Data Provenance at Internet Scale: Architecture, Experiences, and the Road Ahead

Ang Chen*
University of Pennsylvania

Yang Wu*
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Boon Thau Loo
University of Pennsylvania

Wenchao Zhou
Georgetown University

ABSTRACT

Provenance is a way to answer "why" questions about computations. It has found a number of uses in the database community, such as debugging query answers or tracing unexpected results to database tuples. In fact, the ability to ask "why" can be useful for a much broader range of applications. In this paper, we summarize our experiences over the past few years in adapting provenance for diagnostic and forensic uses in networks and distributed systems. Our work draws inspirations from database provenance, yet the deployment scale, use cases, and distributed nature of networks require a significant re-design of traditional data provenance models. We review a number of use cases, ranging from investigating intrusions to diagnosing (and even automatically fixing) software-defined networks, and present a unified system architecture that we have designed and implemented for provenance in distributed systems. We conclude with a discussion of open issues in this space.

1 Introduction

Provenance is, in essence, a way to answer "why" questions about computations. It works by linking each effect, such as a computed value, to its direct causes, such as the corresponding inputs and the operation that was performed on them. Thus, it becomes possible to recursively "explain" a particular result by showing its direct causes, and then their own causes, and so on, until a set of basic inputs is reached. This explanation is the *provenance* of the result: it is a tree whose vertexes represent computations or data and whose edges indicate direct causal connections.

Provenance originated in the database community [13], and it has found a number of interesting uses, such as diagnosing query answers [31], reverse data management [34], or tracking unexpected results to specific tuples in the input data [32]. However, the concept itself is not database-specific: it can potentially help in any situation where a system has shown some unexpected behavior that must now be investigated and tracked to a set of "root causes". This kind of situation is common in many areas of computer science.

Over the past several years, we have been working on *network provenance*, which is a general class of solutions that adapt provenance for diagnostic and forensic uses in computer networks and, more generally, distributed systems. As with database applications, these systems frequently do something that their operators did not expect. In the context of database applications, a SQL query may have been written incorrectly, resulting in erroneous query results. Likewise, in computer networks, a network operator may observe unusual routes or dropped packets, which may be symptoms of errors in network configurations, or worse still, bugs introduced by intentional manipulation and even targeted attacks. And, just as in the database world, the operators are faced with the challenging problem of tracing the observed symptoms to a set of root causes.

One way to apply provenance to distributed systems is to essentially model the distributed system as a giant database: the state of each node can be stored in tables, and the programs can be modeled as a set of declarative rules [29]. The provenance of each tuple can then be tracked just like it would be in a database, and diagnostic queries (say, "Why is the network using route R to reach host H?") can be formulated as provenance queries (say, "What is the provenance of route R?"), which then reveal the corresponding root causes (say, a recent configuration change).

Given the distributed nature of networks, we can partition and distribute the state such that each node maintains only a portion of the information required to reconstruct any tuple provenance, rather than storing all the network state in a centralized database. This was the approach we took in our first solution [53], but it quickly became clear that there were a number of additional challenges. One simple example is the fact that network state, unlike tuples in traditional databases, tends to be short-lived: by the time that the operator is notified of the problem with route R, that route may already have been replaced by another. We solved this by adding a temporal dimension to the provenance, so that questions could be asked about past states [52]; however, this massively increased the amount of metadata that needed to be kept, so we added garbage collection and a range of strong compression techniques, along with a cost model to choose the "right" technique for a given system [52].

Other challenges were more fundamental, however. For instance, one interesting application area is security. Given that the provenance graph in our setting is stored in a distributed and open fashion across administrative domains, the entire system is vulnerable to attacks. The operator might wish to learn how an attacker "got into" the system, or what changes she has made to it. However, if the attacker has compromised the system, what prevents her from tampering with the provenance and giving false or misleading responses? In another class of scenarios in network debugging, the problem is not the presence of an unexpected event, but rather the *absence* of an *expected* event; here, it is not immediately clear how to even apply provenance, since there is no starting point for a possible explanation. Our solutions to both problems involve new data structures, as well as substantial re-designs and refinements of the

*These authors contributed equally to this work.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017, Chaminade, California, USA.

