2017

Secure Diagnostics And Forensics With Network Provenance

Ang Chen
University of Pennsylvania, saxtonac@gmail.com

Secure Diagnostics And Forensics With Network Provenance

Abstract
In large-scale networks, many things can go wrong: routers can be misconfigured, programs can be buggy, and computers can be compromised by an attacker. As a result, there is a constant need to perform network diagnostics and forensics. In this dissertation, we leverage the concept of provenance to build better support for diagnostic and forensic tasks. At a high level, provenance tracks causality between network states and events, and produces a detailed explanation of any event of interest, which makes it a good starting point for investigating network problems.

However, in order to use provenance for network diagnostics and forensics, several challenges need to be addressed. First, existing provenance systems cannot provide security properties on high-speed network traffic, because the cryptographic operations would cause enormous overhead when the data rates are high. To address this challenge, we design secure packet provenance, a system that comes with a novel lightweight security protocol, to maintain secure provenance with low overhead. Second, in large-scale distributed systems, the provenance of a network event can be quite complex, so it is still challenging to identify the problem root cause from the complex provenance. To address this challenge, we design differential provenance, which can identify a symptom event's root cause by reasoning about the differences between its provenance and the provenance of a similar "reference" event. Third, provenance can only explain why a current network state came into existence, but by itself, it does not reason about changes to the network state to fix a problem. To provide operators with more diagnostic support, we design causal networks – a generalization of network provenance – to reason about network repairs that can avoid undesirable side effects in the network. Causal networks can encode multiple diagnostic goals in the same data structure, and, therefore, generate repairs that satisfy multiple constraints simultaneously. We have applied these techniques to Software-Defined Networks, Hadoop MapReduce, as well as the Internet's data plane. Our evaluation with real-world traffic traces and network topologies shows that our systems can run with reasonable overhead, and that they can accurately identify root causes of practical problems and generate repairs without causing collateral damage.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Andreas Haeberlen

Keywords
Diagnostics, Forensics, Provenance, Root-Cause Analysis, Software-Defined Networks

SECURE DIAGNOSTICS AND FORENSICS WITH NETWORK PROVENANCE

Ang Chen

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2017

Andreas Haeberlen, Associate Professor of Computer and Information Science
Supervisor of Dissertation

Lyle Ungar, Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee:
Boon Thau Loo, Associate Professor of Computer and Information Science (Chair)
Zachary G. Ives, Professor of Computer and Information Science
Vincent Liu, Assistant Professor of Computer and Information Science
Wenchao Zhou, Assistant Professor of Computer Science, Georgetown University

SECURE DIAGNOSTICS AND FORENSICS WITH NETWORK PROVENANCE

Soli Deo Gloria

Acknowledgments

First and foremost, I thank my advisor, Andreas Haeberlen, for taking me under his wing. Andreas has been a wonderful advisor to me, providing a constant source of wisdom, guidance, and support over the five years of my Ph.D. journey. Were it not for his mentorship, I would not be here today.

I also thank my dissertation committee members, Boon Thau Loo, Zack Ives, Vincent Liu, and Wenchao Zhou. In addition to providing valuable feedback on this dissertation, they have also given me much helpful advice over the years.

I am very grateful for Boon's mentorship on research, life, and beyond. His encouragement and support have been instrumental in my journey. Likewise, I thank Wenchao for being a close mentor and friend to me.

This dissertation would not have been possible without my collaborators, whom I am fortunate to have worked with, including Andreas Haeberlen, Boon Thau Loo, Micah Sherr, Wenchao Zhou, Linh Thi Xuan Phan, Clay Shields, as well as fellow students Yang Wu, Hanjun Xiao, Brad Moore, Mingchen Zhao, Yuankai Zhang, Tavish Vaidya, Chirag Shah, Bob DiMaiolo, Max Demoulin, and Akshay Sriraman. I have also greatly enjoyed the enlightening discussions with Zack Ives, Val Tannen, Rajeev Alur, Joe Devietti, Mayur Naik, Ani Nekova, Chris Callison-Burch, Dave Walker, and Jennifer Rexford. Moreover, I wish to thank my mentors and collaborators before Penn: Rocky Chang, Edmond Chan, Xiapu Luo, Peng Zhou, Weichao Li, Daoyuan Wu, Ka Pui Mok, Wai Ting Fok, Byron Gao, Zhaohui Cai, Chunwu Yu, and Jingsong Cui.

The companions at CIS@Penn have made this journey full of fond memories, for which I thank Yang Wu, Hanjun Xiao, Mingchen Zhao, Antonis Papadimitriou, Arjun Narayan, Yifei Yuan, Chen Chen, Yang Li, Behnaz Arzani, Meng Xu, Nimit Singhania, Steven Wu, Nathan Dautenhahn, Kevin Tian, and Wenrui Meng. My gratitude also goes to my long-time friends, Jihong Qiao and Jinfeng Huang, Huan Luo, Yue Li, Dongwei Yu, Peng Yi, Yaopeng Ge, and Yunfei Song.

I wish to thank Micah Sherr, Jennifer Rexford, Vincent Liu, and T. S. Eugene Ng for their support in my job search, as well as many others whom I cannot even list here exhaustively.

I am extremely grateful to have Tenth Presbyterian Church and Tenth International Fellowship as my spiritual home in Philadelphia, and to walk closely with my mentors and companions – Rev. Enrique Leal, Rev. Bruce McDowell, Rev. Carroll Wynne, Rev. Jinan Zhang, Lani and Steve Shade, Joe and Yoon Park, Nasrat Ghattas, Yang He, Yige Zhou, Yi-Lin Chiang, Stephen Russell, Xiaoping Chen, Antonis Papadimitriou and Thenia Karavasili, Guiyun Zhang and Biao Zuo, Darryl John, Saki and Charinet Georganas, Jerry Joyce, Evangeline Sim, Martin and Machiko Whittaker, Teng Zhang, Yingchun Liu, Ping Wei, Marie Liu, Russ Pfeifer, Candy Chen, Hung Bui, Qianhui Lin, Robert Johnson, Randall Drain, Nick Lamelza, and Tom Jackson. I also thank Yongquan Yan and Yingting Jiao for their friendship, and Sr. Mary Ann McIntyre, M.S.B.T., Rev. Doh Chun Ning, and Fr. Dan Joyce, S.J., for their spiritual direction and prayers.

I thank my parents for their love, support, and encouragement. They have taught me by example the value of hard work and persistence. Although they are half the world away, they have always been there for me.

Last but not least, it is my tremendous blessing to have known and married my wife, Xin, who has made this long journey worthwhile. Words cannot express how thankful I am to have her always by my side. I am deeply indebted to Xin for her loving care and unwavering support, and I dedicate this dissertation to her.

ABSTRACT

SECURE DIAGNOSTICS AND FORENSICS WITH NETWORK PROVENANCE

Ang Chen

Andreas Haeberlen

In large-scale networks, many things can go wrong: routers can be misconfigured, programs can be buggy, and computers can be compromised by an attacker. As a result, there is a constant need to perform network diagnostics and forensics. In this dissertation, we leverage the concept of *provenance* to build better support for diagnostic and forensic tasks. At a high level, provenance tracks causality between network states and events, and produces a detailed explanation of any event of interest, which makes it a good starting point for investigating network problems.

However, in order to use provenance for network diagnostics and forensics, several challenges need to be addressed. First, existing provenance systems cannot provide security properties on high-speed network traffic, because the cryptographic operations would cause enormous overhead when the data rate is high. To address this challenge, we design secure packet provenance, a system that comes with a novel lightweight security protocol, to maintain secure provenance with low overhead. Second, in large-scale distributed systems, the provenance of a network event can be quite complex, so it is still challenging to identify the root cause of a problem from the complex provenance. To address this challenge, we design differential provenance, which can identify a symptom event's root cause by reasoning about the differences between its provenance and the provenance of a similar "reference" event. Third, provenance can only explain why a current network state came into existence, but by itself, it does not reason about changes to the network state to fix a problem.

