Data Center Diagnostics with Network Provenance

Ang Chen[⋆], Chen Chen[◇], Yang Wu[†], Andreas Haeberlen[†],
Limin Jia[‡], Boon Thau Loo[†], Wenchao Zhou[♯]

[⋆]Rice University, [◇]Recruit Institute of Technology, [†]University of Pennsylvania,
[‡]Cargenie Mellon University, [♯]Georgetown University

Abstract

Diagnosing problems in data centers has always been a challenging problem due to their complexity and heterogeneity. Among recent proposals for addressing this challenge, one promising approach leverages provenance*, which provides the fundamental functionality that is needed for performing fault diagnosis and debugging—a way to track direct and indirect causal relationships between system states and their changes. This information is valuable, since it permits system operators to tie observed symptoms of a faults to their potential root causes. However, capturing provenance in a data center is challenging because, at high data rates, it would impose a substantial cost. In this paper, we introduce techniques that can help with this: We show how to reduce the cost of maintaining provenance by leveraging structural similarities for compression, and by offloading expensive but highly parallel operations to hardware. We also discuss our progress towards transforming provenance into compact actionable diagnostic decisions to repair problems caused by misconfigurations and program bugs.*

1 Introduction

Data center diagnostics has always been a challenging problem: with changing workloads, complex protocols, and a heterogeneous mix of devices, system faults can happen in data centers for many different reasons, including design flaws, software bugs, and security vulnerabilities. The recent move towards software-defined networking (SDN) has further added to this complexity: networks are now fully programmable. This provides the operator with great flexibility and power, but, at the samew time, a buggy controller program can introduce subtle malfunctions into the network that can be very difficult to find.

The research community has recognized this challenge [19], and has responded by developing a line of new diagnostic tools [20, 26, 27]. One approach, in particular, is based on *provenance* – a concept that was originally developed in the database community [4] but that has recently found new uses in the networking domain [9,52,59,61]. Provenance provides the fundamental functionality required for performing fault diagnosis and debugging—the capability to "explain" the existence (or change) of system state. Provenance is a form of metadata that tracks direct and indirect causal relationships between system states and state changes. Such information is of great value, especially in complex systems like data centers, since it permits system operators to tie observed faults to their potential root causes, and to assess the damage that these faults may have caused.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Provenance has been successfully applied to a wide range of areas, including distributed systems. In earlier work [8], we have already discussed some of the challenges and sketched generalizations of the provenance model that can help to address them. This paper focuses on our experience in adopting provenance for diagnosing data centers, and the unique challenges we faced during this process. First of all, given the ever-increasing amount of data processed and stored at data centers, the storage and computation overhead for provenance maintenance is high. This is especially important when provenance is used to track *data-plane* events – that is, the flow of actual network packets, as opposed to merely control messages. It is not unusual for the data plane to process data at a rate of several terabits per second, whereas control-plane messages are much less frequent.

We consider several techniques that could reduce the overhead of maintaining provenance. Besides traditional content-level compression techniques, such as gzip, we explore compression opportunities that leverage the structure of provenance trees [10]. We observe that the provenance of different packets share significant similarities in their structures, presenting opportunities for provenance compression across different provenance trees. For example, whenever a new packet traverses the network, its entire provenance tree is created and maintained. However, it is not hard to realize that *all* the packets in the same flow (i.e., packets with same source and destination) would take the same route and therefore have almost identical provenance trees. Therefore, we can achieve massive storage savings if we remove the observed redundancy when maintaining provenance. Maintaining provenance with security guarantees adds another layer of overhead—cross-nodes communications need to be cryptographically signed and acknowledged to be resilient to tampering of provenance [59]. To amortize the cost of these cryptographic operations, we leverage the use of Merkle Hash Trees (MHTs) and offload embarrassingly parallel operations to hardware such as FPGAs [7].

Another challenges is that provenance, at least in its raw state from runtime recording, often times is still hard for human users to understand. In fact, provenance trees can be quite large for system admins to reason about. For example, in an average-sized SDN, the provenance graph of why a specific packet p arrived at server s can easily contain hundreds of vertices. Identifying the root cause of an observed system problem is still a non-trivial task even for experts. To address this challenge, we consider approaches that distill from provenance compact and actionable information or suggestions for system admins. We explore an approach called differential provenance [9] that repairs misconfigurations through differential analysis on the provenance for working and non-working execution instances. The result of such differential analysis pinpoints the exact root causes that caused two execution instance to diverge. Meta provenance [50] takes a step further and reason about not only causality among data but also dependencies on program code. We treat the code as another kind of data, where syntactic elements of the program is captured as meta tuples and its semantics as meta rules. Such capability of reasoning dependencies of execution results on different components in a program empowers us to automatically repair bugs in the program. Given that there are literally an infinite number of possible way to modify a program, it is infeasible to use meta provenance to repair *all* possible bugs. We focus on bugs that do not require significant rewrite of the program.