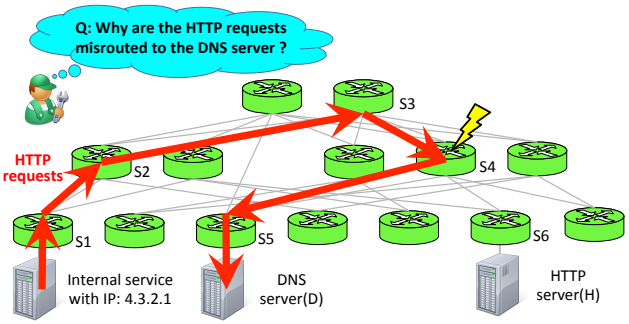

Figure 1: An example diagnostic scenario.

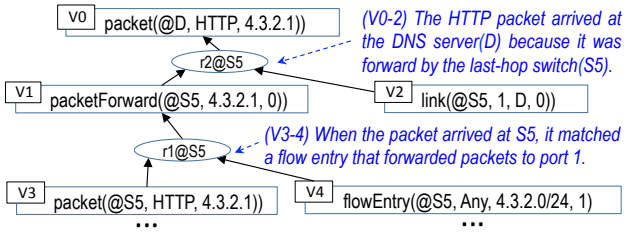

Figure 2: The provenance tree for a misrouted packet (excerpt).

provenance graph [50, 45, 46].

Along the way, we also discovered several interesting variants of the problem that provenance was designed to solve. For instance, operators often wish not only to diagnose a problem (say, a software bug) but also to find a suitable fix. Existing solutions in the area of automated program repair tended to involve trying out random program mutations, which seemed inefficient. By lifting provenance to cover not only data but also code, we were able to specifically target the parts of a program that contributed to a faulty outcome, which can yield higher-quality fixes [43, 44]. Another example is the fact that operators often have additional information available, e.g., in the form of outputs that are similar to the faulty output but are nevertheless computed correctly (say, an unexpected network path versus a correct path). In this case, we found it useful to return not the provenance of the faulty output but the provenance of the *differences* between the correct and the faulty output; this is often enough to identify a single "root cause" [15, 16].

This paper is our attempt to (1) unify our point solutions over the years, (2) present a range of applications of provenance in the distributed systems and networking space, (3) discuss some lessons we learned during this work, and (4) highlight inspirations from the database literature and new insights introduced by our environment.

2 Motivating Example

Figure 1 shows an example network debugging scenario. A network operator manages a network consisting of switches and servers, and he is alerted that some HTTP requests are misrouted to the DNS server. To diagnose this, the operator needs a comprehensive explanation of the problem, and some aid on how to repair it. This is just a small example of a wide variety of problems can happen in a distributed system: switches can be misconfigured [42], hosts can be hacked [2], and the control software can have bugs [39]. This is further complicated by the complexity of today's networks – even a medium-sized network can contain as many as 757,000 routing entries and 1,500 access control rules [47].

Provenance can help with network diagnostics, because it can track the causal connections from a network symptom to a set of root causes, leaving out irrelevant factors which can become a source of confusion. For instance, we have drawn an example provenance tree for the misrouted HTTP packet in Figure 2, which can explain how the packet traversed the network (e.g., `V0`, `V1`, `V3`), the series of routing table matches (e.g., `V4`), and how the routes were computed (we omitted this from the figure, but `V4` can be further explained by how the route was computed by the routing protocol). This provides a useful starting point for the operator to understand the symptom, find the root cause(s), and roll out a fix.

Over the years, the networking community has developed a suite of diagnostics tools, but very few of them are based on provenance. For instance, existing approaches include statistical learning [26, 6, 42, 3, 48], fuzz testing [39, 38], and distributed tracing [54, 27, 17, 49, 4, 28], just to name a few. Since they do not track causality in a systematic manner as in provenance, the identified root cause(s) and proposed repair(s) may have false positive and/or false negatives as a result. "Backtraces" akin to provenance have also been used for network diagnostics, but they typically track much less information than what provenance does. For instance, NetSight [21] can produce a packet's path through a network and the routing entry matches, but does not explain how the routing entries were computed in the first place. Therefore, provenance seems to be a suitable candidate for enabling new ways of network debugging.

