The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance

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 distributed systems. Our approach is based on the insight that many of the trickiest problems are anomalies. For instance, in a network, problems often affect only a small fraction of the traffic (perhaps a certain subnet), or they only manifest infrequently. Thus, it is quite common for the 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 present the design of a debugger that can leverage this information using a novel concept that we call *differential provenance*. Differential provenance tracks the causal connections between network states and state changes, just like classical provenance, but it can additionally perform root-cause analysis by reasoning about the *differences* between two provenance trees. We have built a diagnostic tool that is based on differential provenance, and we have used our tool to debug a number of complex, realistic problems in two scenarios: software-defined networks and MapReduce jobs. Our results show that differential provenance can deliver very concise diagnostic information; in many cases, it can even identify the precise root cause of the problem.

CCS Concepts

•**Networks** → **Network management;** *Network experimentation;* •**Information systems** → **Data provenance;**

Keywords

Network diagnostics, debugging, provenance

SIGCOMM '16, August 22 - 26, 2016, Florianópolis, Brazil

 ISBN 978-1-4503-4193-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2934872.2934910

1. INTRODUCTION

Distributed systems are not easy to get right. Despite the fact that researchers have developed a wide range of diagnostic tools [16, 30, 31, 19, 27, 29, 10], understanding the intricate relations between low-level events, which is needed for root-cause analysis, is still challenging.

Recent work on *data provenance* [36] has provided a new approach to understanding the details of distributed executions. Intuitively, a provenance system keeps track of the causal connections between the states and events that a system generates at runtime; for instance, when applied to a software-defined network (SDN), it might associate each flow entry with the parts of the controller program that were used to compute it. Then, when the operator asks a diagnostic question – say, why a certain packet was routed to a particular host – the system returns a comprehensive explanation that recursively explains each relevant event in terms of its direct causes. A number of provenance-based diagnostic tools have been developed recently, including systems like ExSPAN [36], SNP [34], and Y! [30].

However, while such a comprehensive explanation is 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 arrived 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?". The debugger can then 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 that a similar approach should work for diagnos-

ing distributed systems: 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" [12]); 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 [5]. 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 S1-S2-S3-S4-S5, but that a broken flow entry on S2 causes P' to be forwarded along S1-S2-S6 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 very different paths. (We elaborate on this scenario in Section 2.) A good debugger should be able to pinpoint just the broken flow entry and leave out the irrelevant consequences.

In this paper, we present a concrete algorithm called DiffProv for generating differential provenance, as well as a prototype debugger that leverages such information for root-cause analysis. We report results from two diagnostic scenarios: software-defined networks and Hadoop MapReduce. Our results show that differential provenance can explain network events in far simpler terms than existing systems: while the latter often return elaborate explanations that contain hundreds of events, DiffProv can usually pinpoint one critical event which, in our experience, represents the "root cause" that a human operator would be looking for. We also show that the cost for the higher precision is small: the runtime overheads are low enough to be practical, and diagnostic queries can usually be answered in less than one minute. We make the following contributions:

- The concept of differential provenance (Section 3);
- DiffProv, a concrete algorithm for generating differential provenance (Section 4);
- a DiffProv debugger prototype (Section 5); and
- an experimental evaluation in the context of SDNs and Hadoop MapReduce (Section 6).

We discuss related work in Section 7, and conclude the paper in Section 8.

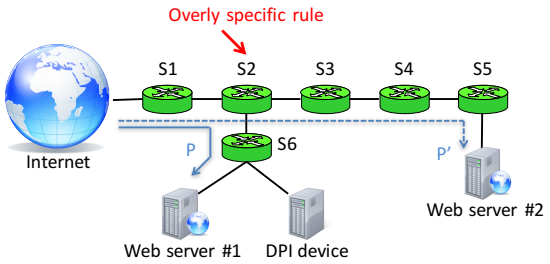


Figure 1: Example scenario (SDN debugging).

2. OVERVIEW

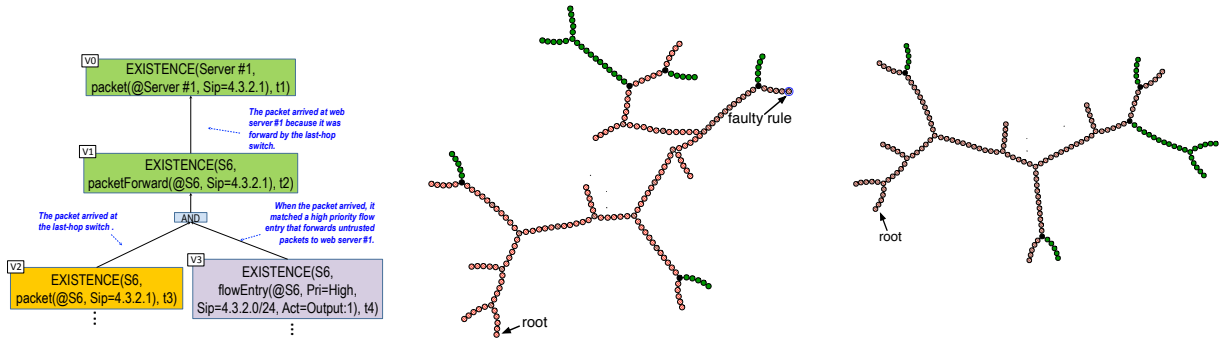
Figure 1 shows a simple example of the problem we are addressing. The illustrated network consists of six switches, two HTTP servers, and one DPI device. The operator wants web server #2 to handle most of the HTTP requests; however, requests from certain untrusted subnets should be processed by web server #1, because it is co-located with the DPI device that can detect malicious flows based on the mirrored traffic from S6. To achieve this, the operator configures two OpenFlow rules on switch S2: a) 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, writing the untrusted subnet `4.3.2.0/23` as `4.3.2.0/24`. As a result, only some of the requests from this subnet arrive 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`.

2.1 Background: Provenance

Network provenance [36] is a way to describe the causal relationships between network events. At a high level, the provenance of an event e is simply a tree of events that has e at its root, and in which the children of each vertex represent the direct causes of that vertex. Figure 2(a) sketches the provenance of the packet P from Figure 1 when it arrives at web server #1. The direct cause of P's arrival is that P was sent from a port on switch S6 (vertex V1); this, in turn, was caused by 1) P's earlier arrival at S6 via some other port (V2), in combination with 2) the fact that P matched some particular flow entry in S6's flow table (V3), and so on.

To answer provenance queries, systems use the abstraction of 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 e, we can simply locate e's vertex in the graph and then project out the tree that is rooted at that vertex. The leaves of the tree consist of "base events" that cannot be further explained, such as external inputs or configuration states.

Provenance itself is not a new concept; it has been explored by the database and networking communities, and there are techniques that can track it efficiently by maintaining some additional metadata [6, 11, 30].



(a) Provenance example (b) Full provenance of P′ at server #2 (c) Full provenance of P at server #1

Figure 2: Simplified excerpt from a provenance tree (a) and the full provenance trees for P' (b) and P (c) from Figure 1. Each circle in (b) and (c) corresponds to a box in (a), but the details have been omitted for clarity. Although the two full trees have some common subtrees (green), most of their vertexes are different (red). Also shown is the single vertex in (b) that represents the root cause of the routing error that affected P'.

2.2 Why provenance is 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 queried 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 2(b), 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 diagnosis. However, 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 were discussed in the literature, the provenance often contains tens or even hundreds of vertexes [30]. Hence, extracting a concise root cause from a complex causal explanation remains challenging.