In the remainder of the paper, we briefly introduce the concept of provenance and our system model in Section 2. Section 3 presents our recent progress in reducing the provenance maintenance overhead when applying provenance to data center networks. Section 4 further discusses our effort in transforming provenance into compact actionable diagnostic decisions. Finally, we present related work in Section 5 and conclude the paper in Section 6 with a discussion on future research directions in applying provenance to system diagnosis.

2 Background

Provenance is a general concept that has been applied to many kinds of systems written in many different languages. For ease of exposition, we will use a declarative language—specifically, *network datalog (NDlog)* [30]—to explain our approach. The main reason is that, in this language, it is relatively easy to introduce the causality relations that provenance needs to track.

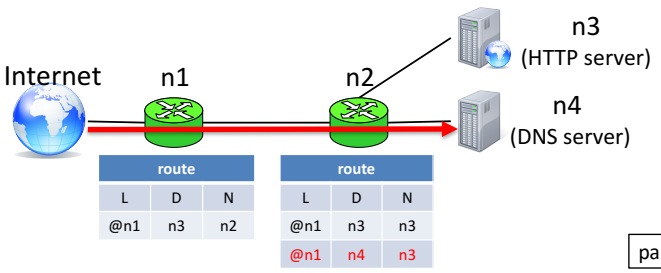


Figure 1: Example scenario.

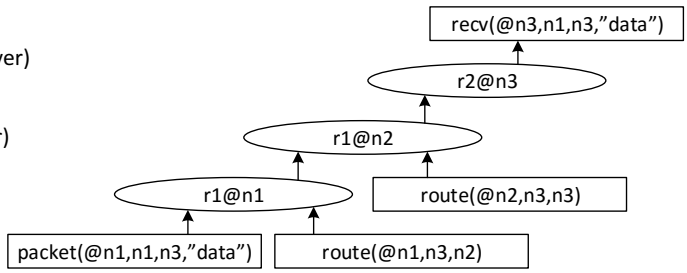


Figure 2: Example provenance.

In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*, each of which contains a number of *tuples*, such as configuration state or network events. For instance, an SDN switch might contain a table called `flowtable`, where each tuple represents a flow entry. Tuples can be manually inserted or programmatically derived from other tuples; in the former case, we refer to them as *base tuples*, and in the latter case as *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from each other. For example, the rule `r1: a(@X,P):-b(@X,Q),Q=2*P` says that a tuple `a(@X,P)` should be derived on node `X` whenever a) there is also a tuple `b(@X,Q)` on that node, and b) `Q=2*P`. The ID after the `@` symbol specifies the node on which the tuple resides, and the `:-` symbol is the derivation operator. Rules may include tuples from different nodes: for instance, `r2: c(@X,P):-c(@Y,P)` says that tuples in table `c` on node `Y` should be sent to node `X`.

In NDlog, the provenance of a tuple is easy to see, even syntactically: if a tuple (say, `a(@X,5)`) was derived using rule `r1` from above, then it must be the case that all the preconditions in `r1` were true (here, `b(@X,10)`), and all the constraints in `r1` were satisfied (in this case, `10=2*5`). This concept can be applied recursively to explain the existence of the preconditions until a set of base tuples (such as configuration settings or packets from external links) is reached that cannot be explained further. The result is a *provenance tree*, in which vertices represent tuples or rule derivations that generate these tuples, and edges represent direct causality.

Example Provenance. Figure 1 shows a (very simple) illustrative scenario with a network that connects two servers (one DNS server `n4` and one HTTP server `n3`) to the Internet. Here, switch `n1` and `n2` each have a routing table – each routing entry records the next hop towards a given destination. Each node runs an NDlog program consisting of two rules:

```
r1: packet(@N,S,D,Data) :- packet(@L,S,D,Data), route(@L,D,N).
r2: recv(@L,S,D,Data)   :- packet(@L,S,D,Data), D==L.
```

Rule `r1` forwards a packet to the next hop, based on the destination of the packet and the local routing information. Rule `r2` receives a packet if the packet's destination is identical to the local address.