However, in order to use data provenance at Internet scale, and to provide support for network diagnostics, significant enhancements to traditional data provenance has to be made. This includes maintaining and querying provenance in a *distributed* setting, processing provenance queries on *historical* system state, ensuring the *integrity* of provenance records, explaining why an event *did not* happen, identifying a concise *root cause* from a large provenance tree, and lifting provenance to a "meta" level to *suggest repairs* for buggy networks. In the next two sections, we discuss each of those challenges and the proposed enhancements in more detail.

3 Towards network provenance

First, we provide a starting point for network provenance, as proposed in ExSPAN [53]. We then present a series of enhancements to add support for temporal queries, security guarantees, and queries on missing events. We use the scenario in Figure 1 as a running example, where certain HTTP requests are misrouted to a DNS server due to network misconfigurations.

3.1 Starting point: Distributed provenance

Figure 3 shows a part of the routing program in our example scenario. We assume that the distributed system is written in Network Datalog (NDlog), which is a distributed variant of Datalog proposed in declarative networking [29]. (See Section 5.1 for discussion on capturing provenance for non-NDlog systems.)

Figure 2 shows an excerpt of the provenance tree of a misrouted HTTP packet, as model by ExSPAN [53]. It has two vertex types: a) a TUPLE vertex (shown as a rectangle) corresponds to a network event or configuration state; and b) a RULE vertex (shown as an oval) corresponds to a step in the network execution which derive new tuples from existing ones. The tuples can be further categorized as base (EDB) tuples and derived (IDB) tuples.

ExSPAN differs from traditional data provenance (e.g., ORCHESTRA [24]) in three ways. First, the provenance graph is partitioned and *distributed* across many nodes. For instance, the `packet(@S5,HTTP,4.3.2.1)` tuple is stored at node `S5`, as indicated by the location symbol `@`. As a result, distributed recursive queries are

```
r1: packetForward(@Swi,Pro,Dip,Sip,Prt) :- packet(@Swi,Ipt,Pro,Dip,Sip), flowEntry(@Swi,Ipt,Pro,Dip,Sip,Prt).
r2: packet(@Swi',Ipt',Pro,Dip,Sip) :- packetForward(@Swi,Pro,Dip,Sip,Prt), link(@Swi,Prt,Swi',Ipt').
```

Figure 3: Part of the NDlog program that describes our example scenario. A routing entry matches an incoming packet and forwards it to an outgoing port (r1). A packet reaches a neighboring device by traversing a link that is connected to the device (r2).

needed to collect a tuple's provenance. Second, the provenance graph is *dynamic* – packets can arrive at any time, and routing entries can be inserted or updated. To address this, ExSPAN uses incremental view maintenance on the provenance information [29]. Third, ExSPAN is targeted at Internet-scale deployments with relatively small network state per node, unlike in a traditional database where tens of nodes host a large amount of data. Therefore, the design of ExSPAN focuses on network-centric metrics, such as reducing communication overhead, minimizing query latency, and avoiding slowing down existing protocols' convergence speed.

3.2 Time-awareness

However, there are diagnostic queries that ExSPAN cannot answer. As in traditional databases, ExSPAN only maintains the provenance for the *current* system state, and forgets about tuples the moment they disappear. This is not enough for *network* diagnostics: although many network events and state are short-lived, an operator may still need to ask queries about them after they expire.

Our approach is to extend the provenance model with a temporal dimension [52], and to introduce additional vertexes in the provenance graph. We use EXIST vertexes to record the lifespan of tuples, INSERT, DELETE vertexes to record the actions performed on base tuples, APPEAR, DISAPPEAR vertexes to record the histories of derived tuples, and DERIVE, UNDERIVE vertexes to record how and when tuples acquired or lost support. The new vertex types all carry a timestamp field, and they persist even after the tuples they describe expires. Therefore, the graph comes append-only.