To provide operators with more diagnostic support, we design causal networks – a generalization of provenance – to reason about network repairs that can avoid undesirable side effects in the network. Causal networks can encode multiple diagnostic goals in the same data structure, and, therefore, generate repairs that satisfy multiple constraints simultaneously. To validate these techniques, we have applied them to Software-Defined Networks, Hadoop MapReduce, as well as the Internet's data plane. Our evaluation with real-world traffic traces and network topologies shows that our systems can run with reasonable overhead, and that they can accurately identify root causes of practical problems and generate repairs without causing collateral damage.

Contents

List of Tables

List of Figures

1

Introduction

Distributed systems have become critical infrastructure in our life. They enable many important network services, including online banking, video streaming, and electronic medical records, and they interconnect a wide range of devices, ranging from desktop computers, mobile phones, to Internet-of-Things devices, such as networked printers and cameras. Over time, these systems have also grown in size and complexity – commercial data centers typically consist of tens of thousands of servers [19], and even campus networks can have hundreds of thousands of access control list rules and forwarding entries [101]. Moreover, due to the advent of Software-Defined Networks (SDN), today's distributed systems are also highly dynamic, as their configurations can be constantly changed by software controllers.

In such large-scale, complex systems, many things can go wrong. As the many incident reports on Outages [15] and NANOG [25] mailing lists can attest, even well-maintained networks can experience a variety of problems – links can fail [70], nodes can be misconfigured [167], and software controllers can have bugs [168]. In the infamous AS7007 incident [6], for instance, a domain accidentally announced

routes to a large portion of the Internet and caused disconnectivity for two hours. In a more recent case, Amazon's S3 storage service went down for more than four hours simply due to a mistyped command, bringing down 150,000 websites with it [3]. These outages are very costly – major data centers can lose half a million dollars in a five-minute outage [1], and Amazon's typo command alone was estimated to have caused an overall economic loss of $150 million [2].

To respond to these problems, network operators need to perform *diagnostics* – understanding why a symptom event came about, e.g., why a packet was misrouted, identifying the problem root cause, e.g., a certain misconfigured flow entry, and rolling out a fix to bring the network back to a good state. Unfortunately, this can be quite challenging due to the complexity of today's distributed systems. Blindly digging through system logs and configurations is no easier than finding needles in a haystack. Operators' jobs would be easier if there is a way to understand the causal relations between network events. However, this is not straightforward in a complex network, because it requires reasoning about the interplay between a myriad of factors in the network, and problems can happen due to seemingly "off-path" causes [162]. The difficulty in reasoning about causality also complicates the network repair process, because pushing out a configuration change has network-wide effects and may accidentally cause damage. In fact, there is no shortage of "death-by-recovery" incidents from major data centers [62, 72, 129].

Adding to this complexity, problems can also happen due to adversarial attacks, ranging from brute-force Denial-of-Service (DoS) attacks [26] to vulnerability exploits [61] to subtle side channel [164] or covert channel attacks [49]. Handling attacks is even trickier, because attackers may lie about their actions and plant false data to cover their tracks [10]. Therefore, we also need the ability to perform network *forensics* to track down attackers. Forensic techniques need to be robust even in an adversarial environment, because unreliable forensics can lead to false accusations or even send innocent people to jail [160, 16, 22].

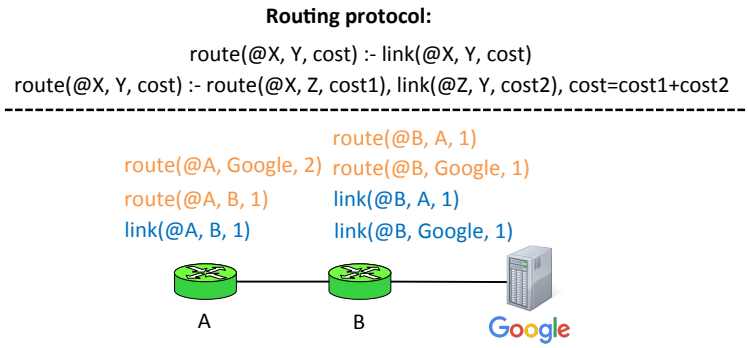


Figure 1.1: An example network with two routers that run a routing protocol. A `link(@X,Y,C)` (`route(@X,Y,C)`) representation means that there is a link (route) between `X` and `Y` with the cost of `C`. The routing protocol computes network routes from links.

1.1 A PROVENANCE-BASED APPROACH

Over the years, researchers have proposed many systems to implement different diagnostic and forensic tasks [118, 110, 37, 157, 113, 41, 98, 99, 162, 64, 80, 119, 135], but they tend to be point solutions that address specific problems. For instance, ICING [135] proposes to extend the Internet architecture so that users can validate network paths that packets have taken, NetReview [80] proposes to allow network domains to audit each other, detect BGP problems, and attribute the problems to misbehaving domains, Paris-traceroute [41] aims to uncover load-balanced paths in the Internet, and Netdiff [119] aims to measure and compare performance differences of ISPs. Each of these tasks represents an important goal. However, many of the existing solutions tend to be problem-specific – solutions developed for one task rarely help with other tasks. As a result, in order to address all identified problems, it is necessary to deploy all solutions, which can be cumbersome and costly. Moreover, sometimes, the proposed solutions may require extensions that are incompatible with each other.

In this dissertation, we propose to build better support for diagnostics and forensics in a more systematic fashion using a technique called *network provenance* [183].

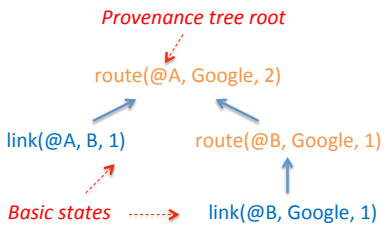


Figure 1.2: An example provenance tree that describes why there is a route between `A` and `Google` with a cost of `2`.

At a high level, provenance tracks causality between network states and events in the form of a directed graph, where vertexes represent network states and events, and edges represent causality. Consider the network in Figure 1.1 as a simplified example, which consists of two routers (each with its local routing table), two links, and a server. When applied to this scenario, provenance would track the causality between routes and links, based on how the routing protocol executes.

Such tracked causality can be helpful for diagnostics and forensics, because it can explain why a certain network state or event came about – for instance, how a particular route was computed. A user can ask provenance queries, e.g., about a certain route, and the provenance system would return its *provenance tree*, where the root represents the event of interest, its children represent the direct causes, until we arrive at the leaves in the tree, which represent basic states and events in the network. Figure 1.2 shows an example provenance tree for `route(@A,Google,2)`, or the route between `A` and `Google` with a cost of 2, in the example network in Figure 1.1. Since network provenance only tracks the relevant factors for a particular diagnostic task, operators can safely leave out any irrelevant factors. For instance, to understand why the route was computed, an operator can narrow down her focus to the nodes present in the route's provenance tree.

There has been existing work on networking provenance, including ExSPAN [183] and DTaP [182] that maintain provenance for distributed systems, SNP [180] that cryptographically signs the provenance data to provide security, and Y! [169] that

explains why a certain expected network event failed to happen. However, there are at least three challenges that have not been addressed in previous work. First, basic provenance does not provide the security and privacy properties needed for Internet diagnostics and forensics, where networks span multiple domains, carry high-speed data, and could even be compromised. SNP [180] is a useful starting point for securing provenance data, but it does not scale to high-speed traffic on the Internet's data plane, nor does it provide the privacy properties needed in multi-domain setting. Second, existing work has mostly focused on maintaining network provenance, but not how to leverage it for root-cause analysis. In a large-scale distributed system, the provenance of an event of interest can be quite complex [169]. Therefore, having the provenance data is not enough, as operators still face the challenge of identifying a concise "root cause" from the complex provenance. Last but not least, provenance by itself only explains how a current network state came into existence, but it does not reason about potential ways of changing the network state to *fix* a problem while avoiding undesirable side effects caused by the change.

1.2 Thesis

In this dissertation, we argue that network provenance can be a powerful tool that enables better support for network diagnostics and forensics, and we demonstrate this by developing novel techniques and systems that address the identified challenges.