Figure 2 sketches the provenance of an HTTP packet that was sucessfully delivered at the HTTP server `n3`. It reads roughly as follows: the packet arrived at switch `n1` and was routed to switch `n2` because it matched a routing entry `route(@n1,n3,n2)` installed at `n1`. The packet was further forwarded to and delivered at `n3`. Due to a faulty routing entry on switch `n2` (highlighted in red in Figure 1), DNS requests are misrouted to the HTTP server. To diagnose this problem, the operator can similarly query the provenance of a DNS packet, which would reveal that DNS requests were misrouted by `n3` after they arrive `n2` due to the faulty routing entry `route(@n1,n4,n3)`.

Application of Provenance in Diagnosing Distributed Systems. In prior work, we have already applied provenance in several common diagnostic scenarios. ExSPAN [61], our first solution, is a general-purpose system that can maintain and query provenance for large distributed systems. SNP [59] added security features that can prevent an adversary from tampering with the provenance to cover her tracks; this made it possible to use prove-

nance not just for diagnostics but also for forensics. Y! [52] introduced the concept of *negative* provenance, which can explain not only why a certain event did occur, but also why an event *failed* to occur.

3 Provenance Maintenance

The potentially large maintenance cost has been a long-standing challenge in provenance research [6] [24]. This becomes particularly challenging for systems like data centers that deal with frequent and high-volume data packets. When there are streams of incoming events, the provenance maintenance overhead, in terms of both computation and storage overhead, can quickly become prohibitively expensive. We present in this section two mechanisms for harnessing the maintenance overhead of provenance in data centers.

3.1 Provenance Compression through Program Analysis

Our first approach in reducing the maintenance overhead of provenance is based on the observation that there is much redundancy when maintaining provenance in a network. For example, when two packets share the same source and destination addresses, their provenances are almost identical: they share all the intermediate traversed nodes, and only differ at the packet payload. This suggests opportunity for significant storage reduction.

Based on this observation, we introduce an equivalence-based, online provenance compression mechanism, reducing the storage overhead of provenance effectively and efficiently [10]. This mechanism (1) groups provenance trees into equivalence classes, so that the storage of provenance trees of the same equivalence class could be compressed by sharing one representative provenance tree; and (2) enables efficient equivalence identification at runtime through the value comparison of *equivalence keys*—a subset of attributes of the input event identified by static analysis at compile time. Our experimental results demonstrate that this compression technique allows for significant—often orders of magnitude—storage reduction.

Provenance equivalence. One of our contributions is to show that the equivalence of provenance trees can be determined by comparing the values of a subset of attributes of the input tuple, if the program is written as a *Distributed Event-driven Linear Program'* (DELP). Intuitively, a DELP is an NDlog program that satisfy the following constraints:

- **Rules are event-driven.** Each rule *r* can be specified in the form: $[head] :- [event], [conditions]$, where $[event]$ is a body relation designated by the programmer, and $[conditions]$ are all non-event body atoms.
- **Rules are linearly dependent.** For consecutive rules r_i and r_{i+1}, the head relation *hd* of r_i is exactly the event relation in r_{i+1}.
- **Events don't have side-effects.** For each head relation *hd* in any rule r_i, *hd* is not used as a non-event relation in another rule r_j.

In a typical network application, events are high-speeding streaming relations that are often not materialized; non-event relations represent the network states. For example, in the packet forwarding program, the events are the `packet` tuples that flow through the network, and the *route* relation is a non-event relation, and is either updated manually or through a network routing protocol. In either case, it changes slowly compared to the fast-rate incoming packets. The distinction between events and non-event relations is provided by the programmer or is decided through profiling during program execution.

For DELP programs, we define the equivalence relation between two provenance trees *tr* and *tr'* as follows: *tr* and *tr'* are equivalent, if and only if they only differ at two nodes: the root node that represents the output tuple and the leaf node that represents the input tuple. This definition implies that two equivalent provenance trees are structurally identical too. For example, in Figure 1, if we insert another packet *packet*$(@n1, n1, n3, \text{“}data2\text{”})$, it will generate another provenance tree that is equivalent to the tree in Figure 2 – the only difference is the payload of the leaf *packet* tuple and the root *recv* tuple.