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 being 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 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 at hand*. 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 very few 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 2(c). There are quite a few shared vertexes (shown in green), but perhaps not as many as one might 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 diagnosis. (The others are either irrelevant, such as complaints about a particular iOS version, or are lacking information that is needed to formulate a diagnosis, such as a news report saying that a cable was vandalized.) We found that 45 of the 64 diagnostic scenarios (70.3%) contain both a fault and at least one reference event; however, in ten of the 45 scenarios, the reference event occurred in another administrative domain, so we cannot be sure that the operator would have had access to the corresponding diagnostic data. Nevertheless, even if we ignore these ten events, this leaves us with 35 out of 64 scenarios (or slightly more than half) in which a reference event would have been available.

We further classified the 45 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.

In most of the scenarios we examined, the reference event could have been found in one of two ways: either a) by taking the malfunctioning system and looking back in time for an instance where that same system was still working correctly, or b) by looking for a *different* system or service that coexists with the malfunctioning system but has not been affected by the problem. Although our survey is far from universal, these strategies are quite general and should be applicable in many other scenarios.

2.5 Why not compare the trees directly?

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 [5] – 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 perhaps 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 states, etc. This is the reason why the two provenances in Figures 2b and 2c still have considerable differences: the former has 201 vertexes and the latter 156, but the naïve "diff" has as many as 278 – even though the root cause is only a single vertex! Thus, a naïve diff may actually be *larger* than the underlying provenances, which completely nullifies the 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 in the network. 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 locate a pair of "seed" vertexes that triggered the diagnostic event and the reference event. We then conceptually "roll back" the state of the network to the corresponding point, make a change that transforms some "bad" vertex into a good one, 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 equivalent. 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 introduce the concept of differential provenance. For ease of exposition, we adopt a declarative system model that is commonly used in database systems when reasoning about provenance. This model describes a system's states as *tuples*, and its algorithm as *derivation rules* that process the tuples. The key advantage of using this model is that provenance is very easy to see in the syntax. Although one can directly program with such rules and then compile them into an executable [18], few deployed systems are written that way today. However, DiffProv is not specific to the declarative model: in Section 5, we describe several ways in which rules and tuples can be extracted from systems that are written in other languages, and our prototype debugger has a front-end that accepts SDN programs that are written in Pyretic [21], an imperative language.

3.1 System model

We assume that the system that is being diagnosed consists of multiple nodes that run a distributed protocol, or a combination of protocols. System states and events are represented as *tuples*, which are organized into *tables*. For instance, the model for an SDN switch would have a table called `FlowEntry`, where each row encodes an OpenFlow rule and each column encodes a specific attribute of it, e.g., incoming port (`in_port`), match fields (`nw_dst`), actions (`actions`), and others. As a simplified example, a tuple `FlowEntry(5,8,1.2.3.4)` may indicate that packets with destination IP `1.2.3.4` that arrive on port 5 should be sent out on port 8.

The algorithm of the system is described by a set of *derivation rules*, which encodes how tuples could be derived when and where. External events to the system, such as incoming packets, are modeled as *base tuples*. Whenever a base tuple arrives, it will trigger a set of derivation rules and cause new *derived tuples* to appear; the derived tuples may in turn trigger more rules and produce other derived tuples. Rules have the form `A :- B,C,...`, which means that a tuple `A` will be derived whenever tuples `B,C,...` are present; for instance, the model for an SDN switch would have a rule that derives `PacketOut` tuples from `PacketIn` and `FlowEntry` tuples. Rules can also specify tuple locations using the `@` symbol to encode a distributed operation: for instance, `A(i,j)@X :- B(i)@X,C(j)@Y` indicates that an `A(i,j)` tuple should be derived on node `X` whenever a) node `X` has a `B(i)` tuple and b) node `Y` has a `C(j)` tuple. Here, `i` and `j` are variables of certain *types*, e.g., IP ranges, switch ports, etc.

The provenance system observes how the primary system runs, keeps track of its derivation chains, and uses them to explain why a particular system event occurred. The provenance of a tuple is very easy to explain in terms of the derivation rules: a base tuple's provenance is itself, since it cannot be explained further; a derived tuple's provenance consists of the rule(s) that have been used to derive it, as well as the tuples used by the rule(s). For instance, if a tuple `A` was derived using some rule `A :- B,C,D`, then `A` exists simply because tuples `B`, `C`, and `D` also exist. Without loss of generality, we model tuple deletions as insertions of special "delete" tuples; this results in an append-only maintenance of the provenance graph.

3.2 The provenance graph

There are different ways to define provenance, and our approach does not depend on the specific details. For concreteness, we will use a simplified version of the temporal provenance graph from [35]. We chose this graph because its temporal dimension enables the graph to "remember" past events; this is useful, e.g., when the reference event is something that happened in the past. The graph from [35] consists of the following seven vertex types:

- INSERT(n, τ, t), DELETE(n, τ, t): Base tuple τ was inserted (deleted) on node n at time t;
- EXIST$(n, \tau, [t_1, t_2])$: Tuple τ existed on node n from time t_1 to t_2;
- DERIVE(n, τ, R, t), UNDERIVE(n, τ, R, t): Tuple τ was derived (underived) via rule R on n at time t;
- APPEAR(n, τ, t), DISAPPEAR(n, τ, t): Tuple τ appeared (disappeared) on node n at time t;

The provenance graph is built incrementally at runtime. When a base tuple is inserted, this causes an INSERT to be added to the graph, followed by an APPEAR (to reflect the fact that a new tuple appeared), and finally an EXIST (to reflect that the tuple now exists in the system). Having three separate vertexes may seem redundant, but will be useful later – for example, when DiffProv must find tuples that "appeared" last. If the appearance of a tuple triggers a derivation via a rule, a DERIVE vertex is added to the graph. The remaining three "negative" vertexes (DELETE, UNDERIVE, and DISAPPEAR) are analogous to their positive counterparts.

3.3 Towards a definition

We are now ready to formalize the problem we have motivated in Section 2. For clarity, we start with the following informal definition (which we then refine in several steps):

DEFINITION ATTEMPT 1. *Given a "good" provenance tree T_G with root vertex v_G and a "bad" provenance tree T_B with root vertex v_B, differential provenance is the reason why the two trees are not the same.*

More precisely, we adopt a counterfactual approach to define "the reason": although the actual provenance of v_G is clearly different from that of v_B, we can look for changes to the system that *would have* caused the provenances to be the same. For instance, in the example from Section 2, the actual reason why the packets P and P' were routed differently was an overly specific flow entry; by changing that flow entry into a more general one, we can cause the two packets to take the same path. Since any change can be captured by a combination of changes to base tuples, we can restate our goal as finding some set $\Delta_{B \to G}$ of changes to base tuples that would transform the "bad" tree into the "good" one.