A key concern in this append-only model is the maintenance and querying overhead. To address this, we allow provenance to be maintained *proactively* or *reactively*, depending on the desired tradeoff. The proactive approach logs all provenance tuples; this requires more storage, but provides lower query latencies. The reactive approach only logs base tuples, nondeterministic events (e.g., incoming packets and their order), and optionally, some intermediate snapshots; upon a query, provenance can be reconstructed using deterministic replay; this requires less storage, but may incur query processing delays. Our query optimizer has a cost model that can automatically choose the best approach for a given use case.

Moreover, several other key design decisions have to be reworked due to the distinct challenges in distributed systems, such as loosely synchronized clocks, interactions via message passing, heterogeneous computing resources, just to name a few. We refer interested readers to [52] for more details.

3.3 Security

If performing network diagnostics were our only goal, the above model would already go a long way; however, we also have a need for secure *forensics*, since today's networks and computers are targets of malicious attacks. In an adversarial setting, compromised nodes can tamper with its local data and/or lie to other nodes. For instance, when answering a provenance query, an adversary may return fabricated provenance to frame innocent parties and derail the investigation. Therefore, when performing forensics, it is important to validate the integrity of the provenance data.

To address this, we introduce additional vertex types to capture cross-node interactions, and validate them using secure cryptographic primitives [50]. The new vertex types include SEND, RECEIVE, and BELIEVE. When a node N derives a tuple τ because it received another tuple τ' from N', our enhanced model would reflect this by inserting a BELIEVE tuple on N, which encodes N's belief that τ' indeed appeared on node N'; it would also insert corresponding SEND and RECEIVE tuples, recording how N developed that belief by receiving τ' from N'. If we later find out that τ' in fact never existed on N' (i.e., N' lied about τ'), N and its belief are still considered innocent. Those vertexes are further stored in a tamper-evident log, and secured by a commitment scheme [20], so a node can always *prove* about a past interaction with other nodes.

We observe that, even with those enhancements, it is still inherently impossible to guarantee that *all* queries can be correctly answered: how can we force compromised nodes to respond to our query, or prevent them from wiping out their local data? Nevertheless, this new model can provide two (weaker but) provable guarantees: a) if any behavior is *observable* by at least one correct node, then the provenance of that behavior will be correct, and b) if a correct node detects a misbehavior, it can tell which part of the provenance is affected, and attribute the misbehavior to at least one faulty node. Our prototype shows that this model works well for BGP routing, Chord DHT, and Hadoop MapReduce applications.

3.4 Negative provenance

So far, our provenance model explains why something happened, like many of its database counterparts, but not why something good failed to happen. But our survey shows that, a common diagnostic task in distributed systems is to explain the *absence* of an expected event [46] – e.g., why a certain route is not available.

To support this, we have developed an approach we call *negative provenance* [46], which further extends the provenance model by adding negative vertexes, and which builds negative provenance trees using *counterfactual reasoning*. At a high level, negative provenance finds all possible ways in which a missing event could have occurred, and the reasons why each of them did not happen. As a result, this enhancement creates a "negative twin" for each vertex type (except for BELIEVE), e.g., SEND/NSEND, INSERT/NINSERT.

Negative provenance is related to "why-not" queries in the database literature, which explain why a particular tuple is absent from a query result. The negative vertexes in our model share similar roles with proxy tuples [23] or c-tuples [22], which also represent missing tuples that should have been derived or inserted. To explain a missing tuple, Huang et al. [23], Artemis [22], and Meliou et al. [34] take an instance-based approach, which aim to obtain missing answers by editing base tuples; Why-Not [14], ConQueR [40], and Why-Not polynomials [9, 8, 10] take a query-based approach, which suggests changes to query operators and query plans.

However, "why-not" questions pose unique challenges in a networking context. Since a negative provenance graph can, in principle, contain a vertex for every tuple that could potentially exist, the graph can have a large number of vertexes or even be infinite. To tackle this problem, we use a top-down procedure to construct the negative provenance of a given event "on demand", without materializing the entire graph. Moreover, our algorithm can simplify the provenance even further by pruning unhelpful branches, and by summarizing common patterns (e.g., a packet being forwarded