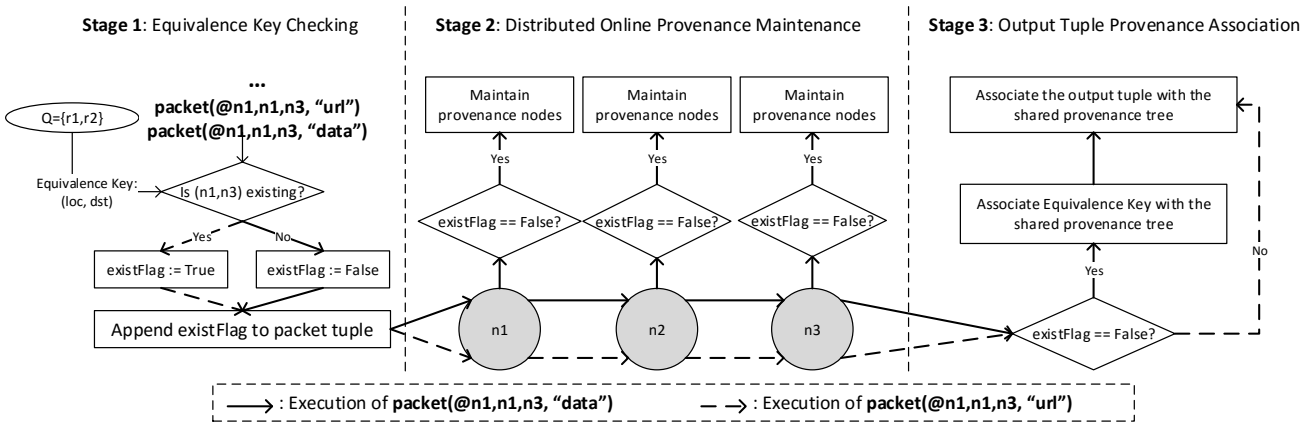


Figure 3: An example execution of the packet forwarding program in Section 2. The program is first triggered by *packet*($@n1, n1, n3$, "data"), followed by *packet*($@n1, n1, n3$, "url").

Based on the definition, the equivalence of two provenance trees can be determined through direct comparison: if all vertices other than the leaf tuple and the root tuple are identical, the two trees are equivalent. However, such comparison is inefficient, especially when provenance trees are large and the incoming tuples arrive at a fast rate, as is often the case with today's data centers.

We identified that, for DELP programs, equivalence classes can be determined by a subset of attributes in the events. We call these attributes *equivalence keys*, and show that they can be identified through static analysis of the program that generates the provenance trees. For example, the packet forwarding program in Section 2 is a DELP, and the equivalence keys are L, D of the *packet* tuple.

Online provenance compression. Once the equivalence keys are identified, we adopt an online provenance compression scheme that compresses equivalent (distributed) provenance trees. In our compression scheme, the execution of a DELP, triggered by an event tuple *ev*, is composed of three stages. Figure 3 presents an example execution of the provenance compression scheme, consisting of two packets traversing the network topology (from $n1$ to $n3$) in Figure 1. *packet*($@n1, n1, n3$, "data") is first inserted for execution (represented by the solid arrows), followed by *packet*($@n1, n1, n3$, "url") (represented by the dashed arrows). The three stages of online compression are logically separated with vertical dashed lines.

- **Stage 1: Equivalence keys checking.** Upon receiving an input event ev, the runtime system first checks whether the value of ev's equivalence keys have been seen before, by checking a hash table that stores all seen equivalence keys. If ev's equivalence keys have a value that already exists in the hash table, a Boolean flag $existFlag$ will be created and set to $True$. This $existFlag$ will accompany ev throughout the execution, notifying downstream nodes to avoid maintaining the concrete provenance tree. Otherwise, $existFlag$ is set to $False$. For example, in Figure 3, when the first packet $packet(@n1, n1, n3,$ "data") arrives, its equivalence keys ($n1$, $n3$) have never been encountered before, so its $existFlag$ is $False$. But $existFlag$ of the second packet $packet(@n1, n1, n3,$ "url") is set to $True$.
- **Stage 2: Online provenance maintenance.** For each rule r triggered in the execution, we selectively maintain the provenance information based on $existFlag$'s value. if $existFlag$ is $False$, the provenance nodes are maintained locally. Otherwise, no provenance information is maintained at all. For example, in Figure 3, when *packet*($@n2, n1, n3$, "data") triggers rule $r1$ at node $n2$, the $existFlag$ is $False$. Therefore, we maintain its provenance. In comparison, we simply execute $r2$ without recording any provenance information for *packet*($@n2, n1, n3$, "url").
- **Stage 3: Output tuple provenance maintenance.** For the execution whose $existFlag$ is $True$, we need to associate its output tuple to the shared provenance tree maintained by previous execution. To do this, we use a hash table $hmap$ to store the reference to the shared provenance tree, wherein the key is the hash

