network and configuration states and the packets that were affected by them; however, it can additionally reason about the causes of any discrepancies between different provenances. We have performed a case study in the context of software-defined networks, and our initial results are encouraging: they suggest that differential provenance can often identify the root cause of even very subtle network issues.

Categories and Subject Descriptors

D.2.5 [**Testing and debugging**]: Diagnostics

Keywords

Diagnostics, Debugging, Provenance

1. INTRODUCTION

Networks are not easy to get right. Despite the fact that researchers have developed a wide range of tools for network diagnostics [8, 16, 17, 11, 14, 15, 5], understanding the intricate relations between network events for a root-cause analysis is still challenging. This is perhaps especially true for software-defined networks (SDNs), which offers exceptional flexibility by introducing a programmable controller but also further complicates reasoning about network faults.

Network provenance [21] is a promising candidate for understanding the details of network executions, as it provides step-by-step explanations of network events. It encodes network events as vertexes in a distributed database and their causal relations as links; they form a *provenance graph* on which an operator could issue queries that ask for an explanation of a certain event. For example, an operator can ask why a certain route exists in her router's BGP table by issuing a query on the vertex that represents this route in the provenance graph; the answer to her query would be a provenance tree rooted at that vertex, chronicling how the route has been derived from other BGP events. A number of provenance-based diagnostic tools have been developed recently, including systems like ExSPAN [21], SNP [19], and Y! [16]. Conceptually, these systems all provide variants of the same capability: a way to ask for a comprehensive explanation of a certain network event.

However, while a comprehensive explanation is certainly useful for diagnosing a problem, it is not the same as finding the actual *root causes*. We illustrate the difference with an analogy from everyday life: suppose Bob wants to know why his bus arrives at 5:05pm, which is five minutes late. If Bob had a provenance-based debugger, he could submit the query "Why did my bus arrive at 5:05pm?", and he would get a comprehensive explanation, such as "The bus was dispatched at the terminal at 4:00pm, and arrived at stop A at 4:13pm; it departed from there at 4:15pm, and arrived at stop B at 4:21pm; ... Finally, it departed from stop Z at 5:01pm, and arrived at Bob's platform at 5:05pm". This is very different from what Bob really wanted to know; the actual root cause might be something like "At stop G, the bus had to wait for five minutes because of a traffic jam".

But suppose we allow Bob to instead ask about the *differences* between two events – perhaps "why did my bus arrive at 5:05pm today and not at 5:00pm, like yesterday?". Thus, the debugger can omit those parts of the explanation that the two events have in common, and instead focus on the (hopefully few) parts that caused the different outcomes. We argue

HotNets-XIV, November 16–17, 2015, Philadelphia, PA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4047-2/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2834050.2834111.

that a similar approach should work for network diagnostics as well: reasoning about the *differences* between the provenance of a bad event and a good one should lead to far more concise explanations than the provenance of the bad event by itself. We call this approach *differential provenance*.

Differential provenance requires some kind of "reference event" that produced the correct behavior but is otherwise similar to the event that is being investigated. There are several situations where such reference events are commonly available, such as 1) partial failures, where the problem appears in some instances of a service but not in others (Example: DNS servers A and B are returning stale records, but not C); 2) intermittent failures, where a service is available only some of the time (Example: a BGP route flaps due to a "disagree gadget" [7]); and 3) sudden failures, where a network component suddenly stops working (Example: a link goes down immediately after a network transition). As long as the faulty service has worked correctly at some point, that point can potentially serve as the needed reference.

At first glance, it may seem that that differential provenance merely requires finding the differences between two provenance trees, perhaps with a tree-based edit distance algorithm, such as [2]. However, this naïve approach would not work well because small changes in the network can cause the provenance to look wildly different. To see why, suppose that the operator of an SDN expects two packets P and P' to be forwarded along the same path A-B-C-D-E, but that a broken flow entry on B causes P' to be forwarded along A-B-X-Y-Z instead. Although the root cause (the broken flow entry) is very simple, the provenance of P and P' would look very different because the two packets traveled on different paths. (We elaborate on this scenario in Section 2.) A good network debugger should be able to pinpoint just the broken flow entry and leave out the irrelevant consequences.