First and foremost, in order to use provenance for Internet diagnostic and forensic tasks, we need to capture *secure provenance data in high-speed networks*. This requires addressing two challenges: a) overhead, and b) privacy. Since the Internet's data plane has very high data rates, capturing secure provenance on the data plane can lead to substantial computation and storage overheads. Moreover, since there is often a need to perform diagnostics and forensics across domains, the diagnostic process also needs to preserve the privacy of users and ISPs. Therefore, we need to design a practical provenance system that can operate on high-speed data and provide the

necessary security and privacy properties.

Second, although having a practical provenance system can be helpful, the ability to capture provenance is only a starting point. In large-scale distributed systems, the provenance of an event of interest can also be quite complex: in many SDN scenarios, for instance, typical provenance trees can contain hundreds or even thousands of tuples [169]. Operators still need additional help in digesting the collected provenance when performing diagnostics. Therefore, we need to develop techniques to process the provenance data and *identify the root cause of a network problem*.

Ultimately, network operators need to find a repair to fix network problems. Provenance by itself merely explains why the current network state came into existence, but it does not offer support to reason about repairs, or changes, to the current state. This step is challenging because the repair process can have network-wide effects – it is necessary to ensure that a repair not only just fixes the symptom at hand, but also does not cause undesirable side effects elsewhere in the network. Therefore, we need to develop techniques that generate *high-quality network repairs that avoid collateral damage* to the network.

1.3 Contributions

In Chapter 2, we start by discussing related work on network diagnostics and forensics, and introduce more background on network provenance. We then address the challenges identified above and make the following contributions:

1. In Chapter 3, we present *secure packet-level provenance*, a system that can provide security and privacy properties on high-speed Internet traffic.

   Our key insight is that many identified diagnostic and forensic tasks share a common functional core, and that provenance is a good primitive for supporting this core. We design a provenance system called SPP that can maintain and query secure provenance at line rate in the presence of Byzantine adversaries.

We show that SPP can support a wide range of diagnostic and forensic tasks in the Internet. Using software and hardware prototypes, we also demonstrate that SPP has low computation, storage and bandwidth overheads.

2. In Chapter 4, we present *differential provenance*, a technique to identify the root causes of network problems from complex provenance data.

   Our key insight is that network problems are often anomalies rather than the norm, so we typically have "working" and "non-working" instances ready available – for instance, a packet that was misrouted, and a similar packet that was routed correctly. When such "reference" events exist, it is often effective to reason about the differences between the provenance of the symptom event and of the reference event to identify the root causes.

   We present an algorithm that can generate differential provenance, and a system called DiffProv that implements the algorithm. Our evaluation in the context of SDNs and Hadoop MapReduce demonstrates that DiffProv can identify problem root causes very accurately.

3. In Chapter 5, we propose *causal networks* – a generalization of provenance – that can generate network repairs while avoiding undesirable side effects.

   Our observation is that network operators often expect a successful repair to satisfy multiple constraints – for instance, the repair should make a server receive DNS traffic, and another server to receive HTTP traffic. Existing provenance systems only reason about individual events, not complex goals that involve multiple events. We add support for this capability by generalizing network provenance to causal networks, which can encode multiple constraints in a single data structure, and be used to generate network repairs that satisfy complex diagnostic goals simultaneously.

   We present an algorithm that leverages causal networks to perform network repair, and a prototype debugger called NetGenie that implements the algo-

rithm. Our evaluation shows that NetGenie can generate repairs for SDNs without causing collateral damage.

Finally, in Chapter 6, we conclude the dissertation with potential future work.

2

Background

In this chapter, we discuss related work on network diagnostics and forensics in Section 2.1, and we then provide more background on provenance in Section 2.2, as well as its applications in distributed systems in Section 2.3.

2.1 Network diagnostics and forensics

Over the years, many researchers have considered the problem of network diagnostics and forensics, and addressed a variety of challenges in this space.

2.1.1 Network diagnostics

Many debuggers have been proposed for diagnosing distributed systems. Like traditional debuggers, they can produce a form of "backtrace" to help operators understand what happened in a distributed system. For instance, `ndb` [86] and NetSight [87] are two example systems that can produce "packet histories" in SDNs. As packets traverse SDN switches, the switches can assemble "postcards" about the

packets and send them to centralized NetSight servers. The servers can then answer diagnostic queries on what happened to a particular packet. SDN traceroute [28] considers the problem of performing traceroutes without causing the network state to change upon the probe traffic. CherryPick [161] also traces packets in SDNs, and it aims to reduce the tracing overhead by only recording a small number of traversed links that can represent an end-to-end path. OFRewind [171] provides a record-and-replay technique in SDNs that can generate traces that are temporally consistent in split forwarding architectures such as OpenFlow. X-Trace [66] is a tracing framework that can tag network operations with task identifiers, and trace the requests across different layers of a distributed system.

Debugging can also be done by dynamic testing, such as in ATPG [174], BUZZ [63], DEMi [150], and MCS [151]. ATPG [174] can generate a minimal set of test packets that can exercise all rules in a network for testing purposes. BUZZ [63] can generate test cases for stateful networks with context-dependent policies, and help uncover policy violations. MCS [151] can identify a minimal sequence of inputs in SDNs that triggers a certain bug. DEMi [150] uses delta debugging to minimize faulty execution traces in distributed systems.

Statistical learning can also be an effective approach to diagnostics. NetMedic [97] uses an inference-based approach to find likely root causes based on historical behaviors of system components. PeerPressure [167] applies Bayesian estimation to configurations collected from a large number of machines in order to diagnose misconfigured machines. NetPoirot [39] uses machine learning on TCP statistics to identify responsible entities for an observed network problem. NetPrints [29] accumulates and retrieves shared knowledge of home PCs and uses decision trees to perform configuration mutations, or fixes.

2.1.2 Network forensics

Researchers have also worked on different aspects of network forensics. Source authentication and path validation systems are one class of forensic techniques. AIP [34] uses self-certifying addresses to detect and prevent spoofed network traffic. Passport [113] can verify whether a certain packet originated from a particular source domain. SPIE [157] can trace a packet's traversed path and determine the source of spoofed traffic. HAL [82] can attest to the transmission of a particular packet. ICING [135] not only authenticates the source addresses of network traffic, but also validates its traversed paths.

Performance accountability systems can help verify the SLA of service providers. Argyraki et al. [36] propose that, whenever a packet is dropped, the corresponding network component generates a feedback report (a "packet obituary") that is sent back to the source; previous hops remember each packet for a short while and add some path information when they see an obituary pass by. A later system, AudIt [37], adds the security features but drops the per-packet granularity in favor of aggregate delay and loss rates. Network Confessional [38] provides similar properties, but allows individual domains to tune the amount of statistics reporting.

Researchers have also proposed solutions to address specific kinds of network misbehaviors. For instance, Liberatore et al. [111] investigate the problem of child porn trafficking and propose to enhance the evidence of child porn possession by tagging application-level data with forensic tags, Glasnost [58] and NetPolice [178] perform end-to-end measurements to detect traffic differentiation, and Web Tripwires [145] introduces self-checks into web pages to detect in-flight modifications.

2.1.3 Verification and synthesis

Recently, there has been an active line of research that uses verification and synthesis techniques to ensure the correctness of networks before deployment, in contrast to diagnosing problems after they happen.

Verification can eliminate bugs for certain types of networks. Anteater [120], Header Space Analysis [101], NetPlumber [100], VeriFlow [102], and Libra [175] can perform static analysis on data planes and detect violation of a high-level specification. Batfish [65] can derive data planes that would emerge from a set of configurations, and check whether desirable properties hold on these data planes. ConfigChecker [31] and FlowChecker [30] convert network rules into Boolean formulas and check network invariants on them. Flowlog [137], NetKAT [35], and Kinect [104] are new domain-specific languages for programming SDNs.