System	Goal	Information offered	Capabilities				
			Secure evidence	Supports forensics	Covers entire Internet	Fine-grained entities	Fine-grained traces
Tulip [32]	Fault localization	Loss, delay, reordering	×	×	✓	✓(Routers)	✓(Packets)
NetPolice [57]	Traffic differentiation detection	Loss	×	×	✓	× (ISPs)	× (Flows)
SPIE [46]	IP traceback	Backward routes	×	✓	✓	✓(Routers)	✓(Packets)
NetSight [21]	Network debugging	Packet histories	×	✓	×	✓(Routers)	✓(Packets)
Netdiff [33]	ISP performance benchmarking	Delay	×	×	✓	× (ISPs)	✓(Packets)
Paris-traceroute [3]	Load-balancer detection	Load-balanced routes	×	×	✓	✓(Routers)	✓(Packets)
HAL [17]	Packet attestation	Packet transmissions	✓	✓	×	✓(Links)	✓(Packets)
AudIt [2]	Performance accountability	Loss, delay	×	×	✓	× (ISPs)	✓(Both)
SPP	Single network-level primitive	All of the above	✓	✓	✓	✓(Routers)	✓(Packets)

Table 12: Comparison between SPP and some existing diagnostic and forensic primitives.

value of the equivalence keys, and the value is a pointer to the shared provenance tree. We then associate each output tuple of the same equivalence class to this shared representative provenance tree, by looking up the equivalence keys' values in $hmap$.

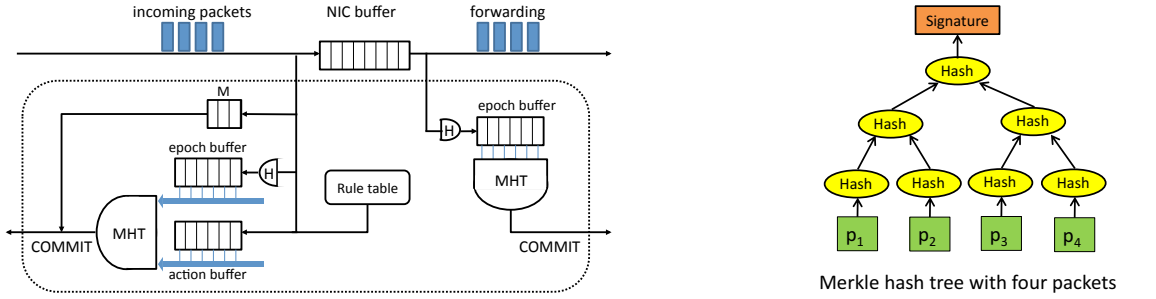
Our evaluation shows that the equivalence-based, online compression techniques achieve orders of magnitude reduction in storage and significant reduction in query latency, with only negligible network overhead added to each monitored network application at runtime.

3.2 Hardware-enabled Secure Provenance Collection

Another aspect of the challenge is to maintain provenance over high-speed traffic with security guarantees. Modern routers can process packets at Gigabits per second, so maintaining provenance, even without security guarantees, can result in non-trivial overhead; when we need to additionally achieve security guarantees, heavy-weight cryptographic operations are necessary, further amplifying the overhead. In [7], we designed Secure Packet Provenance (SPP), which addresses these challenges when maintaining per-packet provenance for high-speed Internet traffic. We used two key techniques to lower the overhead. First, we designed in SPP a lightweight protocol that can collect per-packet provenance data with tamper-proof evidence. The design avoids extensive use of heavy-weight cryptographic operations; instead, it uses Merkle Hash Trees (MHT), a data structure that allows efficient signatures over a large batch of packets. This greatly improves performance by reducing the processing logic mostly to hashing. Second, SPP offloaded the hashing and MHT construction to NetFPGAs to utilize its high parallelism, further improving the efficiency to maintain provenance over high-speed traffic.

Our key design of the SPP primitive is shown in Figure 4a. It consists of several modules: a) a *hash module*, which hashes each incoming packet at line speed, and stores the hash values at an *epoch buffer*; b) the *epoch buffer* temporarily holds packet hashes in the most recent batch, and sends the hashes to the *MHT constructor* for hash tree construction; c) the *MHT constructor* builds the MHT over the most recent epoch of packet hashes, and produces a top-level hash as the final commitment of the current batch; and d) the *loss detection* buffer (shown as M in the figure) identifies lost packets in the epoch, and feeds the information back to the sender so that the sender and receiver can "agree" on which packets have been successfully delivered. We have further sketched the MHT algorithm using an example of four packets in Figure 4b.