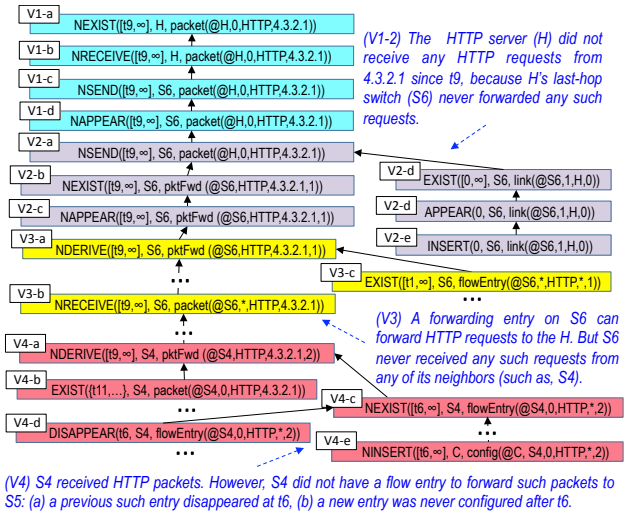

Figure 4: The provenance tree for a missing HTTP request, as generated in our final provenance model.

along its path), so that the final provenance tree is much more compact and readable. We have applied negative provenance to a range of diagnostic scenarios in software-defined networks (SDN) and Internet routing. Please refer to [46] for more details.

3.5 Final provenance model

Starting from the basic provenance model, we have enhanced the vertex types, developed novel provenance algorithms, and addressed several practical challenges unique to distributed systems. As a result, our final provenance model is significantly different from that in traditional databases. Figure 4 shows a provenance tree for our running example in this new model, which contains substantially richer information than the initial version. Here, the operator asks a "why-not" question: "Why did the HTTP server *not* receive any HTTP requests?". We note that a) besides explaining the presence of a unexpected behavior, it can also explain the absence of the desired behavior; b) it contains new vertex types, such as SEND and RECEIVE, which can attribute actions to specific nodes (we omitted the BELIEVE vertexes); and c) it contains past events and state, e.g., the disappearance of an expired routing entry.

4 Automated Network Repair

With these enhancements, we have arrived at a more comprehensive model that can answer far more useful queries than the model we started with. But so far, our model has focused on explaining what did or did not happen in the network, but it cannot answer questions of the form "How should we *change* the network state, so that a wrong output, such as a wrong route, can be fixed?". Such queries are analogous to their database counterparts – "how-to" queries [34], and are a form of reverse data management [33]. They are more challenging to answer, but can potentially help the operator a lot more, as they can identify the root cause of a problem and/or suggest a repair. In the networking context, we refer to this problem as *automated network repair*. Next, we discuss two applications of provenance in automated network repair.

4.1 Differential provenance

Understanding how to change an underlying database so that query answers become correct is an important problem in the database literature. Examples include Tiresias [34] that answers "how-to" queries by formulating them as mixed integer programs, functional causality [31] that ranks the responsibility of database tuples in a given query result, and view-conditioned causality (VCC) [32] that correlates correct and incorrect results to narrow down likely culprits, and others. Among these, VCC is most closely related to our first application, which repairs network misconfigurations by a differential analysis across provenance trees.

Although network misconfigurations are quite common, they oftentimes only affect a *subset* of traffic/nodes, and they only manifest infrequently. Therefore, an operator typically has both working and non-working instances of similar traffic or service. Our key insight is that, in those cases, we have more diagnostic information that classic provenance is using: if we reason about the *differences* between the provenance trees for the working and non-working instances, it would be much easier to identify the root cause of the problem. We call this approach *differential provenance* [15, 16].

Differential provenance shares a similar insight with VCC, but it works quite differently. One challenge in networks is that a small, initial difference can lead to wildly different network executions afterwards – for instance, a different routing decision may send packets down an entirely different path. Therefore, the differences between working and non-working provenance trees can be much larger than one might expect. Differential provenance addresses this with a counterfactual approach: it "rolls back" the network execution to a point where the provenance trees start to diverge; then, it changes the mismatched tuple(s) on the non-working provenance tree to the correct version, and the "rolls forward" the execution until the two trees are aligned.