Refinement #1 (Mutability): Importantly, not all changes to base tuples make sense in practice. For instance, in our SDN example, it is perfectly reasonable to change base tuples that represent configuration states, but it is not reasonable to change base tuples that represent incoming packets, since the operator has no control over the kinds of packets that arrive at her border router. Thus, we distinguish between *mutable* and *immutable* base tuples, and we do not consider changes that involve the latter. (Note that this restriction implies that a solution does not always exist.) We thus arrive at our next attempt:

DEFINITION ATTEMPT 2. *Given two provenance trees T_G and T_B, their differential provenance is a set of changes $\Delta_{B \to G}$ to mutable tuples that transforms T_B into T_G.*

Refinement #2 (Preservation of seeds): Even when restricted to mutable tuples, the above definition is not quite right, because we are not looking to transform T_B into T_G *verbatim*: this contradicts our intuition that T_B is about a *different* event, and that a meaningful solution must preserve the events whose provenance the trees represent. To formalize this notion, we designate one leaf tuple in each tree as the *seed* of that tree, to reflect that the tree has "sprung" from that event, and we require that the seeds be preserved while the trees are being aligned. To identify the seed, observe that, whenever a tuple A is derived through some rule `A:-B,C,D,...`, one of the underlying tuples `B`, `C`, `D`, ... was the last one to appear and thus has "triggered" the derivation. Thus, we can follow the chain of triggers from the root to exactly one of the leaves, which, in a sense, triggered the entire tree.

Refinement #3 (Equivalence): If the changes to T_B must preserve its seed, the question arises how the two trees could ever be "the same" if their seeds are different. Therefore, we need a notion of *equivalence*. For instance, suppose that `pkt(1.2.3.4,80,X)` and `pkt(1.2.3.5,80,Y)` are the seeds, representing two HTTP packets for two different interfaces of the same server. Then, when aligning the two trees, we must account for the fact that the IP addresses and payloads are different. In simple cases, this might simply mean that all the occurrences of `1.2.3.4` in T_G are replaced with `1.2.3.5` in T_B, but there are more complicated cases – e.g., when the controller program computes different flow entries for the two IPs, perhaps even with different functions. We will discuss this more in Section 4.3.

With these refinements, we arrive at our final definition:

```
function DIFFPROV(T_G, T_B)
    s_G ← FINDSEED(T_G)
    s_B ← FINDSEED(T_B)
    if s_G ≠̸ s_B then FAIL
    Δ_{B→G} ← ∅
    while T_G ≠̸ T_B do
        (τ_G, τ_B) ← FIRSTDIV(s_G, s_B)
        τ'_G ← APPLYTAINT(τ_G)
        MAKEAPPEAR(τ'_G, τ_G)
        T_B ← UPDATETREE(T_B, Δ_{B→G})
    return Δ_{B→G}

function FIRSTDIV(s_G, s_B)
    for each field s_G[i] ≠ s_B[i]
        CREATETAINT(s_G[i], s_B[i])
    τ_G ← s_G, τ_B ← s_B
    while τ_G ≃ τ_B do
        PROPTAINT(τ_G → PARENT(τ_G))
        PROPTAINT(τ_B → PARENT(τ_B))
        τ_G ← PARENT(τ_G)
        τ_B ← PARENT(τ_B)
    return (τ_G, τ_B)

function MAKEAPPEAR(τ'_G, τ_G)
    if BaseTuple(τ'_G) then
        if ImmutableTuple(τ'_G) then FAIL
        Δ_{B→G} ← Δ_{B→G} ∪ {τ'_G}
    else
        for τ_i ∈ CHILDREN(τ_G) do
            PROPTAINT(τ_G → τ_i)
            τ'_i ← APPLYTAINT(τ_i)
            if ∄τ'_i then MAKEAPPEAR(τ'_i,τ_i)
    return
```

Figure 3: Pseudocode of the DiffProv algorithm. The FINDSEED, FIRSTDIV, MAKEAPPEAR, and UPDATETREE functions are explained in Sections 4.2, 4.4, 4.5, and 4.6 respectively. The CREATETAINT, PROPTAINT, and APPLYTAINT functions are introduced to establish equivalence between corresponding tuples in T_G and T_B (Section 4.3).

DEFINITION 1 (DIFFERENTIAL PROVENANCE). *Given two provenance trees T_G and T_B with seed tuples s_G and s_B, the differential provenance of T_G and T_B is a set of changes $\Delta_{B\rightarrow G}$ to mutable tuples that 1) transforms T_B into a tree that is equivalent to T_G, and 2) preserves s_B.*

Figure 4 illustrates this definition with a simple derivation rule `C(x,y^2,z+1):-A(x,y),B(x,y,z)` and three example tuples. The seeds `A(1,2)` and `A(2,2)` are considered to be equivalent (and immutable). To align the two provenance trees, the differential provenance of T_B and T_G would be a change from the mutable base tuple `B(1,2,3)` in T_B to `B(1,2,4)`, which makes it equivalent to its corresponding tuple `B(2,2,4)` in T_G. This update will be propagated and further change `C(1,4,4)` to `C(1,4,5)` in T_B, which now becomes equivalent to tuple `C(2,4,5)` in T_G.

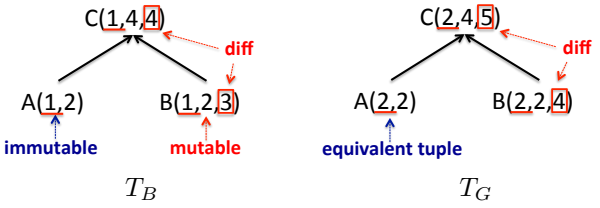


Figure 4: A simplified example showing the differential provenance for a one-step derivation. `A(1,2),A(2,2)` are the seeds; equivalent fields are underlined, and differences are boxed. Differential provenance transforms `B(1,2,3)` into `B(1,2,4)` to align this derivation.

4. THE DIFFPROV ALGORITHM

In this section, we present DiffProv, a concrete algorithm that can generate differential provenance. Initially, we will assume that the two trees are completely materialized and have been downloaded to a single node; however, we will remove this assumption at the end of this section.

4.1 Roadmap

The DiffProv algorithm is shown in Figure 3. We begin with an intuitive explanation, and then explain each step in more detail.

When invoked with two provenance trees – a "good" tree T_G and a "bad" tree T_B – DiffProv begins by identifying the seed tuples of both trees (Section 4.2). DiffProv then verifies that the two seed tuples are of the same type; if they are not, T_G and T_B are not really comparable, and the algorithm fails. Otherwise, DiffProv defines an equivalence relation that maps the seed of the "bad" tree to the seed of the "good" tree (Section 4.3). This helps DiffProv to align a first tiny subtree of the two trees, which provides the base case for the following inductive step.

Starting with a pair of subtrees that are already aligned, DiffProv then identifies the parent vertexes τ_G and τ_B of the two trees and checks whether they are already the same under the equivalence relation defined earlier (Section 4.4). If so, DiffProv has found a larger pair of aligned subtrees, and repeats. If not, DiffProv checks which children of τ_G are not present in T_B, and then attempts to make changes so as to make these children appear (Section 4.5–4.6). In doing so, DiffProv heavily relies on the "good" tree T_G as a guide: rather than trying to guess combinations of base tuple changes that might cause the missing tuples to be created, DiffProv creates them *in the same way* that they were created in T_G (modulo equivalence), which reduces an exponential search problem to a linear one.

During alignment, DiffProv accumulates a set of base tuple changes. Once the roots of T_G and T_B have been reached, DiffProv outputs the accumulated changes as $\Delta_{B\rightarrow G}$ and terminates.

4.2 Finding the seeds

Given the two provenance trees T_G and T_B, DiffProv's first step is to find the seed of each tree. To do this, DiffProv uses the following insight: unlike databases, distributed systems and networks usually do not perform one-shot computations; rather, they respond to external stimuli. For instance, networks route incoming packets, and systems like Hadoop process incoming jobs. Thus, the provenance of an output is not a uniform tree; rather, there will be one "special" branch of the tree that describes how the stimulus made its way through the system (say, the route of an incoming packet), while the other branches describe the reasons for what happened at each step (say, configuration states). The seed of the tree is simply the external event, which can be found at the bottom of this "special" branch.