We have demonstrated that SPP is able to collect per-packet provenance data at line rate, and that it incurs low overheads in terms of storage and bandwidth. Moreover, with SPP, we have shown that many challenging tasks for Internet diagnostics and forensics can be approximated on top of this primitive with very few lines of code. Table 12 shows a comparison between SPP and other identified diagnostic and forensic tasks in the literature, and that network provenance can be a rather useful primitive that, once enabled, benefits many existing applications.



(a) The dataflow in SPP.

(b) An example Merkle Hash Tree.

Figure 4: The design of the SPP primitive.

4 Provenance for Diagnostics

Collecting and maintaining provenance is just the first step towards diagnosing data centers. Provenance often times is still hard for human users to understand. In fact, provenance trees can be quite large for system admins to reason about. For example, in an average-sized SDN, the provenance graph of why a specific packet p arrived at server s can easily contain hundreds of vertices [9, 52]. To address this challenge, we introduce two complementary approaches to distill from provenance compact and actionable suggestions for system admins.

4.1 Root Cause Detection with Differential Provenance

Our first approach explores the potential of repairing faults caused by misconfigurations through differential analysis across provenance trees [9]. The key observation is that misconfigurations, especially the subtle ones that need help of diagnostic tools, usually only affect a subset of traffic/nodes or only manifest themselves sporadically. Therefore, an operator typically has collected both working and non-working instances of similar traffic or service. Contrasting the provenance for the working and non-working instances is likely to provide more insights than studying only the non-working instance: it very much resembles the human debugging procedure—if we can understand why the two provenance are different and how to align them, we can likely identify the root cause of the problem. We call this approach differential provenance [9].

One challenge in differential provenance is that a small, initial difference can have a significant magnifying effect, for example, packets may take two completely different path and arrive at different destinations due to a small difference in the routing table of a gateway router. Therefore, a superficial "tree-diff" approach is likely to generate unusable results; our case studies have confirmed that the differences between working and non-working provenance trees can be even larger than the original provenance trees. To address this challenge, differential provenance identifies the first diverging point in the working and non-working provenance. We "roll back" the network execution to that point, change the mismatched tuple(s) on the non-working provenance tree to the correct version, and then "roll forward" the execution. We repeat this process until the two provenance are completely aligned.

Differential provenance has been demonstrated to be quite effective in a number of case studies on repairing software-defined networks and Hadoop MapReduce jobs. Our results show that it can always pinpoint exactly the induced misconfigurations to be the root cause of observed network problems.

4.2 Automated Program Patch with Meta Provenance

Differential provenance by itself is still fundamentally unable to help when a fault is caused by a bug in the program code: it can answer questions about the *data* in the network – such as "show me the configuration

entries that have caused this packet to be dropped" – but it treats the *code* as given and immutable, so it cannot answer questions such as "show me the line in the controller program that has caused this packet to be dropped".

To address this limitation, we have generalized traditional provenance to *meta provenance* [49,50]. Our idea is, essentially, to treat the code as another kind of data: we represent the syntactic elements of the program as a special class of tuples, which we call *meta tuples*, and we capture the semantics of the programming language (e.g., a language for SDN controllers, such as Pyretic [36]) with a set of special rules, which we call *meta rules*. For instance, an NDlog rule could be represented as meta tuples for the head and for each precondition, and one of the meta rules for NDlog could say that the tuple at the head of a rule is derived whenever all of the preconditions hold at the same time.

Once we have a meta model for a given language, we use a form of negative provenance [51,52] at the meta level to ask why some conditions did not hold. For instance, if the operator observes that a certain packet has been dropped and failed to reach a certain server, she can generate the negative provenance of the packet's arrival at that server (which is essentially asking "Why did this packet not reach that particular server?"). Negative provenance will then generate a recursive explanation, whose leaves include not just changes to data, but also changes to code; in this process, we use solvers such as Z3 [12] to find concrete values for the changes and thus generate repairs. Unlike positive provenance trees, negative provenance trees are typically extremely large, or even infinite—there are almost always many different ways to "fix" a given problem, ranging from small tweaks to rewriting the entire program. With the expectation that it is infeasible to fix programs that require significant or even complete rewrite, we prioritize modifications that minimize changes to the original program.