Although verification can be a powerful technique, a full verification of complex networks is still hard to achieve, and most existing verification efforts are restricted to stateless networks [101, 100, 102, 35]. Diagnostics and forensics, however, is a less ambitious goal – it does not aim to prevent problems from happening, and in return, these techniques are oftentimes applicable to more complex networks, even distributed systems in general. Moreover, verification does not obsolete diagnostics, as they are orthogonal problems – if the verification process finds a violation, one still needs to (manually) identify the root cause and roll out a fix.

The goal of *synthesis* is to produce a network that satisfies a high-level specification. For instance, NetEgg [173] can synthesize SDN policies from from example scenarios, Condor [149] can synthesize network topologies that satisfy high-level requirements expressed in a Topology Description Language, [123] can synthesize network updates and their correct ordering. Like network verification, synthesis is also restricted to relatively simple types of networks, and it requires a human operator to explicitly write down her specification. Network diagnostics, on the other hand, does not aim to synthesize a network from scratch; rather, it aims to find a small change to an existing network to fix observed problems.

2.2 Provenance

Now that we have provided a high-level overview on the literature on network diagnostics and forensics, we turn to introduce more background on *provenance* in this section.

The concept of provenance was first developed in the database community [47] to describe the origin and history of data. Over time, several notions of provenance have been proposed [52], including why-provenance, where-provenance, and how-provenance, and they capture different aspects of the data lineage. Why-provenance associates a query result with a set of database tuples that it has been computed with. One of its applications is to address the "view deletion" problem – finding a set of input data items to delete, so that a result tuple in a particular view can be deleted [48]. How-provenance is a further generalization, as it also captures how the result tuple came about following the query execution steps. It has been used, for instance, in data sharing systems to assign trust to tuples according to how they have been computed [75]. Different than why-provenance and how-provenance, where-provenance does not describe how a result tuple was computed, but rather where a particular data field has been copied from. One example application of where-provenance is to propagate annotations from source data items to a particular view [44].

Over the years, provenance has become a rich problem domain in the database community. Green et al. [74] lay the theoretical foundation of provenance as a semiring algebraic structure, and Amsterdamer et al. [32] further extend the structure to semimodules to capture aggregate queries. Meliou et al. [125, 124, 126] use provenance to analyze how to make a particular query answer appear or disappear, which can help diagnose wrong database queries or problematic source data. Ives et al. [91] apply random walk algorithms to provenance graphs for ranking and recommending data items. Davidson et al. [55] consider the problem of answering provenance queries when the privacy of data items or of the workflow is a concern.

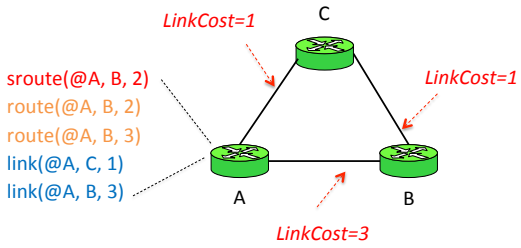


Figure 2.1: Another example network with a shortest-path routing protocol.

In fact, provenance has found its use in many other application domains beyond databases. In an operating systems context, LPM [43] maintains provenance in the Linux kernel and uses it for data loss prevention. For storage systems, PASS [132] leverages provenance to detect system changes and to perform intrusion detection, and Muniswamy-Reddy et al. [133] further extend PASS for cloud storage systems. On mobile platforms, Quire [57] tracks provenance on Android to defend against confused deputy attacks. For distributed systems, SPADE [69] is a middleware for cross-platform provenance collection, and ExSPAN [183] uses network provenance to explain why a certain network state came into existence.

2.3 Network provenance

This dissertation is particularly related to *network provenance* [183], which uses a *declarative networking* model [115] that views networks and distributed systems as databases, and network events and states as database tuples. Here, a distributed system consists of a set of nodes that are interconnected by a network, and they communicate with each other by sending and receiving messages. Each node has a set of states, or tuples, that are stored in a database. For instance, BGP routers would have a `route` relation, and an entry `route(@X,Y,C)` in the relation would mean that router `X` has a path to router `Y` with cost `C`. The symbol `@` describes the distributed nature of the system–it specifies the location where a particular relation is hosted; in the above example, the entry is hosted on router `X`. There is a set of *intensional* tuples that represent basic facts about the system; for instance, the fact that there

```
r1: route(@S,D,C) :- link(@S,D,C)
r2: route(@S,X,C) :- route(@S,X,C1), link(@X,D,C2), C=C1+C2
r3: sroute(@S,D,MIN<C>) :- route(@S,D,C)
```

Figure 2.2: The rules for shortest-path routing.

exists a link with a cost of `C` between two routers can be represented by a tuple `link(@X1,X2,C)`. There is also a set of *extensional* tuples that are derived from the intensional tuples; for instance, a one-hop route `route(@X1,X2,C)` can be derived from a direct link `link(@X1,X2,C)`. In Figure 2.1, we show an example network that is slightly more complex than our earlier example in Figure 1.1.

The nodes run a distributed protocol that specifies how tuples should be derived when and where; derived tuples can also be sent and received via the network as messages. The distributed protocol can be written as a set of declarative rules in Network Datalog (NDlog) [115], which is a Datalog variant with an extension that specifies tuple locations with the `@` symbol. NDlog rules are of the form $\texttt{q:-p}_1\texttt{,p}_2\texttt{,}\cdots\texttt{,p}_k$, which means that the head tuple q (i.e., the conclusion) should be derived whenever the body tuples $\texttt{p}_1$ through $\texttt{p}_k$ (i.e., the predicates) are all present.

For instance, consider the shortest-path routing protocol shown in Figure 2.2 with three rules. The first rule, `r1: route(@S,D,C) :- link(@S,D,C)`, describes that whenever there is a link from `S` to `D` with the cost `C`, a one-hop route from `S` to `D` with the same cost should be derived. The second rule, `r2: route(@S,D,C) :- route(@S,X,C1),link(@X,D,C2), C=C1+C2`, describes that a route from `S` to `D` with a cost of `C` can be derived by a route from `S` to an intermediate hop `X` with a cost of `C1`, together with a direct link from `X` to `D` with a cost of `C2`, where `C=C1+C2`. The final rule, `r3: sroute(@S,D,MIN<C>) :- route(@S,D,C)`, describes that the shortest distance from `S` to `D` is obtained by aggregating all routes between them and picking the minimum cost.

Under this model, a network execution can be viewed as a series of tuple inser-

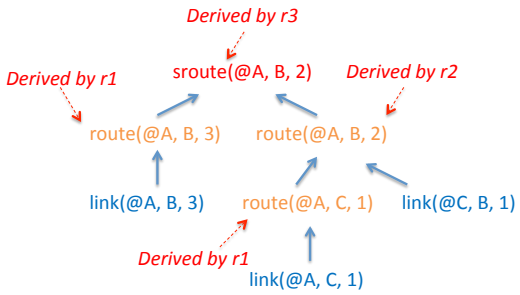


Figure 2.3: The provenance tree of the state `sroute(@A,B,2)`.

tions and deletions. An execution is always triggered by the insertion or deletion of *base tuples*, which are entries in the intensional relations, or external events, e.g., a network operator installs a new configuration, a packet arrives at a border router, etc. They will then trigger a set of NDlog rules, and generate additional *derived tuples* in the extensional relations, e.g., a new configuration results in a routing table update, a received packet gets forwarded to the next hop, etc. NDlog programs are executed using *pipelined semi-naïve* evaluation [115], where each received tuple can be immediately processed. We also assume that, after a set of updates, the system eventually stabilizes and converges to a fixedpoint.

This declarative model makes provenance very easy to see. For instance, Figure 2.3 shows the provenance tree of the routing state `sroute(@A,B,2)` on the node `A` in Figure 2.1. It can be interpreted as follows. The shortest-path routing protocol has derived a routing state `sroute(@A,B,2)` from `route(@A,B,2)` and `route(@A,B,3)` using the rule `r3`. Therefore, the provenance of `sroute(@A,B,2)` is simply the existence of the states `route(@A,B,2)` and `route(@A,B,3)`, as well as the execution of the rule `r3`. In turn, the provenance of the state `route(@A,B,2)` is the application of rule `r2` on the states `route(@A,C,1)` and `link(@C,B,1)`, where node `C` is the intermediate hop between `A` and `B`. Finally, the provenance of the states `route(@A,B,3)` and `route(@A,C,1)` is the application of `r1` on the basic states `link(@A,B,3)` and `link(@A,C,1)`, respectively.