In this paper, we present an initial algorithm for generating differential provenance (Section 3), we report results from a preliminary case study in the context of SDNs that demonstrates the potential benefits (Section 4), and we discuss some of the challenges ahead (Section 5). We believe that differential provenance can become the basis for new, powerful root-cause analysis tools – not only for SDNs, but also for other types of networks and protocols.

2. OVERVIEW

We begin by explaining our high-level goals using the very simple scenario in Figure 1, which consists of an SDN with six switches, two web servers, and one DPI device. The operator wants web server #2 to handle most of the web requests; however, requests from certain untrusted subnets should be processed by web server #1, because it is colocated with the DPI device that is detecting malicious flows based on the mirrored traffic from S6. To achieve this, she configures two OpenFlow rules on switch S2: a specific rule R_1 that matches traffic from the untrusted subnets and forwards it to S6; and b) a general rule R_2 that matches the rest of the traffic and forwards it to S3. However, the operator made R_1 overly specific by mistake (e.g., by writing the untrusted subnet `4.3.2.0/23` as `4.3.2.0/24`). As a result, only a subset of requests from this subnet arrives at server #1 (e.g., those from `4.3.2.1`), whereas others arrive at server #2 instead (e.g., those from `4.3.3.1`). The operator would like to use a network debugger to investigate why requests from `4.3.3.1` went to the wrong server. One example of a suitable reference event would be a request that arrived at the correct server – e.g., one from `4.3.2.1`.

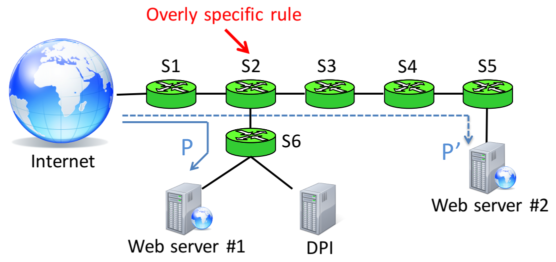
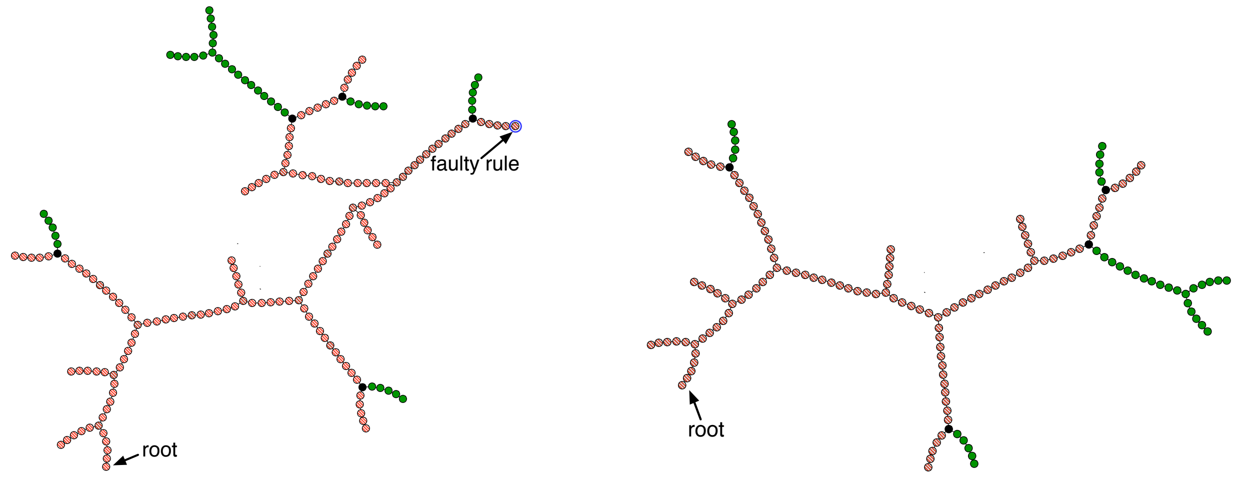


Figure 1: An SDN debugging example.