Once potential modifications are identified, we use "backtesting" to further examine each of them and evaluate whether it is actually a "correct" modification. That is, given the same system configurations and external inputs, the modified program should rectify faulty execution traces, and avoid altering the progress of execution traces that are previously correct. We leverage multi-query optimization [16, 31] from the database literature to speed up the backtesting, which enables us to validate multiple repairs in a single run.

We have applied meta provenance to subsets of three different SDN controller languages: NDlog [30], Pyretic [36], and Trema [48]. Results from several case studies show that our system can generate high-quality repairs for realistic bugs, typically in less than one minute.

5 Related Work

Provenance. Provenance is a concept initiated from the database community [4], but it has recently been applied in several other areas, including network systems [52, 58–61], storage systems [37], cloud computing platforms [23,40], and natural language processing [13]. Our work is mainly related to projects that use network provenance for diagnostics in distributed systems. In this area, ExSPAN [61] was the first system to maintain network provenance at scale; SNP [59] added integrity guarantees in adversarial settings, DTaP [60] a temporal dimension, and Y! [52] support for negative events. These systems generate an explanation on why some data exist or do not exist, but provide limited support for further diagnosis such as providing suggestions for system repair. There have been proposals for automating system repair based on provenance. Huang et al. [22] and Meliou et al. [34, 35] focus on instance-based explanations for missing answers, that is, how to obtain the missing answers by making modifications to the value of base instances (tuples); Why-Not [5] and ConQueR [47] provide query-based explanations for SQL queries, which reveal over-constrained conditions in the queries and suggest modifications to them.

System debugging and repair. Network debugging can be achieved by static analysis, e.g., as in Batfish [15], Header Space Analysis [26], NetPlumber [25], VeriFlow [27], Libra [55] rcc [14], or dynamic testing, e.g., as in Minimal Causal Sequence analysis [43], DEMi [42], OFRewind [53], and ATPG [54]. Some domain-specific languages, e.g., NetKAT [1], Flowlog [38], Kinect [28], can enable verification of specific classes of SDN programs. The software engineering community has used genetic programming [29] and symbolic

execution [39], for system repair; they are designed to fix small programs, or to propose specific kinds of fixes. ClearView [41] mines invariants in programs, correlates violations with failures, and generates fixes at runtime; ConfDiagnoser [56] compares correct and undesired executions to find suspicious predicates in the program; and Sidiroglou et al. [44] runs attack vectors on instrumented applications and then generates fixes automatically. These systems primarily rely on heuristics, whereas provenance-based approaches can track causality and can thus pinpoint specific root causes.

6 Future Directions

Timing causality. Faulty temporal behaviors can happen in distribute systems. For instance, suppose that a virtual machine takes unusually long to boot, because a misconfigured machine is overloading the shared storage backend. Existing tools, such as DTaP [60] or Dapper [45], offer little help with this scenario – in fact, the actual root cause (the misconfigured machine) would not even appear in the explanation! Because existing tools only capture *functional* causality, that is, events which directly contributed to the occurrence of the observed symptom, and will miss root causes that contributed only in terms of timing. We are currently studying ways to generalize provenance to track not only functional causality but also *temporal* causality, which is any event that has contributed to the timing of the observed symptom, regardless of whether it is functionally-related. This should allow provenance to explain faulty temporal behaviors.

Causality networks. Existing work on provenance only explains why a *single* event did or did not come about. While this is a useful starting point for diagnosis, it does not capture a range of other types of causality between events. For instance, operators may be interested to know why two events e_1 and e_2 always happen at the same time, or why e_1 and e_1 never happen at the same time, or even why only one of these events happens. To capture such complex, *inter-event* causality, we would need to extend the existing provenance abstraction to allow for reasoning about multiple events at the same time. Currently, we are looking at the possibility to extend provenance to causal networks [18], which encodes the inter-dependencies among events, and then using causal networks to answer queries of the more complex form. This advanced form of provenance may be helpful to process diagnostic queries that cannot be easily answered with the current provenance tree abstraction.