Using the above system model, ExSPAN [183] has opened up a line of work

on *network provenance* by tackling a set of challenges in maintaining and querying provenance in a distributed setting, and showing that network provenance is practical for a range of distributed protocols, such as BGP and Chord. DTaP [182] further adds a *temporal* dimension in the provenance data, because network states – unlike traditional database tuples – tend to be short-lived. SNP [180] addresses challenges that arise due to adversaries in distributed systems that may corrupt or fabricate provenance data. Y! [169] uses counterfactual reasoning to explain why a missing event in a distributed system failed to happen. Meta provenance [168] generalizes the notion of provenance to capture both data and program code to repair SDN controller software. Chen et al. [51] propose a technique to compress distributed provenance data for storage savings. This dissertation is related to these work on network provenance, and indeed, builds on some of the above projects; but it addresses several open challenges in network provenance, as we have explained in Section 1.2.

3

Secure Packet Provenance

Diagnostics and forensics were not among the top priorities for the original design of the Internet [53], and as a result, the architecture offers relatively little direct support for them. At the interdomain level, the only features that are likely to be available are ICMP and a few IP header options, and even these are often disabled [81] or implemented inconsistently [153]. Thus, when an operator encounters a problem that is not limited to her own network (such as bad performance on a given path), there is relatively little tool support; the best option is still often to post a message to a mailing list like NANOG, or to call other operators on the phone.

Over the years, a variety of diagnostic and forensic challenges have been identified. These include diagnosing high delay, reordering, or loss [118, 110, 37], identifying the source of attack traffic [157, 113], localizing failures [41, 98, 99], detecting prefix hijacking [179], testing for traffic differentiation [178], topology mapping [153], finding the root cause of routing problems [162, 64], collecting evidence of cybercrimes [111], and verifying SLAs between ISPs [80, 119, 135], and so on. Each of these challenges involves a specific problem *deep within the network*,

which is difficult to diagnose without network-level support.

In the absence of direct support from the network, most existing work takes one of the following two approaches. The first is to approximate the missing functionality by creatively "abusing" a feature that exists for some other purpose (such as certain header options [153] or ICMP responses [41]). This is often surprisingly effective, but it typically relies on underspecified behavior and/or idiosyncrasies of certain router implementations, which can diminish data quality and require an enormous amount of ingenuity to work around (e.g., [153, 98]). The second approach is to extend the architecture with a new feature of some kind (e.g., [135, 34]), such as a new protocol, header field, etc. Such extensions provide a "clean" solution for the problem at hand, but deploying new features in the entire network is extremely difficult – so difficult, in fact, that hardly any of the proposed solutions have been widely deployed so far. To make matters worse, existing proposals typically focus on solving one particular problem and do not help with any of the other diagnostic and forensic problems that have been identified; thus, a comprehensive solution would require deploying *all* of the proposed extensions *in combination*. Given the ISPs' reluctance to make major changes to the network, this seems unrealistic.

In this chapter, we ask the following question: If ISPs were willing to deploy *only one* new primitive in the network to help with diagnostics and forensics, what should that primitive be? Our key observation is that, while the existing solutions seem very different at first glance, *they all essentially answer variants of the same question: "What were the causes and/or effects of a given past event in the network?"*. If the network could remember recent events (such as packet transmissions) and the corresponding causes and effects, even for a short amount of time, many forensic problems would be easy to solve. For instance, reverse traceroute [98] could locate the source of packet drops simply by following packets on their way from the sender to the receiver (and potentially back), and note the point at which they no longer made progress. Other forensic problems would require some post-processing: for

instance, WhyHigh [110] could find the source of high latencies by inspecting the differences between packet transmission and arrival times, and Netdiff [119] could calculate the throughput on each path segment to find the bottleneck. However, this processing could be done at the edge without further changes to the core.

There is one additional feature that some forensic systems require: the ability to *prove* the correctness of a given answer [80, 180, 113, 34, 135]. This is necessary because attackers may falsify evidence to cover their tracks [109]. Although the strength of proofs and the properties being proven sometimes vary, in essence they are all concerned with the presence or absence of particular entries in our hypothetical ledger: for instance, the PoPs in ICING [135] essentially correspond to a chain of entries that connects a packet to a particular sender, the signatures in Passport [113] and AIP [34] correspond to the beginning of this chain, and the logs in SNP [180] correspond to causal connections along the chain.

This commonality suggests that it may be possible to deploy a *single* primitive in the network, once and for all, and then re-implement the previously proposed diagnostic and forensic systems as "applications" on top of it, without further changes to the network core. In this chapter, we propose one specific candidate for such a primitive that we call *secure packet provenance* (SPP). SPP is based on the concept of data provenance from the database literature [47], which has already been used for diagnostics and forensics in other contexts, such as operating systems [89] and distributed systems [180, 183, 170]. However, as we show experimentally in Section 4.5, existing solutions would be completely overwhelmed with the high data rates at the network data plane. SPP solves this problem by avoiding cryptographic operations on the fast path and by relying mostly on ephemeral state; as a result, it outperforms the state-of-the-art secure provenance system by several orders of magnitude.

While the key insights of this chapter are architectural (that there can be a single shared primitive, and that SPP is a good candidate), we have also designed and im-

plemented a concrete protocol that could provide SPP in the Internet. Other than a small link-layer header, our protocol does not require any changes to the current data plane and can be implemented efficiently in hardware. (We demonstrate this with a NetFPGA prototype that runs at 10 Gbps.) We also used SPP to approximate six different diagnostic primitives from the literature, and we show that with SPP, each primitive can be implemented with just a few lines of code. Our main contributions are:

- Two architectural insights: that a single shared primitive can support a wide variety of diagnostic and forensic tasks, and that provenance is a good candidate for such a primitive (Section 5.1);
- the definition of a secure provenance model for the Internet's data plane (Section 3.2);
- SPP, a concrete protocol for maintaining secure provenance (Section 5.3);
- case studies showing that SPP can approximate a number of existing diagnostic systems (Section 3.4);
- software and hardware prototypes (Section 4.4); and
- an experimental evaluation, as well as proof-of-concept implementations of six diagnostic primitives on SPP (Section 4.5).

We discuss deployment strategies and their implications in Section 3.7, and we present related work in Section 4.6.

3.1 Overview

Diagnostics and forensics were not among the top priorities for the original Internet [53], as it was small in scale and experimental in nature. But today's Internet, with its broad range of applications, has attracted problems of all kinds [144, 110, 88, 95, 21, 109], which sometimes cause losses of millions of dollars [76]. But the available tools are far from adequate: packet traces, IP addresses [22], and even

thumbnail images [16] are serving as evidence; due to the lack of reliable forensics, innocent users have been falsely accused of wrongdoings [160, 16, 22]. We believe that it is time to add better support for diagnostics and forensics.

3.1.1 Goal: A single primitive

There is a rich body of work on diagnostic and forensic systems that solve specific variants of this problem, typically by extending the Internet in one way or another [110, 34, 38, 41, 58, 59, 82, 87, 113, 98, 118, 130, 135, 145, 148, 157, 159, 178]. However, the resulting variety of problem-specific, mutually incompatible extensions represents a challenge for widespread deployment. Hence, rather than trying to improve any individual one of these systems, we ask: *Is there a single primitive that could be added to the Internet to solve a wide range of diagnostic and forensic challenges?* Such a primitive would not necessarily match the efficiency of the more specialized solutions, since a shared primitive would need to provide a strict superset of the functionalities of the individual primitives; but it could certainly be more efficient than deploying all of them together. Moreover, if so many existing diagnostic and forensic systems are based on some variant of this primitive, we have good reasons to believe that it will be useful for solving future, as-yet-unknown forensic challenges as well – which is a key requirement for any possible addition to the network architecture.