But now a second challenge arises. The two provenance trees describe inherently different network executions (e.g., how two different packets traversed the network); therefore, short of changing one packet to another, we cannot hope to align the trees *completely*. How, then, should we preserve the packets as invariants, and at what point should the alignment terminate? Differential provenance introduces an "equivalence" relation between two provenance trees, and uses this to distinguish between differences that need to be aligned, and differences that need to be preserved.

We have applied differential provenance in a number of case studies on repairing software-defined networks and Hadoop MapReduce jobs. Our results show that reasoning about the differences of provenance is quite effective – it can pinpoint one or two misconfigured entries to be the root cause of the problem.

4.2 Meta provenance

When generating network repairs, differential provenance only considers changes to network configuration data, but not how computations are performed on the data. Today's networks, however, can also be affected by wrong computations due to buggy router software, especially in SDNs where a controller program can modify router behaviors dynamically – a significant departure from traditional hardware-only routers. Our second application can repair *both* network configuration data *and* controller program, using a novel data structure that we call *meta provenance* [43, 44].

This problem also has its counterparts in the database literature, which consider modifications to relational queries so that they yield expected results on the underlying data [14, 40, 9, 8, 10]. However, these database approaches face some challenges in the networking setting. For instance, a common assumption is that there are known ways to identify "unpicked" [14] or "compatible" tuples [8], which are tuples in the source database that satisfy a user's criteria of desired tuples but are absent from the result. In the context of networks, such tuples are difficult (if not impossible) to identify. For

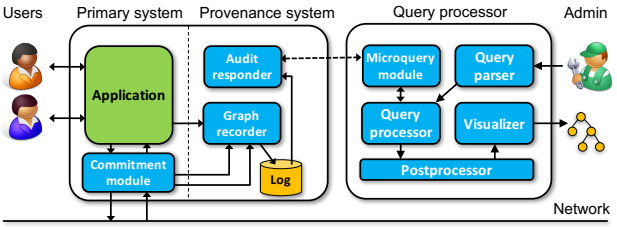

Figure 5: System architecture.

example, if a router is missing a routing entry, it is very challenging for the operator to know upfront which set of routing entries–distributed across hundreds of switches–can derive the missing entry, and using which route computation algorithm(s).

In meta provenance, we take a fundamentally new approach to this problem, by extending provenance to reason about changes to *both* program *and* configuration data. Meta provenance has a set of *meta tuples*, which represent the syntactic elements of the program itself, and a set of *meta rules*, which describe the operational semantics of the programming language. We next use negative provenance (Section 3.4) to ask why some conditions did not hold. The result is a set of changes to the program and/or to the configuration data that makes the condition become true.

One challenge is that there can be infinitely many potential repairs. To overcome this, we leverage the fact that most bugs affect only a small part of the program, and that programmers tend to make certain errors more often than others [37]. This allows us to rank the possible repairs according to plausibility, and to explore only the most plausible ones. Another challenge is that repairs can have *side-effects*: they may cause new problems elsewhere in the network. To avoid such repairs, we backtest candidates by replaying them with historical information collected in the network.

Moreover, we leverage *multi-query optimization* [19, 30] from the database literature to speed up the backtesting, which can backtest multiple repairs in a single run. In [43, 44], we have validated this approach by finding high-quality repairs for moderately complex SDN programs written in NDlog, Trema [41], and Pyretic [35].

5 System Architecture

Figure 5 shows the architecture of the system that we have designed and implemented. The runtime components include the primary system that runs the actual network program, and the provenance system that collects diagnostic and forensic information about the primary system. The query processing components answer diagnostic queries. Overall, we have implemented the core components of our system using more than 67,000 lines of code in C++. In this section, we describe each component in more detail.

5.1 Runtime components

The runtime components include a) a *graph recorder* that extracts provenance from a node's primary application and records it in a local log; b) a *commitment module* that adds cryptographic commitments to outgoing messages and verifies the commitments in incoming messages to detect tampering; and c) an *audit responder* that sends excerpts from the log to the query processor when they are needed to respond to a query.

Since these components must run continuously even if there are no queries, it is important to keep the overhead as low as possible. To this end, the graph recorder does not materialize the provenance graph; rather, it records only just enough information to reconstruct the graph if and when necessary. Typically, it is enough to record just the application's inputs and any nondeterministic events. When a query arrives, our system can selectively reconstruct an application's state and its provenance using deterministic replay [52].