At first glance, it may seem difficult to find this stimulus in a given provenance tree, but in fact there is an easy way to do this. Notice that each derivation is triggered because its last precondition has been satisfied; for instance, if a tuple `A` was derived through a rule `A:-B,C,D`, then one of the three tuples `B`, `C`, and `D` must have appeared last, when the other two were already present. Thus, this last tuple represents the stimulus for the derivation. Conveniently, the provenance graph we have adopted (see Section 3.2) already has a special vertex – the APPEAR vertex – to identify this tuple.

Thus, DiffProv can find the seed as follows. Starting at the root of each tree, it performs a kind of recursive descent: at each vertex v, it scans the direct children of v, locates the APPEAR vertex with the highest timestamp, and then descends into the corresponding branch of the tree. By repeating this step, DiffProv eventually reaches a leaf that is of type INSERT, which it then considers to be the seed.

4.3 Establishing equivalence

Next, DiffProv checks whether the seeds of T_G and T_B are of the same type. It is possible that they are not; for instance, the operator might have asked DiffProv to compare a flow entry that was generated by the controller program to one that was hard-coded. In this case, the two trees are not really comparable, and DiffProv fails.

Even if the seeds s_G and s_B do have the same type, some of their fields will be different. For instance, s_G might be a packet `pkt(1.2.3.4,80,A)`, and s_B might be a packet `pkt(1.2.3.5,80,B)`; in this case, the two packets have the same port number (80) but different IP addresses and payloads. This is not a problem for the seeds themselves, since they are equivalent by definition (Section 3.3); however, it is a problem for tuples that are – directly or indirectly – derived from the seeds. For instance, if a tuple τ:=`portAndLastOctet(80,4)` was derived from s_G via a chain of several different rules, how can DiffProv know what tuple would be the equivalent of τ in T_B? A human diagnostician could intuitively guess that it should be `portAndLastOctet(80,5)`, since the last octet in s_B was 5, but DiffProv must find some other way.

To this end, DiffProv taints all the fields of tuples in T_G that have been computed from fields of s_G in some way, and maintains, for each tainted field, a *formula* that expresses the field's value as a function of fields in s_G. In the above example, both fields of τ would be tainted. If `X`, `Y`, and `Z` are the three fields of s_G, then the formula for the first field of τ would simply be `Y` (since it is just the port number from the original packet), and the formula for the second field would be `X&0xFF` (since it is the last octet of the IP address in s_G). With these formulae, DiffProv can find the equivalent of any tuple in T_G simply by plugging in the values from s_B. This will become important in the next step, where DiffProv must make missing tuples appear in T_B.

DiffProv computes the taints and formulae incrementally as it works its way up the tree, as we shall see in the next step. Initially, it simply taints each field in s_G and annotates each field with the identity function.

4.4 Aligning larger subtrees

Next, DiffProv attempts to align larger and larger subtrees of T_G and T_B. Each step begins with a pair of subtrees that are already aligned (modulo equivalence); initially, this will be just the two seed tuples.

First, DiffProv propagates the taints to the parent vertex of the good subtree, while updating the attached formulae to reflect any computations. For instance, suppose the root of the subtree was APPEAR`(foo(1,2,3))`, its parent was DERIVE`(bar(1,7),R)`, and that we have a derivation rule that states `bar(a,d):-foo(a,b,c),d=2*c+1`. Then DiffProv would propagate the taint from the `1` in `foo` to the `1` in `bar` and leave its formula unmodified. DiffProv would also propagate the taint from the `3` in `foo` to the `7` in `bar`, but it would attach a different formula to the `7`: if `f` was the formula used to compute the `3` in the good tree from some field(s) of s_G that were different in s_B (see Section 4.3), then DiffProv would attach `g:=2*f+1` to the `7`, to reflect that it was computed using `d=2*c+1`.

Then, DiffProv evaluates the formulae for all the tainted tuples in the parent to compute the tuple that *should* exist in the bad tree. For instance, in the above example, suppose the formulae that are attached to the `1` and the `7` in `bar(1,7)` are `H+1` and `2*(G+1)+1`, where `H=9` and `G=0` are the values of some fields in T_B's seed (see Section 4.3). Then DiffProv would conclude that a `bar(10,3)` tuple ought to exist in T_B, since this would be equivalent to the `bar(1,7)` in T_G based on the equivalence relation.

If the expected tuple exists in T_B and has been derived using the expected rule, DiffProv adds the parent vertexes to both subtrees (as well as any other subtrees of those vertexes) and repeats the induction step with the larger subtrees. If the expected tuple does *not* exist in T_B, DiffProv detects the first "divergence", and will try to make the tuple appear using the procedure we describe next.

4.5 Making missing tuples appear

At first glance, it is not at all clear how to create an arbitrary tuple. The tuple might be indirectly derived from many different base tuples, and attempting random combinations of changes to these tuples would have an exponential complexity. However, DiffProv has a unique advantage in the form of the "good" tree T_G, which shows how an equivalent tuple has already been derived. Thus, DiffProv uses T_G as a guide in its search for useful tuple changes.

DiffProv begins by propagating the taints from the parent of the current subtree in T_G to the other children of that parent. For instance, suppose that the current parent in T_G is a `flowEntry(1.2.3.4,5,8)` that has been derived using `flowEntry(ip,s,d):-pkt(ip,s),cfg(s,d)` on a `pkt(1.2.3.4,5)`, which is the root of the current subtree. Then, DiffProv can simply propagate any taints, and their formulae, from the `5` and the `8` in the `flowEntry` to the corresponding fields in the `config` tuple.

Note that, in general, propagating taints from a vertex v to one of its children can require inverting computations that have been performed to obtain a field of v. For in-

stance, if a tuple `abc(5,8)` has been derived using a rule `abc(p,q):-foo(p),bar(x),q=x+2`, DiffProv must invert `q=x+2` to obtain `x=q-2` and to thus conclude that a `bar(6)` is required. While not all rules are injective or surjective, or are simple enough to be inverted, in practice, the rules we have encountered are usually simple enough to permit this. In cases when automatic inverting is not possible, we depend on the model to provide inverse rules. When there are several preimages (for example, if $q=x^2+4$), DiffProv can try all of them.

DiffProv then uses the formulae to compute, for each child in T_G, the equivalent tuple in T_B, and it checks whether this tuple already exists. The tuple may exist even if it is not currently part of T_B: it may have been derived for other reasons, or it may have been created by earlier changes to base tuples (see Section 4.6). If a tuple does not exist, DiffProv checks whether it is a base tuple. If not, DiffProv looks up the rule that was used to derive the missing tuple in T_G, and then recursively invokes the current step to make the missing children of that tuple appear. If the missing tuple is indeed a base tuple, DiffProv adds that base tuple to $\Delta_{B\rightarrow G}$ and then performs the step we discuss next.

4.6 Updating T_B after tuple changes

Once a new change has been added to $\Delta_{B\rightarrow G}$, DiffProv must update T_B to reflect the change. Since DiffProv is meant to be purely diagnostic, we do not want to actually apply the new update directly into the running system, since this would affect its normal execution. Rather, DiffProv clones the current state of the system when it makes the first change, and applies its changes only to the clone. (Cloning can be performed efficiently using techniques such as copy-on-write.)