2.1 Background: Provenance

At a high level, network provenance is a way to describe the causal relationships between events in the network. For instance, if a packet P arrives at web server #1 from Figure 1, the direct cause is P's transmission from a port on switch S6. This, in turn, was caused by 1) P's earlier arrival at S6 via some other port, in combination with 2) the fact that P matched some particular flow entry in S6's flow table. And so on. The events and their causal relationships can be represented as a *provenance graph*, which is a DAG that has a vertex for each event and an edge between each cause and its direct effects. To find the provenance of a specific event, we can simply locate the corresponding vertex in the graph and then walk backwards along the causal edges until we arrive at a set of "base events" that cannot be further explained, such as external inputs or configuration state. Thus, the provenance of an event is simply the tree that is rooted at the corresponding vertex in the provenance graph.

For simplicity, we assume that the network already has a way to track provenance – e.g., a system from existing work [21, 16, 20] – and that the controller program is written in a declarative language, specifically Network Datalog (NDlog) [10]. In NDlog, the network states are viewed as *tables* that can each contain a set of *tuples*. For instance, a switch could have a table called `FlowEntry`, and each of its tuples could encode a flow entry; it could also contain tables `PacketIn`, `PacketMatch`, and `PacketOut` that contain tuples for packets that have been received, matched against a particular flow entry, and sent out to the next hop. Tuples can either be inserted from outside the system – e.g., incoming packets or configuration state – or they can be derived programmatically via *rules* of the form `A:-B,C,...`, which essentially say that a tuple `A` should exist whenever tu-



(a) Provenance of packet P' at web server #2. (b) Provenance of packet P at web server #1.

Figure 2: Provenance of the problematic event (left) and a correct reference event (right) in the scenario from Figure 1. The green vertexes are common to both; and the black ones are "bordering" vertexes. Details of the vertexes have been omitted.

ples `B,C,...` are present. For instance, a rule in an SDN switch could derive `PacketOut` tuples from `PacketIn` and `PacketMatch` tuples.

We use NDlog here primarily because causality is very easy to see in a declarative program: if tuple `A` was derived via rule `A:-B,C`, then its provenance simply consists of tuples `B` and `C`. (For a more complete explanation of NDlog provenance, please see [21, 16].) However, provenance is not specific to a particular language and has been used with existing languages for SDNs, such as Pyretic [16].

2.2 Why is provenance not enough?

Provenance can be helpful for diagnosing a problem, but finding the actual root cause can require substantial additional work. To illustrate this, we used the Y! system [16] to query the provenance of the packet P' in our scenario after it has been (incorrectly) routed to web server #2. The full provenance tree, shown in Figure 2a, consists of no less than 201 vertexes, which is why we have omitted all the details from the figure. Since this is a complete explanation of the arrival of P', the operator can be confident that the information in the tree is "sufficient" for diagnostics, but the actual root cause (the faulty rule; indicated with an arrow) is buried deep within the tree and is quite far from the root, which corresponds to the packet P' itself. This is by no means unusual: in other scenarios that have been discussed in the literature, the provenance often contains tens or even hundreds of vertexes [16]. Because of this, we believe that it would be useful to offer the operator additional help with identifying the actual root causes.

2.3 Key idea: Reference events

Our key idea is to use a *reference event* to improve the diagnosis. A good reference event is one that a) is as similar as possible to the faulty event that is to be diagnosed, but b) unlike that event, has produced the "correct" outcome. Since the reference event reflects the operator's expectations of what the buggy network ought to have done, we must rely on the operator to supply it together with the faulty event.

The purpose of the reference event is to show the debugger which parts of the provenance are actually relevant to the problem. If the provenance of the faulty event and the reference event have vertexes in common, these vertexes cannot be related to the root cause and can therefore be pruned without losing information. If the reference event is sufficiently similar to the faulty event, it is likely that almost all of the vertexes in their provenances will be shared, and that only a very few (perhaps 3 out of 100) will be different. Thus, the operator can focus only on those vertexes, which must include the actual root cause.

For illustration, we show the provenance of the reference packet P from our scenario in Figure 2b. There are quite a few shared vertexes (shown in green), but perhaps not as many as the reader would have expected. This is because of an additional complication that we discuss in Section 2.5.