In this chapter, we propose *secure packet provenance* as a candidate for such a primitive. At a high level, provenance [47] tracks how data flows through the network by recording each event, e.g., the transmission of a packet, or the installation of a new route, and its direct causes and effects. With this information, any event of interest can be explained by recursively looking up the causes of the event until a set of "root causes" (such as the transmission of a new packet at an end host, or the origination of a new route) is reached. Additionally, our proposed primitive collects cryptographic evidence of network-level events; this can be used to authenticate the

provenance even in adversarial settings.

3.1.2 Challenges

Intuitively, maintaining a secure provenance graph for the Internet would be sufficient for diagnostics and forensics, since it contains a complete and accurate description of what happened, why, when, and where. However, two key challenges need to be solved to make this approach practical.

Challenge #1: Overhead. A complete provenance graph of the entire Internet data plane would quickly consume any amount of space that could realistically be provided. We propose to solve this by keeping the full provenance only very briefly, and by offering a way to save (and later authenticate) any parts of the graph that are relevant for ongoing diagnostic and forensic tasks. To keep the computational overhead low, our proposed solution relies mostly on cryptographic primitives that can work at high speeds, such as hashing, and it applies several optimizations, such as batching.

Challenge #2: Privacy. Collecting all the provenance in a central location would be a privacy disaster. Our proposed solution avoids this by distributing the graph, and by allowing each network-level component to keep the part of the graph that pertains to itself. Also, we do not allow "global" queries of the form "show me all the packets that Bob sent" – users can only explore the provenance graph hop by hop, starting from a vertex they already know about. In effect, users can only query the provenance of packets they have already seen in their entirety. Moreover, we allow ISPs to restrict the visibility of their own subgraph; for instance, an ISP might permit its local admins to see its complete provenance, including routing policies and link statuses, but it might limit queriers from other domains to only forwarding-related information, e.g., the path that packets were sent on.

3.2 The provenance graph

We begin by defining the data model for the provenance information we wish to provide. A common way to represent the provenance is as a *provenance graph* [183] – a DAG in which each vertex represents an event and edges connect causes to effects. In this graph, the explanation of an event is simply the tree that is rooted as the corresponding vertex.

3.2.1 What is the right layer?

For a single provenance model to work for heterogeneous networks, it needs to be detailed enough to encode useful debugging information, but also general enough to abstract away hardware-specific features. We observe that this challenge resembles that of the original Internet, which needed to interconnect a variety of different network types and protocols. The answer in the original design was IP's "narrow waist", which was itself universal but permitted diversity at layers above and below. Thus, if our provenance model captures the network's operation at the IP layer, it will form a basis that different networks could agree on. As we will show in Section 3.2.4, operations on other layers can still be encoded as extensions to the IP-level provenance graph.

Network model: We model the network as a graph whose nodes are IP-capable devices. Each node has a number of ports, which can be connected to ports on other nodes using links. Nodes can transmit packets on their ports to some or all of the nodes that are connected to the corresponding (unicast or multicast) link. Therefore, this model not only includes routers and middleboxes, but also end hosts. Packets can be lost or corrupted in transmission, and nodes can mutate, duplicate, or drop any packet. Moreover, each node has a set of rules that decide how packets should be processed, and an increment-only local timer to obtain timestamps.

3.2.2 The provenance graph G

For clarity, we define the *provenance graph* $G := (V, E)$ from the perspective of a hypothetical global observer that can observe every single event in the network – i.e., every time a packet is sent or received, a link goes up or down, and a rule is inserted or deleted. G is a DAG and contains one vertex for each event, as well as a directed edge (v_1, v_2) whenever v_2 causally depends on v_1. Vertexes can have multiple incoming edges; for instance, a node might send a packet on a particular port because a) it received the packet earlier, b) it had a rule that matched the packet and specified this port, and c) the link on that port was up. Specifically, we define six vertex types, using a provenance model similar to the one from DTaP [182]:

- When a link l goes up/down on node N at time t, add a vertex LINKUP(N,t,l) / LINKDOWN(N,t,l).
- When a node N adds/removes a rule r at time t, add a vertex RULEADD(N,t,r) / RULEDEL(N,t,r).
- When a node N receives a packet p on port P at time t, insert a vertex v:=RECEIVE(N,p,P,t) to V; also, find the vertex v_1:=LINKUP(N,t,l) for the link l that is currently connected to P, and add an edge (v_1, v) to E.
- When a node N sends a packet p on port P at time t, add a vertex v:=SEND(N,p,P,t) to V. If p is sent because a packet p' was previously received by N at time t' on port P' and is forwarded to port P because of a rule r, find the vertexes v_2:=RULEADD(N,t'',r) and v_3 :=RECEIVE(N,p',P',t') in V and add edges (v_2, v) and (v_3, v) to E.

G is, in effect, a complete chronicle of everything that happened in the network: in principle, it is possible to "replay" the entire execution of the network in simulation. Thus, if a question can be answered in this very detailed simulation, it must also be possible to answer it using the information in G. In particular, to explain why an event e has occurred, we can simply find the corresponding vertex v in G and look

at the subtree that is rooted at it, the leaves of which are the "root causes" that, in conjunction, have caused e to occur. Later, we will describe a *distributed* algorithm that maintains a close approximation of G without assuming a central entity. This is based on the observation that each vertex $v \in G$ has a natural "home": the node N that appears as the first entry will store the vertex v.

3.2.3 Querying and evidence

We allow users to examine the graph G with a query primitive QUERY(v) that returns v's adjacent vertexes in G. Thus, users could start with a vertex they know (say, the transmission or arrival of a packet at their local node) and explore the graph by invoking QUERY recursively. However, recall that G is distributed, and that each node stores the vertexes that pertain to it. So a malfunctioning or compromised node could fabricate or destroy vertexes that it stores locally. To prevent this, nodes are required to store not only the vertexes themselves, but also *evidence* to prove that the adjacent vertexes exist. The evidence e_v of a vertex v can be thought of as a statement that is signed by the "home" node of v saying that v is a part of G. Conceptually, nodes exchange evidence whenever they add an edge to G between two of their vertexes. Thus, each node can use the evidence to prove to any third-party that the other end of the edge must exist in G.

Hence, we augment the query primitive with evidence. QUERY(v,e_v) returns two sets of vertexes: all the predecessors and successors of v in G. Each returned vertex v' is accompanied with evidence that 1) v' is in V, and that 2) the relevant edge ((v',v) for predecessors and (v,v') for successors) is in E. As before, users can use QUERY to explore a larger portion of G by invoking QUERY recursively, starting from some vertex they know and have evidence for.

3.2.4 Extensions

The above data model only captures IP-level provenance. But as we discussed in Section 3.2.1, operations above and below the IP layer can be encoded as extensions to this basic provenance graph to support richer diagnostic and forensic capabilities. Below, we briefly sketch two examples.

Control-plane diagnostics: In the basic provenance model from Section 3.2.2, changes to link statuses and rules are "root causes" that cannot be explained further. However, it would be easy to add more entries to the TELs to further explain the provenance of these events. For instance, NetReview [80] already records a type of secure provenance for the BGP control plane; this provenance could be integrated with the IP-level model to further explain the RULE vertexes. Provenance tools that do not understand the new vertex types in the TEL could simply ignore them and continue to treat the RULE vertexes as basic events in the provenance graph.

Summarizations: To enable longer-term forensic queries, it could be useful to have a less detailed but smaller version of the IP-level provenance graph (say, a flow-level version); thus, the detailed version could be discarded after a few seconds, while the aggregated version could remain available for hours, or even days. Our basic model can accommodate such extensions by having routers commit to the basic graph and its summarizations simultaneously. As long as both endpoints of a link generate the summarizations in the same way (e.g., by using the same sampling technique), they can verify correctness exactly as in the basic IP-level version.

Visualization: To help operators to better understand the diagnostic results, the evidence can be displayed using provenance visualizers such as NetTrails [181].