The obvious consequence of each update is that one missing tuple in T_B appears. However, the update might cause *other* missing tuples to appear elsewhere that have not yet been encountered by DiffProv, or remove existing tuples that transitively depend on the original base tuple. Therefore, DiffProv allows the derivations in the cloned state to proceed until the state converges. These updates only affect the cloned state, and are not propagated to the runtime system.

If the seeds of the two trees are of the same type, and if DiffProv can successfully invert any computations it encounters while propagating taints, it returns the set of tuple changes $\Delta_{B\rightarrow G}$ as the estimated root cause.

4.7 Properties of DiffProv

Complexity: The number of steps DiffProv takes is linear in the number of vertexes in T_G. This is substantially faster than a naïve approach that attempts random changes to mutable base tuples (or combinations of such tuples), which would have an exponential complexity. DiffProv is faster because of a) its use of provenance, which allows it to ignore tuples that are not causally related to the event of interest, and b) its use of taints and formulae, which enables it to find, at each step, a specific tuple change that will have the desired effect – it never needs to "guess" a suitable change.

False positives: When DiffProv outputs a set of tuple changes, this set will always satisfy our definition from Section 3.3, that is, it will transform T_B into a tree that is equivalent to T_G, while preserving the seed s_B. There are no "false positives" in the sense that DiffProv would recommend changes that have no effect, or recommend changes to tuples that are not related to the problem. However, there is no guarantee that the output will match the operator's intent: if the operator inputs a packet P and a reference packet P', DiffProv will output a change that will make the network treat P and P' the same, even if, say, the operator would have preferred P to take a different path. For this reason, it is best if the operator carefully inspects the proposed changes before applying them.

False negatives: DiffProv can fail for three reasons. First, the seeds of T_G and T_B have different types – for instance, the "good" event is a packet and the "bad" event is a flow entry. In this case, there is no valid solution, and the operator must pick a suitable reference. Second, the solution would involve changing an immutable tuple – for instance, a static flow entry that the operator has declared off limits, or the point at which a packet entered the network. In this case, there is again no valid solution, but DiffProv can show the operator what would need to be changed, and why; this should help the operator in picking a better reference. Third, DiffProv fails if it encounters rules that cannot be inverted (say, a `SHA256` hash). We have not encountered non-invertible rules in our case studies. However, if such a rule prevents DiffProv from going further, DiffProv can output the "attempted change" it would like to try, which may still be a useful diagnostic clue.

4.8 Extensions

Distributed operation: So far, we have described DiffProv as if the entire provenance trees T_G and T_B are materialized on a single node. We note that, in actual operation, DiffProv is decentralized: it never performs any global operation on the provenance trees, and all steps are performed on a specific vertex and its direct parent or children. Therefore, each node in the distributed system only stores the provenance of its local tuples. When a node needs to invoke an operation on a vertex that is stored on another node, only that part of the provenance tree is materialized on demand.

Temporal provenance: When DiffProv tries to make tuples appear, it must consider the state of the system "as of" the time at which the missing tuple would have had to exist, and it must apply the new updates to base tuples "early enough" to be present at the required time. DiffProv accomplishes the former by keeping a log of tuple updates along with some checkpoints, similar with DTaP [35], so that the system state at any point in the past can be efficiently reconstructed. DiffProv accomplishes the latter by applying the updates shortly before they are needed for the first time.

4.9 Limitations and open problems

We now discuss a few limitations of the DiffProv algorithm, and potential ways to mitigate some of them in future work.

Minimality: We note that the set of changes returned by DiffProv is not necessarily the smallest, since it attempts to derive missing tuples only via the specific rule that was used

to derive their counterpart in T_G. Other derivations may be possible, and they may require fewer changes. This is, in essence, the price DiffProv pays for using T_G as a guide.

Reference events: DiffProv currently relies on the operator to supply the reference event. This works well for the majority of the diagnostic cases we have surveyed (Section 2.4), where the operators have explicitly mentioned some potential reference events as starting points. But we are also exploring to automate this process using inspirations from Automatic Test Packet Generation [32] and the "guided probes" idea in Everflow [37].

Performance anomalies: Provenance in its plainest form works aims to explain individual events. We note that debugging performance anomalies, e.g., high per-flow latencies, resembles answering aggregation queries, and may require similar extensions to the current provenance model [2] that considers provenance for explaining aggregation results.

Non-determinism: Replay-based debuggers such as DiffProv, ATPG [32], etc., assume that the network is largely deterministic. In the presence of load-balancers that make random decisions, e.g., ECMP with a random seed, DiffProv would need to reason about the balancing mechanism using the seed. Under race conditions, DiffProv would abort at the point where applying the same rule does not result in the same effect, and suggest that point as a potential race condition.

5. IMPLEMENTATION

Next, we present the design and implementation of our DiffProv prototype. We have implemented a DiffProv debugger in C++ based on RapidNet [1], with five major components: a) a provenance recorder, b) a front-end, c) a logging engine, d) a replay engine, and e) the DiffProv reasoning engine.

Provenance recorder: The provenance recorder can extract provenance information from the primary system in three possible modes. First, it can directly *infer* the provenance if the primary system explicitly captures data dependencies, e.g., it is compiled into running code from declarative rules [18]. Since RapidNet is a declarative networking engine based on Network Datalog (NDlog) rules, DiffProv can infer provenance directly from any NDlog program; we applied this technique to the first three SDN scenarios.

Alternatively, the primary system can be instrumented with hooks that *report* dependencies to the recorder, e.g., as in [22]. We applied this to MapReduce by instrumenting Hadoop MapReduce `v2.7.1` to report its internal provenance to DiffProv. Our instrumentation is moderate: it has less than 200 lines of code, and it reports dependencies at the level of individual key-value pairs (e.g., words and their counts), as well as input data files, Java bytecode signatures, and 235 configuration entries.

Finally, we can treat the primary system as a black box, and use *external specifications* to track dependencies between inputs and outputs, e.g., as in [34]. We applied this to the complex SDN scenario in Section 6.7, where the recorder tracks packet-level provenance in Mininet [20] based on the packet traces it has produced, as well as an external specification of OpenFlow's match-action behavior.

Front-end: For our SDN scenario, we have built a front-end for controller programs that accepts programs written either in native NDlog or in NetCore (part of Pyretic [21]). When a NetCore program is provided, our front-end internally converts it to NDlog rules and tuples using a technique from Y! [30].

Logging and replay engines: The logging and replay engines are needed to support temporal provenance as described in Section 4.8, and they assist the recorder to capture provenance information in one of the following two approaches: a) in the runtime based approach, the logging engine writes down base events *and* all intermediate derivations, so that the provenance is readily available at query time; b) in the query-time based approach, the logging engine writes down base events *only*, and the replay engine then reconstructs derivations using deterministic replay. Although our prototype supports both approaches, we have opted for the latter in our experiments as it favors runtime performance – diagnostic queries would take longer, but they are relatively rare events; moreover, it enables an optimization that allows the replay engine to selectively reconstructs relevant parts of the provenance graph only.

Reasoning engine: The DiffProv reasoning engine retrieves the provenance trees from the recorder, performs the DiffProv algorithm we described in Section 4, and then issues replay requests to update the trees.

6. EVALUATION

In this section, we report results from our evaluation of DiffProv in two sets of case studies centered around software-defined networks and Hadoop MapReduce. We have designed our experiments to answer four high-level questions: a) how well can DiffProv identify the actual root cause of a problem?, b) does DiffProv have a reasonable cost at runtime?, c) are DiffProv queries expensive to process?, and d) does DiffProv work well in a complex network with realistic routing policies and heavy background traffic?