2.4 Are references typically available?

To understand whether reference events are typically available in practical diagnostic scenarios, we reviewed the posts on the Outages mailing list from 09/2014–12/2014. There are 89 posts in total, and 64 of them are related to network diagnostics (the rest are either irrelevant, e.g., a complaint about iOS 8.0.1, or lacking necessary information to formulate a diagnosis, e.g., a piece of news reporting that a cable was vandalized). We found that 42 of the 64 diagnostic scenarios we examined (66%) contain both a fault and at least one reference event.

We further classified the 42 scenarios into three categories: partial failures, sudden failures, and intermittent failures. The most prevalent problems were *partial failures*, where operators observed functional and failed installations of a service at the same time. For instance, one thread reported that a batch of DNS servers contained expired entries, while records on other servers were up to date. Another class of problems were *sudden failures*, where operators reported the failure of a service that had been working correctly earlier. For instance, an operator asked why a service's status suddenly changed from "Service OK" to "Internal Server Error". The rest were *intermittent failures*, where a service was experiencing instability but was not rendered completely useless. For instance, one post said that diagnostic queries sometimes succeeded, sometimes failed silently, and sometimes took an extremely long time.

2.5 Why not just diff the trees?

Intuitively, it may seem that the differences between two provenance trees could be found with a conventional tree comparison algorithm – e.g., some variant of tree edit distance algorithms [2] – or perhaps simply by comparing the trees vertex by vertex and picking out the different ones. However, there are at least two reasons why this would not work well. The first is that the trees will inevitably differ in some details, such as timestamps, packet headers, packet payloads, etc. These details are rarely relevant for root cause analysis, but a tree comparison algorithm would nevertheless try to align the trees perfectly, and thus report differences almost everywhere. Thus, an equivalence relation is needed to mask small differences that are not likely to be relevant.

Second, and more importantly, small differences in the leaves (such as forwarding a packet to port #1 instead of port #2) can create a "butterfly effect" that results in wildly different provenances higher up in the tree. For instance, the packet may now traverse different switches and match different flow entries that in turn depend on different configuration state, etc. This is the reason why the two provenances in Figures 2a and 2b have so few vertexes in common: the former has 156 vertexes and the latter 201, but the naïve "diff" has as many as 278 – even though the root cause is only a single vertex! Thus, a naïve diff can actually result in a *larger* provenance, which completely nullifies the potential advantage from the reference events.

2.6 Approach: Differential provenance

Differential provenance takes a fundamentally different approach to identifying the relevant differences between two provenance trees. We exploit the fact that a) each provenance describes a particular sequence of events in the network, and that b) given an initial state of the network, the sequence of events that unfolds is largely deterministic. For instance, if we inject two packets with identical headers into the network at the same point, and if the state of the switches is the same in each case, then the packets will (typically) travel along the same path and cause the same sequence of events at the controller. This allows us to predict what the rest of the provenance *would have been* if some vertex in the provenance tree had been different in some particular way.

This enables the following three-step approach for comparing provenance trees: First, we use standard tree comparisons to find a pair of vertexes close to the bottom where the trees are different. Then we conceptually "roll back" the state of the network to the corresponding point, make a change that transforms the one vertex into the other, and then "roll forward" the network again while keeping track of the new provenance along the way. Thus, the provenance tree for the diagnostic event will become more and more like the provenance tree for the reference event. Eventually, the two trees are identical. At this point we output the set of changes (or perhaps only one change!) that transformed the one tree into the other; this is our estimate of the "root cause".

3. DIFFERENTIAL PROVENANCE

In this section, we define the problem from Section 3.1 a bit more formally, and we sketch an initial solution that follows the three steps discussed in Section 2.6. We use the terms "leaves", "base events", and "base tuples" interchangeably in this discussion.