3.2.5 Does G reveal too much information?

End users might be concerned that QUERY could be used to spy on their traffic. But our design prevents this: to query a vertex in G, the querier must already have

evidence for an adjacent vertex. So, in order to access the provenance of a packet p that was sent from A to B, the querier must have some evidence of p's existence – which is only available at the sender A, the recipient B, and the ASes along the path, all of whom have already seen p in its entirety. Thus, there are only two cases: 1) the querier already knows that p exists, and what exactly it contains; in this case, QUERY will reveal where p came from, where it went, and what exactly happened along the way. Or 2) the querier does not yet know that p exists; in this case, the querier learns nothing from QUERY because the invocation will fail.

ISPs could have similar concerns about the topology and the configuration of their own infrastructure. But the Internet's topology can already be learned in great detail today [153], so G does not reveal much additional information – it merely reduces the effort that is needed to obtain it. Moreover, networks can protect policy-related information by hiding certain vertexes: each node can implement its own policy to decide which vertexes should be hidden. For instance, a network may want to reveal RULE and LINK vertexes only to its own admins, and hide them from users in other domains. Thus, each querier is presented with a *view* of the provenance graph, and can explore only the parts that are visible to her. To preserve usability, our provenance model prescribes that the SEND and RECEIVE vertexes be included in any view. Therefore, queries with a restricted view, e.g., inter-domain queries, can only trace packet paths from the returned SEND and RECEIVE vertexes; queries with an admin's view, e.g., intra-domain queries, can additionally learn why, i.e., from the RULE and LINK vertexes.

3.3 The SPP protocol

We now describe a distributed algorithm called SPP that implements the proposed provenance graph.

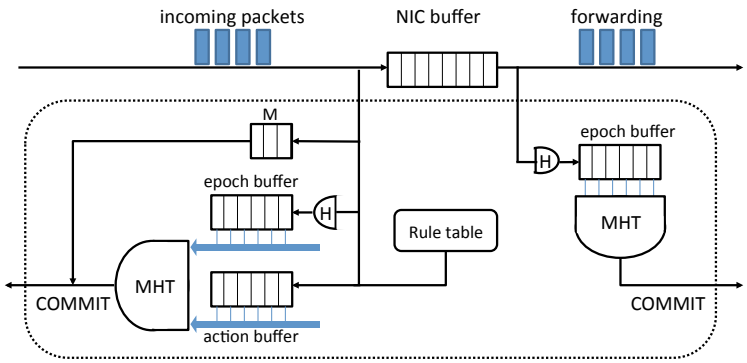


Figure 3.1: Data flow in the commitment protocol.

3.3.1 Assumptions and threat model

We design SPP based on the following assumptions:

- There is a hash function $H(\cdot)$ that is pre-image resistant and collision resistant.
- Each node i has a key pair π_i/σ_i that can be used to sign messages. A node i's signatures cannot be forged without knowing i's private key σ_i.
- If a link $i \to j$ exists, then j has a back channel for sending a small number of messages back to i.

The first assumption could be satisfied, e.g., by SHA-256. The second assumption could be satisfied with a small extension to the RPKI. The third assumption holds trivially for all bidirectional links; for other links, it could be satisfied by using a different link for the back channel.

Threat model: We assume that nodes can fail or be compromised by a *Byzantine* attacker, i.e., we make no special assumptions about the affected nodes, other than that they cannot break cryptographic keys. In particular, these nodes can drop, alter, or fabricate packets, they can destroy or tamper with any local state, and/or collude with each other.

3.3.2 Commitment protocol

The purpose of the commitment protocol (illustrated in Figure 3.1) is to generate evidence for the provenance graph. The protocol runs between the two endpoints A and B of each link $A \rightarrow B$; bidirectional links run two separate instances of the protocol, and nodes with multiple ports run separate instances for each local port. By this protocol, A would be able to prove that the packets it has sent have been received by B, and B would be able to prove that the packets it has received were indeed sent by A. This is done as follows.

Sender: A uses its local timer to divide time into *epochs* of some fixed length, e.g., 100ms, and both A and B maintain a small number of *epoch buffers* in which they record information about the packets they have sent or received. A begins a new epoch E_i by sending a message NEWEPOCH(i) to B. After that, whenever A sends a packet p to B, A appends the hash $H(p)$ to the buffer and then prepends an index j of packet in the epoch buffer as a small extra header before p. A ends E_i by sending message ENDEPOCH(i,n) to B, where n is the total number of packets it has sent to B in this epoch.

Receiver: B has meanwhile forwarded each packet as usual, but it has also recorded in its own epoch buffer the hashes of all the packets it has received correctly – i.e., without link-layer errors or CRC mismatches – from A; moreover, B has identified any missing index numbers (by looking for gaps in the sequence of numbers) and has recorded these in a small separate buffer M, so it can later report them to A. When A's ENDEPOCH message arrives, B computes a Merkle Hash Tree [127] (MHT) over the hashes in the epoch buffer, extracts the top-level hash h_0, writes an entry $s_B :=$ EPOCHLOCAL$(A \rightarrow B, i, h_0)$ to its tamper-evident log (see Section 3.3.3), and returns a message COMMIT(i, a_B, CHAIN(a_B, s_B), M) back to A. a_B is an *authenticator* (defined in Section 3.3.3), and CHAIN(a_B, s_B) is a hash chain that connects a_B to s_B. (This is used to enable audits later on.) By sending this message, B commits to having received the packets in its epoch buffer.

Agreement: While B is working on its commitments, A continues to forward packets, and it records the corresponding hashes in other epoch buffers to avoid being stalled. However, once the COMMIT message arrives, A locates the corresponding buffer, removes the packets with sequence numbers in M, and then computes a MHT over it in the same way as B, which should yield the same top-level hash h_0. A then records an entry $s_A := \text{EPOCHREMOTE}(A \to B, i, h_0, a_B, \text{CHAIN}(a_B, s_B))$ in its local tamper-evident log and returns a $\text{CONFIRM}(i, a_A, \text{CHAIN}(a_A, s_A))$ message to B, which records a $\text{FINAL}(i, a_A, \text{CHAIN}(a_A, s_A))$ entry in its log. At this point, A and B have agreed on the set of packets that have been sent over the link in this epoch, and they both hold evidence of this fact (the authenticators and hash chains) in their respective logs. Note that this does not attempt to make packet transmissions reliable, but merely enables the endpoints to agree on the set of correctly transmitted packets.

Actions: Nodes not only have to remember each packet they received, but also what happened to it, so that it can be tracked down a path. SPP represents this information as 1) a time offset to the beginning of the epoch to indicate the time when the packet was received; 2) a set of links to which the packet was forwarded (i.e., to capture both unicast and multicast protocols), if any; and 3) for each such link, a list of rule identifiers and mutations that were applied. Such information is collected in *action buffers* that are "parallel" to the epoch buffers. The nodes commit to their actions by building an MHT over the action buffer, just as it does for the epoch buffers; and the top-level hash h_0^a is recorded in an entry $\text{ACTION}(A \to B, i, h_0^a)$ in its tamper-evident log, just after the EPOCHLOCAL entry.

SPP uses the action buffers to produce the RULE vertexes that link a $\text{RECEIVE}(p)$ vertex to its $\text{SEND}(p_i')$ vertex. This is crucial because some nodes can apply mutations to packets in transit: for instance, a NAT will change the port numbers and IP addresses in the header, and many routers will decrement the TTL field. In these cases, the hash $H(p_i')$ of the forwarded packet will differ from the hash $H(p)$ of the packet that was received. But given the recorded actions, an auditor whose view

includes the RULE vertexes can reapply them to p and verify whether $H(p'_i)$ is the correct hash.

Epoch faults: If any of the required messages does not arrive, or if A and B compute different top-level hashes, they report this as an *epoch fault* to their local administrator, e.g., by incrementing an SNMP counter. Absent link failures and attacks, epoch faults can only occur due to undetected packet corruption that has not been handled at the MAC layer, or due to loss of control packets (which could be avoided with extra FEC on these packets, or by sending control packets multiple times.) Therefore, a non-trivial number of epoch faults suggests either a link failure or an attack, and should be investigated immediately by an administrator.