6.1 Experimental setup

The majority of our **SDN** experiments are conducted in RapidNet `v0.3` on a Dell OptiPlex 9020 workstation with an 8-core 3.40 GHz Intel i7-4770 CPU, 16 GB of RAM, a 128 GB OCZ Vector SSD, and a Ubuntu 13.12 OS. They are based on a 9-node SDN network setup similar with that in Figure 1, where we replayed an OC-192 packet trace obtained from CAIDA [7], as well as several synthetic traces with different traffic rates and packet sizes.

We further carry out an experiment on a larger and more complex SDN network, replicating ATPG's [32] setup of the Stanford backbone network. We replicated this setup because it is a network with complex policies and heavy background traffic, thus a suitable scenario to evaluate DiffProv's capability of finding root causes in a realistic setting. Since their setup involves a different platform (emulated Open vSwitch in Mininet [20] with a Beacon [4] controller), we defer the discussion of this experiment to Section 6.7.

Our **MapReduce** experiments are conducted in Hadoop MapReduce `v2.7.1`, on a Hadoop cluster with 12 Dell PowerEdge R300 servers with a 4-core 2.83 GHz Intel Xeon X33363 CPU, 4GB of RAM, two 250 GB SATA hard disks in RAID level 1 (mirroring), and a CentOS 6.5 OS. As a further point of comparison, we also re-implemented the MapReduce scenarios in a declarative implementation, and evaluated them in RapidNet.

6.2 Diagnostic scenarios

For our experiments, we have adapted six diagnostic scenarios from existing papers and studies of common errors. Our four SDN scenarios are:

- **SDN1: Broken flow entry [23].** An SDN switch is configured with an overly specified flow entry. As a result, traffic from certain subnets is mistakenly handled by a more general rule, and routed to a wrong server (T_B), while other traffic from other subnets continues to arrive at the correct server (T_G). This is the scenario from Section 2.
- **SDN2: Multi-controller inconsistency [10].** An SDN switch is configured with two conflicting rules by different controller apps that are unaware of each other. The lower-priority rule sends traffic to a web server (T_G), and the higher-priority rule sends traffic to a scrubber. However, the header spaces of the rules overlap, so some legitimate traffic is sent to the scrubber accidentally (T_B).
- **SDN3: Unexpected rule expiration [25].** An SDN switch is configured with a multicast rule that sends video data to two hosts (T_G). However, when the multicast rule expires, the traffic is handled by a lower-priority rule and is delivered to a wrong host (T_B). Notice that in this case the "good" example is a packet that was observed in the past.
- **SDN4: Multiple faulty entries.** In this scenario, we extended SDN1 with a larger topology and injected two faulty flow entries on two consecutive hops (S2–S3). Although some traffic can always arrive at the correct server (T_G), traffic from certain subnets is originally misrouted by S1 (T_{B1}), and then by S2 after the first fault is corrected (T_{B2}). As a result, DiffProv needs to proceed in two rounds to identify both faults.

Our MapReduce scenarios are inspired by feedback from an industrial collaborator about typical bugs he encounters in his workflow. Since the workflow is proprietary, we have translated the problems to the classical `WordCount` job example, which counts the number of occurrences of each word in a text corpus. We have evaluated them with a declarative implementation in RapidNet (MR1-D and MR2-D) and an imperative implementation in Hadoop's native codebase (MR1-I and MR2-I). The MR1 and MR2 scenarios are:

- **MR1-D and MR1-I: Configuration changes.** The user sees wildly different output files (T_B) from a MapReduce job he runs regularly, because he has accidentally changed the number of reducers. Because of this, almost all the emitted words end up at a different reducer node than before (T_G).
- **MR2-D and MR2-I: Code changes.** The user deploys a new implementation of the mapper, but it has a bug that causes the first word of each line to be omitted. As a result, the job now produces a different output (T_B) than before (T_G) for a previously used input file.

Query	**SDN1**	**SDN2**	**SDN3**	**SDN4**
Good example (T_G)	156	156	156	201/201
Bad example (T_B)	201	156	201	156/145
Plain tree diff	278	238	74	278/218
DiffProv	1	1	1	1/1
Query	**MR1-D**	**MR2-D**	**MR1-I**	**MR2-I**
Good example (T_G)	1051	1001	588	588
Bad example (T_B)	1051	848	588	438
Plain tree diff	164	306	240	216
DiffProv	1	1	1	1

Table 1: Number of vertexes returned by five different diagnostic techniques; for SDN4, the two rounds of DiffProv are shown separately. DiffProv was able to pinpoint the "root causes" with one or two vertexes in each case, while the other techniques return more complex responses.

6.3 Usability

We begin with a series of experiments to verify that differential provenance indeed provides a more concise explanation of the "root cause" than classical provenance. For this purpose, we ran two conventional provenance queries using Y! [30] to obtain the "good" and the "bad" provenance trees for each of the five diagnostic scenarios, as well as a differential provenance query using DiffProv. We also evaluated a simple strawman from Section 2.5, where we performed a plain tree diff based on the number of distinct nodes, in the hope that the querier would recognize suspicious gaps. We then counted the number of vertexes in each result.

Table 1 shows our results. As expected, the plain provenance trees typically contain hundreds of vertexes, which would have to be navigated and parsed by the human querier to extract the actual root cause. The plain diff is not significantly simpler – in fact, it sometimes contains *more* vertexes than either of the individual trees! (We have discussed the reason for this in Section 2.5.) Therefore, it would still require considerable effort to identify tuples that should not be there (e.g., flow entries that should not have been used) or to guess tuples that are missing. In contrast, differential provenance always returned very few tuples.

In more detail, for SDN1–SDN4, DiffProv returned the missing (or broken) flow entries as the root cause; for MR1-I, DiffProv returned `mapreduce.job.reduces` – the field in the configuration file that specifies the number of reducers; for MR2-I, though DiffProv cannot reason about the internals of the actual mapper code, it was still able to pinpoint the version of the mapper code (identified by the checksum of its Java bytecode) that caused the error; for MR1-D and MR2-D, DiffProv returned those fields' declarative equivalents in the NDlog model.

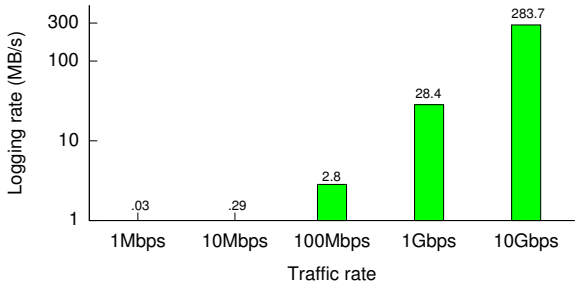


Figure 5: Logging rate for different traffic rates.

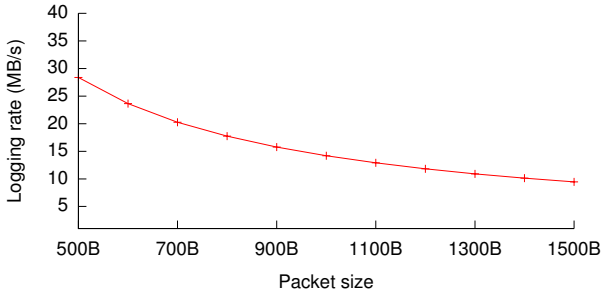


Figure 6: Logging rate with different packet sizes at 1Gbps.