3.1 Problem Statement

Differential provenance takes two inputs from the operator: a) a faulty event $\bar{e}$ to be diagnosed, and b) a reference event e that is considered to be correct with respect to $\bar{e}$. It analyzes their provenance trees T_e and $T_{\bar{e}}$, attempts to change a small set of base tuples (and by derivation, their derived tuples) in the faulty execution, and then replays the changes to see if the fault disappears. Its output is a set of base tuple changes that would correct the fault. For instance, differential provenance might find that making the following changes $\{\Delta\tau_1 = \tau_1 \rightarrow \tau_1', \Delta\tau_2 = \tau_2 \rightarrow \tau_2', \cdots, \Delta\tau_m = \tau_m \rightarrow \tau_m'\}$ would make the fault go away, where $\{\tau_1, \tau_2, \cdots, \tau_n\}$ and $\{\tau_1', \tau_2', \cdots, \tau_m'\}$ are leaves on T_e and $T_{\bar{e}}$, respectively.

3.2 Where do we start?

A trivial way to transform one provenance into another would be to remove all the base tuples from the first and to then insert all the base tuples from the second. However, in practice we prefer transformations that a) change only base tuples that actually can be changed in reality (e.g., configuration state, but not incoming packets), and that b) are as small as possible – we expect the root cause to correspond to the "minimum" set of changes. To this end, we use the following three heuristics to choose the order of the tuples.

Bordering vertexes first: We begin by finding all "bordering vertexes" from the faulty provenance tree $T_{\bar{e}}$. A bordering vertex v is a vertex such that some, but not all, of its children are common subtrees in T_e and $T_{\bar{e}}$ (e.g., the black vertexes in Figure 2). Intuitively, this heuristic is inspired by the observation that similarities between two trees represent similarities in the network executions. Therefore, bordering vertexes are important as they represent the places where two network executions *just begin to diverge*. Thus, changes at those points have a much bigger likelihood of aligning large portions of the provenance.

We identify the bordering vertexes using a tree comparison algorithm based on Largest Common Subtrees (LCSes) as a building block. Intuitively, an LCS captures the maximal amount of similarity between parts of T_e and $T_{\bar{e}}$, and, therefore, between the good and bad executions. It essentially encodes the shared execution that has played a part in causing the appearance of both e and $\bar{e}$. To find LCS, our algorithm first removes irrelevant differences between the two provenance trees (currently by replacing specific fields, such as timestamps and payloads, with zeroes) and then constructs a Merkle Hash Tree (MHT) [12] over each provenance tree. This allows us to find the roots of all common subtrees in one pass, by matching the values of the hashes in the MHT. We then identify the LCSes by performing a BFS from the roots, and reporting the first vertex with a matched MHT top hash. If an unmatched vertex has at least one child that is a LCS, then that vertex is a bordering vertex and we call its subtrees Smallest Distinct Subtrees (SDSes).

Least divergence first: Second, we rank the bordering vertexes by the number of base tuples (i.e., leaves of the corresponding subtrees) in which they differ, and we start with the ones that have the fewest differences. The subtrees rooted at bordering vertexes must have some distinct, unmatched leaves, because otherwise the subtrees would have been the same as well. We use this heuristic because, at such vertexes, only a small number of events have caused the executions to diverge; so they are relatively "simpler" to correct.

After we have ranked all bordering vertexes, we extract the distinct base tuples in their SDSes (according to the order of the bordering vertexes), i.e., tree leaves that are present in one SDS but not the other. We then apply the last heuristic to rank those base tuples. Those ranked tuples will then be processed in the step for rule synthesis.

Recent events first: Finally, we prioritize the base tuples with closer timestamps before the faulty event. This is because for many "sudden failures", their root causes would probably not go too far back.

Although, in our experience, these heuristics tend to work well, they do not provide a "hard" guarantee that a working transformation will be found. Our current algorithm reports "no root causes found" if a) there are no bordering vertexes to start with because the trees have zero overlap, or b) none of the attempted changes to the base tuples work. We are currently working on a solution for these problems, and we provide some additional detail in Section 5.

3.3 How should we make the change?

Given a pair of vertexes identified by the heuristics above, we next decide how to make changes for the subtrees rooted at these two vertexes to be aligned. Since the reference tree gives us an idea of what a correct execution looks like, we have an opportunity to synthesize tentative changes by looking at the reference tree. The result would be a set of additions, deletions, and changes to base tuples in the bad tree.