Probabilistic causality. Another intriguing direction is to develop models and efficient evaluation for probabilistic provenance. Unlike traditional provenance models, the inputs and derivations themselves are set to true or false based on a probabilistic distribution. This is motivated by multiple scenarios. First, the use of machine learning and model-based prediction means that a significant portion of causality relationships are inherently probabilistic in nature. Second, in some environments, the derived provenance themselves can be subjected to probabilistic distributions, for example, on a lossy communication channel, the likelihood of links being up or packets being transmitted correctly may themselves be probabilistic. Finally, in the case of applications written in a non-declarative language, the input-output dependencies themselves may be inferred probabilistically. A key challenge is to maintain provenance for *all* possible combinations of variable assignment. Inspired by prior work on probabilistic databases, a potential solution is to develop a provenance model based on the *possible-worlds semantics* [11]. An associated challenge is potential of state explosion when evaluation probabilistic provenance. One possible approach to address this challenge is to explore tradeoffs between accuracy and performance (e.g., query latency, and communication overhead).

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[2] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and delay accountability for the Internet. In *Proc. ICNP*, 2007.

[3] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proc. IMC*, 2006.

[4] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, pages 316–330, 2001.

[5] A. Chapman and H. V. Jagadish. Why not. In *Proc. SIGMOD*, 2009.

[6] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proceedings of ACM SIGMOD*, pages 993–1006, 2008.

[7] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proc. Eurosys*, 2017.

[8] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, , and W. Zhou. Data provenance at internet scale: Architecture, experiences, and the road ahead. In *Proc. of CIDR*, 2017.

[9] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of ACM SIGCOMM 2016*, Aug. 2016.

[10] C. Chen, H. T. Lehri, L. K. Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou. Distributed provenance compression. In *Proc. SIGMOD*, 2017.

[11] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: Diamonds in the dirt. *Commun. ACM*, 52(7):86–94, July 2009.

[12] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, 2008.

[13] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. In *Proc. of VLDB Endowment*, 2017.

[14] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. NSDI*, Boston, MA, May 2005.

[15] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[16] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. 5(6):526–537, 2012.

[17] A. Haeberlen, P. Fonseca, R. Rodrigues, and P. Druschel. Fighting cybercrime with packet attestation. Technical Report MPI-SWS-2011-002, Max Planck Institute for Software Systems, July 2011.

[18] J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British journal for the philosophy of science*, 56(4):843–887, 2005.

[19] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *Proc. HotSDB*, 2012.

[20] 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*, 2014.

[21] 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*, 2014.

[22] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, 1(1), Aug. 2008.

[23] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. In *Proc. of VLDB Endowment*, 2015.

[24] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *Proceedings of ACM SIGMOD*, pages 951–962, 2010.

[25] 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*, 2013.

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

[27] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proc. NSDI*, 2015.

[29] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. ICSE*, 2012.

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

[31] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, 2002.

[32] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *Proc. SOSP*, 2003.

[33] R. Mahajan, M. Zhang, L. Poole, and V. Pai. Uncovering performance differences among backbone ISPs with Netdiff. In *Proc. NSDI*, 2008.

[34] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *Proc. SIGMOD*, 2011.

[35] A. Meliou, W. Gatterbauer, and D. Suciu. Bringing Provenance to its Full Potential using Causal Reasoning. In *Proc. TaPP*, 2011.

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

[37] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, 2006.

[38] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proc. NSDI*, 2014.

[39] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. ICSE*, May 2013.

[40] H. Park, R. Ikeda, and J. Widom. RAMP: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12), 2011.

[41] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proc. SOSP*, 2009.

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

[43] 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.

[44] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 2005.

[45] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. In *Google Technical Report*, 2010.

[46] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, B. Schwartz, S. Kent, and W. Strayer. Single-packet IP traceback. *IEEE/ACM Trans. Netw.*, 10(6):721–734, 2002.

[47] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proc. SIGMOD*, 2010.

[48] Trema: Full-Stack OpenFlow Framework in Ruby and C. https://trema.github.io/trema/.

[49] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets'15)*, Nov. 2015.

[50] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. In *Proc. NSDI*, 2017.

[51] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering why-not queries in software-defined networks with negative provenance. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets-XII)*, Nov. 2013.

[52] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems negative provenance. In *Proceedings of ACM SIGCOMM 2014*, Aug. 2014.

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

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

[55] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. NSDI*, 2014.

[56] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proc. ICSE*, 2013.

[57] Y. Zhang, Z. M. Mao, and M. Zhang. Detecting traffic differentiation in backbone ISPs with NetPolice. In *Proc. IMC*, 2009.

[58] Y. Zhang, A. O?Neil, M. Sherr, and W. Zhou. Private network provenance. In *Proc. of VLDB Endowment*, 2017.

[59] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *Proc. SOSP*, 2011.

[60] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB'13)*, Aug. 2013.

[61] 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.