To test how DiffProv handles unsuitable reference events, we issued ten additional queries in the SDN1 and MR1-D scenarios for which we picked a reference event at random. (We applied a simple filter to avoid picking events that we knew were suitable references.) As expected, DiffProv failed with an error message in all cases. In three of the cases, the supplied reference event was not comparable with the event of interest because their seeds had different types; for instance, one seed was a MapReduce operation but the other was a configuration entry. In the remaining seven cases, aligning the trees would have required changes to "immutable" tuples; for instance, the packet of interest entered the network at one ingress switch and the reference packet at another. In all cases, DiffProv's output clearly indicated what aspect of the chosen reference event was causing the problem; this would have helped the operator pick a more suitable reference.

6.4 Cost: Latency

Next, we evaluated the runtime costs of our prototype, starting with the latency overhead incurred by logging. For the SDN setup, we streamed 2.5 million 500-byte packets through the SDN1 scenario, and measured the average latency inflation of our prototype to process one packet when logging is enabled. For the MapReduce setup, we processed a 12.8 GB Wikipedia dataset in the MR1-I scenario, and recorded the extra time it took to run the same job with logging enabled. We observed that the latency is increased by 6.7% in the first experiment, and 2.3% in the second.

We note that our prototype was not optimized for latency, so it should be possible to further reduce this cost. For instance, the Y! system [30] was able to record provenance in a native Trema OpenFlow controller with a latency overhead of only 1.6%, and a similar approach should work in our setting. In the MapReduce scenario, the dominating cost was getting the checksums of the data files in HDFS. Instead of computing these checksums every time a file is read (as in our prototype), it would be possible to compute them only when files are created or changed. We tested this optimization in our prototype, and it reduced the latency cost to 0.2%.

6.5 Cost: Storage

Next, we evaluate the storage cost of logging at runtime. We varied the traffic rates in the SDN1 scenario from 1 Mbps to 10 Gbps, with the packet size fixed at 500 bytes, and then measured the rates of log size growth at the border switch. Figure 5 shows that the logging rate 1) scales linearly with the traffic rate, and 2) is well within the sequential write rate of our commodity SSD (400 MB/s), even at 10 Gbps. We also note that DiffProv does not maintain a log for every single switch, but only for border switches: a packet's provenance can be selectively reconstructed at query time through replay (Section 5). Therefore, if DiffProv is deployed in a 100-node network with three border switches, we would only need three times as much storage, not 100 times.

We performed another experiment in which we fixed the traffic rate at 1 Gbps and varied the packet sizes from 500 bytes to 1,500 bytes. Figure 6 shows that the logging rate decreases as the packet size grows. This is because 1) a dominating fraction of the log consists of the incoming packets, and 2) we only store fixed-size information for each packet, i.e., the header and the timestamp, not unlike in NetSight [13] or Everflow [37]: the latter has shown the feasibility of logging packet traces at data-center level with Tbps traffic rates. Moreover, the logs do not necessarily have to be maintained for an extensive period of time, and old entries can be gradually aged out to reduce the amount of storage needed.

Finally, we measured the storage cost in our MapReduce scenarios, where the logs were very small – 26 kB for the 12.8 GB Wikipedia dataset, and 1.5 kB for the 1 GB text corpus. This is because our logging engine records only the metadata of input files, not their contents: our replay engine can identify input files by their checksums upon a query, as long as those files are not deleted from HDFS.

6.6 Query processing speed

Diagnostic queries do not typically require a *real-time* response, although it is always desirable for the turnaround time to be reasonably low. To evaluate DiffProv's query processing speed, we measured the time DiffProv took to answer each of the queries. As a baseline, we measured the time Y! [30] took to answer each of the individual provenance queries for the "bad" tree only.

We first ran our SDN queries on a replay of an OC-192 capture from CAIDA, and the declarative MapReduce queries on a 1 GB text corpus. Figure 7 shows our result: except for SDN4, all other queries were answered within one minute;

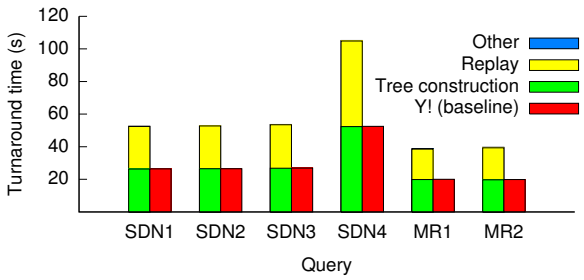


Figure 7: Turnaround time for answering differential provenance queries (left), and Y! queries (right). DiffProv's reasoning time (shown as "Other") is too small to be visible.

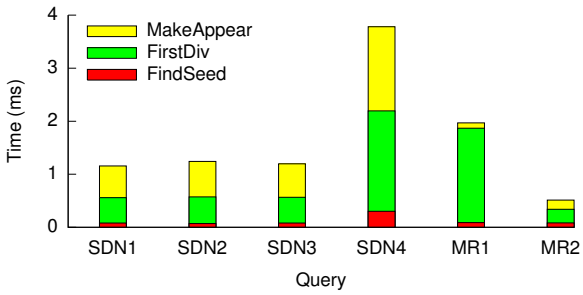


Figure 8: Decomposition of DiffProv's reasoning time. For SDN4, we have stacked its two rounds together.

the most complex DiffProv query (SDN3) was answered in 53.5 seconds. As the breakdown in the figure shows, query time is dominated by the time it takes to replay the log and to reconstruct the relevant part of the provenance graph. As a result, in each case, DiffProv queries took about twice as long as classic provenance queries using the Y! method: both DiffProv and Y! need a replay to query out the trees, but DiffProv replays a second time to update the bad tree after inserting the new tuple. Moreover, for SDN4, both Y! and DiffProv need to repeat this twice, once for each fault; therefore, both tools spent about twice as long on SDN4 as SDN1–SDN3.

If the reference event is contained in a separate, T'-second execution, DiffProv would take an additional T' seconds to replay and construct the reference tree. This is the case for our MapReduce queries that use a reference from a separate job. DiffProv performs three replays for those queries: once on the correct job, another on the faulty job, and a final one to update the tree. (In Figure 7, we have batched the first two replays to run in parallel, as they are independent jobs.) We then ran the imperative MapReduce queries on a larger, 12.8 GB Wikipedia data, without any batching: this time, Y! spent 349 seconds on MR1-I, and 336 seconds on MR2-I; DiffProv took three times as long as Y! in both cases.

We also observe that the actual DiffProv reasoning takes a negligible amount of time – 3.8 milliseconds in the worst case, as shown in a further decomposition in Figure 8. We can see that detecting the first divergence and making missing tuples appear took more time, because they involve tracking taints and evaluating their formulae. The SDN cases took more time in making tuples appear, because the missing (broken) flow entries were generated with more derivation steps. MR1-D took the longest time in divergence detection because its trees are deeper than those in all other cases.

6.7 Complex network diagnostics

Now that we have shown that DiffProv has a reasonably small overhead, we turn to evaluating the effectiveness of DiffProv's diagnostics on a complex network with real-world configurations and realistic background traffic.