We find these changes using *rule synthesis* followed by *rule refinement*, and we explain these using the running example in Figure 1. The rule synthesis step starts with a pair of corresponding but different tuples on the faulty tree and the reference tree; in this case, the former will be `flowEntry(@S2, *, 2)`, which says that switch S2 has a default rule that matches all packets and forwards them to switch S3 via port 2, and the latter will be `flowEntry(@S2, 4.3.2.0/24, 1)`, which matches the suspicious subnet and forwards the traffic to switch S6 via port 1. To align the two trees, the algorithm transforms the former tuple into the latter. In the case of base tuples, this is trivial, but in the case of derived tuples the algorithm must find a change to the base tuples (which are the only tuples that can be changed directly) that causes the correct derivations. We use static analysis on the NDlog rules to compute suitable changes.

However, at this point a subtle problem appears: the "bad" packet was for `4.3.3.1`, whereas the "good" packet was for `4.3.2.1`! Thus, a change to the "bad" tree that causes `flowEntry(@S2, 4.3.2.0/24, 1)` to appear verbatim will *not* cause the correct sequence of events, since it does not match the "bad" packet. Our current solution is to use wildcards during rule synthesis (which would initially result in `flowEntry(@S2, *, 1)` and make *all* packets go to web server #1) and to then refine the header space in a subsequent step. Our goal is to find the largest header space that will correct the problem while avoiding undesirable side effects that might change other parts of the tree. To do this, our refinement technique starts with wildcards at the base tuples and then moves up the provenance tree while narrowing the IP ranges at each step as needed until it either a) reaches the root of the subtree or b) the header space cannot be refined further. In our example, this would result in `flowEntry(@S2, 4.3.3.1/24, 1)`.

Notice that the goal of refinement is not necessarily to find the "correct" tuple to insert or the "correct" IP range against which to match. While this happens to work out in our simple example, it would not be realistic to expect this in general: since the algorithm has only one reference packet, it would be difficult to infer the correct subnet (which could be `4.3.3.1/27`, for instance). Rather, our goal is to identify *where* in the network changes need to be made; it is up to the operator to decide what the correct behavior should be.

4. CASE STUDY

We have set up the scenario from Figure 1 in RapidNet [1], using a similar method of modeling OpenFlow switches as in Y! [16], and we have implemented an initial version of our differential provenance algorithm. We then used this setup to answer the following three provenance queries:

- Q1: a conventional provenance query asking "why did HTTP packet P1 appear at the web server 1?";
- Q2: a conventional provenance query asking "why did HTTP packet P2 appear at the web server 2?"; and
- Q3: a differential provenance query asking "why did HTTP packet P1 appear at the web server 1 *but* packet P2 appear at web server 2?"

Query	Tree size	Base tuples
Q1	156	12
Q2	201	15
Plain diff	278	14
Q3	1	1

Table 1: Sizes of the provenance trees in our case study.

Table 1 reports the overall number of vertexes and the number of base tuples in the resulting provenance trees, as well as in a naïve diff between the trees for Q1 and Q2. We observe that the plain diff contains 278 vertexes, which is far bigger than either Q1 or Q2 separately (and thus presumably even less useful!). In contrast, the differential query Q3 was able to correctly identify the *one* vertex (the faulty flow entry) that caused the discrepancy.

We have also tested our prototype with two other scenarios: one that involves a fault caused by a bad interaction between multiple applications [5], and another that involves a fault caused by an unexpected flow entry expiration. We omit the details due to lack of space, but we briefly report our key findings: the naïve diffs contained 238 and 74 vertexes, respectively, whereas differential provenance was able to identify a single vertex as the root cause in both cases.

5. CHALLENGES AND NEXT STEPS

In this section, we outline some interesting challenges that we plan to address next.

Bad reference events: Differential provenance relies on the assumption that the reference event provided by the operator is sufficiently "comparable" to the faulty event to be diagnosed. However, operators can make mistakes and might provide a reference event that is only superficially similar to the faulty event. In this case, differential provenance would fail to achieve alignment between two executions. We hope to find a way to detect such failures early, and to report them to the operator in a way that is easy to understand – perhaps with a brief description of the dissimilarity that could help the operator find a better reference.