The graph recorder can extract the provenance in three ways, depending on the application. If the application is written in a declarative language, such as NDlog, the provenance can be extracted transparently through the runtime. Otherwise, if the source code is available, the application can be instrumented to record the provenance explicitly by making calls to the recorder, as, e.g., in PASS [36]. If the application is only available in binary and cannot be changed, provenance can still be extracted through an external specification [50].

The commitment module is only activated when the system must work under attack; its purpose is to make the log tamper-evident [20]. This requires, among other things, performing cryptographic operations on outgoing and incoming messages. To keep the corresponding overhead low, messages can be processed in batches.

5.2 Query processing components

The query processing components do not necessarily reside on the same nodes with the runtime components; they are instead used as diagnostic applications that can run separately, for example, on an operator's laptop. There are five components, as shown in the figure: a) a *query parser* that accepts provenance queries from the operator; 2) a *microquery module* that can, given a single vertex in the provenance graph, find the parents and children of that vertex; 3) a *query processor* that uses microqueries to construct larger subtrees needed for answering the query; 4) a *postprocessor* that improves the readability of the provenance, e.g., through aggregation or by reasoning about differences; and 5) a *visualizer* that displays the provenance and allows the operator to explore it interactively.

Our system contains several instances of the microquery module, because this module is specific to the provenance model; but the instances expose a similar set of APIs for compatibility, and they can be dynamically swapped in and out. The microquery module for negative provenance [46] is more complex than the others because there are more vertex types that need to be supported. Another factor that adds complexity is the threat model: our module instance for secure provenance [50] must not only reconstruct the provenance graph but also authenticate it and check for tampering. If a node is found to have tampered with some part of its (local) provenance, the corresponding vertexes are annotated accordingly; the system also guarantees that the actions of a compromised node cannot affect the provenance of a correct node.

In the prototype we have built, there is no nontrivial query language: the operator simply describes a symptom – say, a message that was sent by a particular node – and thus identifies a vertex in the provenance graph. (However, this is not inherent; more complex provenance query languages, e.g., from [25], could be very useful.) The query processor can be as simple as starting from this vertex and expanding the subtree that is rooted at it. However, in use cases where the provenance graph is conceptually infinite (in particular, negative provenance), the query processor has to do more work, e.g., to prune inconsistent or nonsensical subtrees.

When provenance is tracked at such a fine granularity, the resulting provenance trees can be very large, which complicates the operator's job. To mitigate this problem, our postprocessor implements several summarization techniques [46] that can aggregate certain parts of the graph, e.g., a message being forwarded across several hops, and thus reduce the size of the graph in the common case. Our visualizer, NetTrails [51], further improves usability. A more sophisticated form of postprocessing is differential provenance [16], which pinpoints only those vertexes that are causal to the differences between two provenance trees.

6 Open Problems

Looking ahead, we believe that the confluence of data provenance and networking is creating many more exciting research opportunities; we sketch some of them below.

Provenance on sensitive network data: Using provenance to diagnose network problems that span multiple trust domains remains an open problem. Provenance is easier to collect within a single trust domain, e.g., in a centralized database or an enterprise network; but collecting and analyzing provenance *while providing strong privacy guarantees* seems to be a challenging problem, e.g., when performing diagnostics in a multi-tenant cloud. One candidate that could help is differential privacy [18], which offers provable privacy guarantees; secure multi-party computation [7] and/or trusted hardware platforms (e.g., Intel's SGX [1]) may also be able to help, though it remains to be investigated whether they would incur too much overhead when applied to provenance.

Provenance on high-speed network data: Existing uses of provenance have mostly targeted at control-plane protocols that compute routes, such as the BGP protocol, SDN controller programs, etc.; but it remains to be seen whether provenance can work as well for the Internet's data plane used for forwarding packets, where the data rates are on the order of 1-100 Gbps. In fact, even the Internet's backbone itself needs to process such data with highly optimized, hardware-based routing fabric. Therefore, for provenance to work for the data plane, we may need to incorporate hardware elements in our system architecture, e.g., at least for the components that need to operate at linespeed, such as the graph recorder and commitment modules in Figure 5. Fortunately, we could leverage open hardware platforms, such as NetFPGA [11] or P4 [12], for this purpose. New algorithms may also be required to achieve high compression rates on any stored provenance data.