Basic setup: Our scenario is based on the Stanford University network setup obtained from ATPG [32]; it represents a realistic campus network setting with complex forwarding policies and access control rules. The network has 14 Operational Zone (OZ) routers and 2 backbone routers that form a tree-like topology, and they are configured with 757,000 forwarding entries and 1,500 ACL rules. The routers are emulated with Open vSwitch (OVS) in Mininet [20], and controlled by a Beacon [4] controller. We also replicated their "Forwarding Error" scenario that involves two hosts and two switches, which we will refer to as H1, H2, and S1, S2, respectively: in the error-free setting, H1 should be able to reach H2 via a path H1-S1-S2-H2; however, S2 contains a misconfigured OpenFlow entry that drops packets to 172.20.10.32/27, which is H2's subnet. Please refer to [32] for a more detailed description on the configurations and the diagnostic scenario.

Multiple faults: Large networks are often misconfigured in more than one place, and their configuration tends to be changed frequently. The resulting "noise" can be challenging for debuggers that simply look for anomalies or recent changes. To demonstrate that DiffProv's use of provenance prevents it from being confused by bugs or changes that are not causally related to the queried event, we injected 20 additional faulty OpenFlow rules; 10 of them were on-path from H1 to H2, and the other 10 were on other OVS switches. We verified that the original fault we wanted to diagnose remained reproducible after injecting these additional faults.

Background traffic: To obtain a realistic data-plane environment, we ran three different applications in the network, and injected a mix of background traffic: 1) an HTTP client that fetches the homepage from a remote server periodically; 2) a client that downloads a large data file from a file server; 3) an NFS client that crawls the files in the root directory exported by a remote NFS server; and 4) we streamed the OC-192 trace from CAIDA through the network. The experiments took about 10 minutes, and produced 12GB packet captures, in which the `tshark` protocol analyzer detected 69 distinct protocol types.

Result: To diagnose the faulty event (i.e., a packet that is dropped midway from H1 to 172.20.10.32/27), we provided DiffProv with a reference event, which is a packet from H1 to 172.19.254.0/24: this is because we noticed that the subnets 172.19.254.0/24 and 172.20.10.32/27 are co-located in S2's operational zone, yet H1 is only able to reach the for-

mer. We queried out the provenance trees of the faulty event and the reference event. The trees are smaller than those in previous SDN scenarios, as this fault only involves two intermediate hops: they contain 67 and 75 nodes, respectively. Nevertheless, their plain differences contain as many as 108 nodes. We then used DiffProv to diagnose the fault - it correctly identifies the misconfigured OpenFlow entry on S2 to be the "root cause", despite the 20 other concurrent faults and the heavy background traffic.

At first glance, DiffProv's resilience to environments with substantial background traffic might seem surprising; in fact, DiffProv inherits this from the use of provenance, which captures true causality, not merely correlations. Note that this property sets our work apart from heuristics-based debuggers, e.g., DEMi [26] that is based on fuzzy testing, PeerPressure [29] that uses statistical analysis to find the likely value of a configuration entry, NetMedic [14] that ranks likely causes using statistical abnormality detection, and others. Those debuggers do not incur the overhead of accurately capturing causality, but may introduce false positives or negatives in their diagnostics as a result.

7. RELATED WORK

Provenance: Provenance is a concept borrowed from the database community [6], but it has recently been applied in several other areas, e.g., distributed systems [36, 34, 30], storage systems [22], operating systems [11], and mobile platforms [9]. Our work is mainly related to projects that use network provenance for diagnostics. In this area, ExSPAN [36] was the first system to maintain network provenance at scale; SNP [34] added integrity guarantees in adversarial settings, DTaP [35] a temporal dimension, and Y! [30] support for missing events. However, those systems focus on the provenance of individual events, whereas DiffProv uses an additional reference event for root-cause analysis. We have previously sketched the concept of differential provenance in a HotNets paper [8], but that paper did not contain a concrete algorithm or an implementation.

Network diagnostics: A variety of diagnostic systems have been developed over time. For instance, Anteater [19], Header Space Analysis [16], and NetPlumber [15] rely on static analysis, while OFRewind [31], Minimal Causal Sequence analysis [27], DEMi [26], and ATPG [32] use dynamic analysis and probing. Unlike DiffProv, many of these systems are specific to the data plane and cannot be used to diagnose other distributed systems, such as MapReduce. Also, none of these systems use reference events. As a result, they have the same drawback as the earlier provenance-based systems: they return a comprehensive explanation of each observed event and cannot focus on specific differences between "good" and "bad" events.

A few existing systems do use some form of reference: for instance, PeerPressure [29], EnCore [33], ClearView [24], and Shen et al. [28] use statistical analysis or data mining to learn correct configuration values, performance models, or system invariants. But none of them accurately capture causality, or leverage causality to reduce the space of candidate diagnoses. Attariyan and Flinn [3] does take causality into account, but it can only compare equivalent systems (e.g., "sick" and "healthy" computers), not events. NetMedic [14] also models dependencies, but it relies on statistical analysis and learning to infer the likely faulty component.

The idea of identifying the specific moment when a system "goes wrong" has appeared in other papers, e.g., in [17], which diagnoses liveness violations by finding a critical state transition. However, [17] does not use reference events, and its technical approach is completely different from ours.

Some existing solutions have packet recording capabilities that resemble the logging in DiffProv. For instance, NetSight [13] records traces of packets as they traverse the network, and Everflow [37] provides packet-level telemetry at datacenter scales. These systems reproduce the path a packet has taken, but not the causal connections, e.g., to configuration states. Provenance offers richer diagnostic information, and is applicable to general distributed systems.

8. CONCLUSION

Differential provenance is a way for network operators to obtain better diagnostic information by leveraging additional information in the form of reference events – that is, "good" and "bad" examples of the system's behavior. When reference events are available, differential provenance can reason about their differences, and produce very precise diagnostic information in return: the output can be as small as a *single* critical event that explains the differences between the "good" and the "bad" behavior. We have presented an algorithm called DiffProv for generating differential provenance, and we have evaluated DiffProv in two sets of case studies: SDNs and Hadoop MapReduce. Our results show that DiffProv's overheads are low enough to be practical.

Acknowledgments: We thank our shepherd Harsha V. Madhyastha and the anonymous reviewers for their comments and suggestions. We also thank Jeff Mogul, Behnaz Arzani, Yifei Yuan, and Chen Chen for helpful comments on earlier drafts of this paper. This work was supported in part by NSF grants CNS-1065130, CNS-1054229, CNS-1513679, CNS-1218066, CNS-1117052, CNS-1453392, and CNS-1513734; DARPA/I2O contract HR0011-15-C-0098; and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy.

9. REFERENCES

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

[2] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *Proc. PODS*, 2011.

[3] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *Proc. USENIX ATC*, 2008.

[4] The Beacon Controller. https://openflow.stanford.edu/display/Beacon/Home.

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

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

[7] CAIDA. http://www.caida.org/home/.

[8] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Differential provenance: Better network diagnostics with reference events. In *Proc. HotNets*, Nov. 2015.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proc. SIGCOMM*, August 2009.

[15] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI*, Apr. 2013.

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

[17] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proc. NSDI*, 2007.

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

[19] 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.

[20] Mininet. http://mininet.org/.

[21] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.

[22] 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 ATC*, 2009.

[23] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. SOSP*, 2009.

[25] J. Ruckert, J. Blendin, and D. Hausheer. Rasp: Using OpenFlow to push overlay streams into the underlay. In *Proc. P2P*, 2013.

[26] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *Proc. NSDI*, Mar. 2016.

[27] 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.

[28] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *Proc. SIGMETRICS*, 2009.

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

[30] 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.

[31] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proc. ATC*, 2011.

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

[33] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proc. ASPLOS*, 2014.

[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.

[37] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *Proc. SIGCOMM*, Aug. 2015.