Performance: The scenarios we have tried so far are relatively small, but, in an enterprise setting, both the provenance and the network to be "simulated" during tree alignment would be substantially larger. Fortunately, the tree comparison step is easily parallelizable (e.g., by subtrees), and for the replay step we should be able to leverage optimizations from existing work in other areas. Two interesting examples of such work are: a) work on deterministic replay that can identify shared executions across multiple replays, and speed things up by remembering shared states [9]; and b) work from the database literature on incremental view maintenance [21], which can efficiently find the consequences of small changes without repeating the entire computation. We are currently exploring those directions.

Multiple references: The base differential provenance offers an operator the ability to specify one reference event in the system. It would be interesting to consider giving the debugger multiple references when they are available; this might be a way to infer some extra information, such as, in Section 3.3, the correct IP range to be matched.

6. RELATED WORK

Provenance: Provenance is a concept borrowed from the database community [3], and it captures causal relations between database tuples. Researchers have applied provenance to distributed systems [21, 19, 16], storage systems [13], operating systems [6], and mobile platforms [4]. Our work is mainly related to projects that use network provenance for diagnostics, such as ExSPAN [21] that maintains network provenance at scale, SNP [19] that secures network provenance information in adversarial settings, Y! [16] that extends the provenance model to answer negative queries, etc. All those projects use the provenance of *individual* events, while we add *reference* events for root cause analysis.

Diagnostics: Our work is related to many existing network diagnostics projects, including Anteater [11], Header Space Analysis [8], OFRewind [17], Minimal Causal Sequence analysis [14], and ATPG [18]. The most closely related project is PeerPressure [15], which diagnoses misconfigured machines by comparing their registry values to those on a set of reference machines. However, PeerPressure uses Bayesian inference rather than provenance, so it can only compute a probability that a given configuration setting is responsible for the observed symptoms, whereas our approach can establish a causal connection.

7. CONCLUSION

In this paper, we have introduced a novel approach to network diagnostics that is based on the observation that network faults tend to be the exception rather than the norm – they often affect only a small fraction of the traffic, and they often manifest infrequently. Thus, operators often have "examples" of both good traffic and bad traffic readily available. Our approach uses network provenance to find events that are causally connected to the good and/or the bad example, and then reasons about the differences between the two provenances to isolate a set of root causes that is responsible for the different outcomes. Our initial results seem encouraging – the final result can be as precise as a *single* root cause – but some challenges remain to be addressed before the approach can be deployed in an operational setting. We are addressing these challenges in our ongoing work.

Acknowledgments: We thank the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CNS-1054229, CNS-1065130, CNS-1117052, CNS-1218066, CNS-1453392, and CNS-1513734; DARPA contract HR0011-15-C-0098; and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy (NSF CNS-1505799).

8. REFERENCES

[1] RapidNet. `http://netdb.cis.upenn.edu/rapidnet/`.

[2] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, June 2005.

[3] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, Jan. 2001.

[4] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security*, 2011.

[5] R. Durairajan, J. Sommers, and P. Barford. Controller-agnostic SDN debugging. In *Proc. CoNEXT*, 2014.

[6] A. Gehani and D. Tariq. SPADE: Support for provenance auditing in distributed environments. In *Proc. Middleware*, 2012.

[7] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, Apr. 2002.

[8] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012.

[9] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *Proc. OSDI*, 2012.

[10] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, Nov. 2009.

[11] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. SIGCOMM*, 2012.

[12] R. Merkle. Protocols for public key cryptosystems. In *Proc. IEEE Symposium on Security and Privacy*, 1980.

[13] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In *Proc. USENIX Annual Technical Conference*, 2009.

[14] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Proc. SIGCOMM*, 2014.

[15] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proc. OSDI*, 2004.

[16] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proc. SIGCOMM*, 2014.

[17] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proc. USENIX Annual Technical Conference*, 2011.

[18] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.

[19] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proc. SOSP*, Oct. 2011.

[20] 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.

[21] 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.