Provenance for timing faults: Our focus so far has been on diagnosing wrong computation results, much like in the database counterparts. However, this does not cover a broad class of problems that plagues distributed systems – timing faults. For instance, a web client may have received a response correctly, but the delay was much higher than usual. In those cases, provenance cannot be used out-of-the-box, since it only captures *functional, but not timing* causality, so we may need to further enhance the provenance model to capture the latter. Moreover, diagnosing timing faults involves capturing provenance for aggregation queries [5], which is more challenging; this may be needed when explaining why there is a drop in average throughput or an increase in tail latency.

Acknowledgments: Some of this work was done in collaboration with Zachary G. Ives and Micah Sherr. We are also grateful to Susan B. Davidson and Val Tannen for many valuable discussions. This work was supported in part by NSF grants CNS-1054229, CNS-1065130, CNS-1117052, CNS-1218066, CNS-1453392, CNS-1513679, and CNS-1513734.

7 References

[1] Intel SGX. https://software.intel.com/en-us/sgx.

[2] Recent Zero-Day Exploits. https://www.fireeye.com/current-threats/recent-zero-day-attacks.html.

[3] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *Proc. NSDI*, 2009.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. SOSP*, 2003.

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

[6] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with NetPoirot. In *Proc. SIGCOMM*, 2016.

[7] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP - A system for secure multi-party computation. In *Proc. CCS*, 2008.

[8] N. Bidoit, M. Herschel, and A. Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *Proc. CIKM*, 2015.

[9] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *Proc. TaPP*, 2014.

[10] N. Bidoit, M. Herschel, and K. Tzompanaki. Refining SQL queries based on why-not polynomials. In *Proc. TaPP*, 2016.

[11] M. Blott, J. Ellithorpe, N. McKeown, K. Visssers, and H. Zeng. FPGA research design platform fuels network advances. *Xilinx Xcell Journal*, 2010.

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

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

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

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

[16] 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 *Proc. SIGCOMM*, 2016.

[17] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: end-to-end performance analysis of large-scale Internet services. In *Proc. OSDI*, 2014.

[18] C. Dwork. Differential privacy. In *Proc. ICALP*, 2006.

[19] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. In *Proc. VLDB*, 2012.

[20] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, Oct. 2007.

[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] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. In *Proc. VLDB*, 2010.

[23] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. In *Proc. VLDB*, 2008.

[24] Z. Ives, N. Khandelwal, A. Kapur, and M. Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *Proc. CIDR*, 2005.

[25] Z. G. Ives, A. Haeberlen, T. Feng, and W. Gatterbauer. Querying provenance for ranking and recommending. In *Proc. TaPP*, 2012.

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

[27] N. Khadke, M. P. Kasick, S. P. Kavulya, J. Tan, and P. Narasimhan. Transparent system call based performance debugging for cloud computing. In *Proc. MAD*, 2012.

[28] E. Koskinen and J. Jannotti. BorderPatrol: isolating events for black-box tracing. In *Proc. EuroSys*, 2008.

[29] 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.

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

[31] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. Why so? or why no? functional causality for explaining query answers. In *Proc. MUD*, 2010.

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

[33] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(11):1490–1493, 2011.

[34] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *Proc. SIGMOD*, 2012.

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

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

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

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

[39] 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.

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

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

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

[43] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proc. HotNets*, 2015.

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

[45] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering why-not queries in software-defined networks with negative provenance. In *Proc. HotNets*, 2013.

[46] 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.

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

[48] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *Proc. EuroSys*, 2013.

[49] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proc. OSDI*, 2014.

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

[51] W. Zhou, Q. Fei, S. Sun, T. Tao, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. NetTrails: A declarative platform for maintaining and querying provenance in distributed systems. In *Proc. SIGMOD Demo*, 2011.

[52] 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.

[53] 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.

[54] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos. NetCheck: Network diagnoses from blackbox traces. In *Proc. NSDI*, 2014.