3.3.3 Tamper-evident log

To prevent nodes from "changing history" and from presenting different views of their history to different auditors, each node maintains an append-only *tamper-evident log* (TEL) [84]. The TEL consists of entries of the form $s_i := (t, h_i, \mathscr{E}, c)$, where t is a timestamp, $\mathscr{E}$ is an entry type, c is the content of the entry, and $h_i := H(h_{i-1} || t || \mathscr{E} || H(c))$ forms a hash chain of the entries. SPP has nine entry types:

- RULEADD(r,R) / RULEDEL(r): A rule R with rule ID r was added or deleted;
- LINKUP(l) / LINKDOWN(l): Link l went up or down;
- EPOCHLOCAL(l,E,h_0) / EPOCHREMOTE(l,E,h_0,a,c): The top-level hash of the local/remote MHT for an epoch E on link l was h_0. a and c are the remote node's authenticator and hash chain.
- FINAL(l,E,a,c): The final authenticator and hash chain for epoch E on link l were a and c, respectively.
- ACTION(l,E,h_0^a): The top-level hash of the action buffer for an epoch E on link l was h_0^a.
- CHECKPOINT(C): C contains a snapshot of the node's current link statuses and rules.

The TEL can be used to authenticate past entries as follows. Recall from Section 3.3.2, any node A can commit to the contents of its TEL up to some entry s_k by sending an *authenticator* $a_k := (k, h_k, \sigma_A(k||h_k))$ to another node. If A ever tampers with a previously recorded entry s_j, $j \leq k$, this change will invalidate the hash values of all subsequent entries and be inconsistent with a_k, as well as any other authenticators that the node has sent since s_j. Therefore, suppose that A wants to prove to B that an entry s_j was part of the log that was authenticated by a_k, $k \geq j$. Then A can provide a hash chain $\text{CHAIN}(a_k, s_j)$ that consists of $(t_x, \mathscr{E}_x, H(s_x)), j < x \leq k$; using this information, B can recompute $h_j, h_{j+1}, \ldots, h_k$; if h_k matches the value in a_k and a_k is properly signed with A's secret key, B can be sure that the claim is valid [84].

The TEL has two other uses. First, it can be used to reconstruct previous states, e.g., a rule that was used in some past epoch, by loading the most recent checkpoint before that epoch and replaying all the subsequent actions until the epoch of interest has been reached. Second, SPP does not require synchronized clocks across the network; the EPOCHLOCAL and EPOCHREMOTE entries in the TEL provide a form of timeline entanglement [121], which limits how much a compromised node can distort the timing of events to the length of a single epoch.

Notice that the data in the TEL is needed to respond to queries; if a node's TEL is lost or corrupted, that node will no longer be able to respond and thus will (appropriately) register as faulty. However, the loss of the TEL will also prevent a more detailed diagnosis. If this is undesirable, the system can maintain replicas of the TEL.

3.3.4 Query processing protocol

We now describe how to query the evidence in the TELs.

Querier: A can query the fate of some packet p it has previously sent to B as follows. A scans its epoch buffers for the hash $H(p)$ and identifies 1) the epoch i in which p was sent, 2) its index j, and 3) the commitment $c := \sigma_B(A \rightarrow B||i||h_0)$ with which B

System	Goal	Information offered	Capabilities				
			Secure evidence	Supports forensics	Covers entire Internet	Fine-grained entities	Fine-grained traces
Tulip [118]	Fault localization	Loss, delay, reordering	×	×	✓	✓(Routers)	✓(Packets)
NetPolice [178]	Traffic differentiation detection	Loss	×	×	✓	× (ISPs)	× (Flows)
SPIE [157]	IP traceback	Backward routes	×	✓	✓	✓(Routers)	✓(Packets)
NetSight [87]	Network debugging	Packet histories	×	✓	×	✓(Routers)	✓(Packets)
Netdiff [119]	ISP performance benchmarking	Delay	×	×	✓	× (ISPs)	✓(Packets)
Paris-traceroute [41]	Load-balancer detection	Load-balanced routes	×	×	✓	✓(Routers)	✓(Packets)
HAL [82]	Packet attestation	Packet transmissions	✓	✓	×	✓(Links)	✓(Packets)
AudIt [37]	Performance accountability	Loss, delay	×	×	✓	× (ISPs)	✓(Both)
SPP	Single network-level primitive	All of the above	✓	✓	✓	✓(Routers)	✓(Packets)

Table 3.1: Comparison between SPP and some existing diagnostic and forensic primitives.

has acknowledged p's receipt. A then constructs a containment proof $\text{PROOF}(c, H(p))$, which shows that $H(p)$ is a leaf node in the MHT rooted at h_0, and invokes $\text{QUERY}(p)$ on B with the tuple $u := (c, \text{PROOF}(c, H(p))$.

Responder: When B receives the query, it first verifies that the provided commitment is genuine. If so, it uses the epoch number and the link identifier in c to locate the corresponding action buffer, which will tell B the rule that it has applied to p, and which link(s) p was forwarded to. Finally, B provides the following response: 1) for each link $B \to C$ to which p was forwarded, the commitment $c_C := \sigma_C(B \to C \,||\, j \,||\, h_0^{C,j})$; 2) the new hash $H(p')$; 3) a containment proof $\text{PROOF}(c_C, H(p'))$; 4) the relevant entry s_p in the action buffer, and 5) a containment proof $\text{PROOF}(c_C, H(s_p))$. 1)–3) give A all it needs to invoke QUERY on the next hop C (or, if the packet was cloned, on each next hop), and to generate the SEND and RECEIVE vertexes for the next-hop link(s); 4) and 5) allow A to apply the mutations in s_p and verify that p' is the same packet as p.

3.3.5 Retroactive freezing protocol

So far, we have explained SPP as if each node kept all of its epoch buffers forever. In practice, SPP allows each node to expire old epoch buffers after some time T_E, while ensuring that malicious nodes cannot discard their buffers freely to cover their tracks, and that normal queries are given enough time to complete.

SPP uses a retroactive freezing protocol, where a node A can request that the

evidence for a packet p be frozen into stable storage, so that it can be inspected on human timescales. A does so by sending a special *freeze packet* $p' = \text{FREEZE}(H(p))$ on the same port as p before T_E elapses. p' and p maintain the same header so that they will likely take the same path. But if path divergence happens or p' gets dropped, SPP can be recursively applied to p' to investigate such instances. Moreover, the freeze packet is sent *retroactively*, up to several seconds after the packet was originally sent, so that a compromised node cannot predict which packets will be frozen, and then treat these packets differently to avoid detection; by the time the freeze packet arrive, the nodes will have forwarded the packets and committed to their actions. The end users can choose which packets to freeze according to their needs, or randomly freeze a subset of their traffic. To prevent the freeze primitive from being abused (e.g., for DoS attacks), nodes can limit the rate at which they are willing to freeze packets: if a node receives too many freeze requests from a neighbor, it can record the requests and the corresponding commitment, and then deny the excess request. If that node is challenged later because it did not respond to a request, it can show the saved requests to prove its innocence.

3.3.6 Properties

Next, we discuss the properties of SPP. In the presence of Byzantine nodes, the provenance graph G_e constructed from the collected evidence e is only an approximation of the "actual" provenance graph G; for example, a faulty node may refuse to provide an explanation consistent to e in response to a QUERY request. However, G_e is a close enough approximation of G, providing the following guarantees:

- G_e is **accurate**. G_e faithfully reproduces all the vertexes on correct nodes, that is, 1) if a vertex v on a correct node exists in G_e, then v must also exist in G, with the same predecessors and successors; and 2) a correct node will never be accused as faulty.
- G_e is **complete**. Given evidence e from correct nodes, 1) each vertex in G on

a correct node also appears in G_e, and 2) when some node is detectably faulty, recursive QUERY invocations will identify at least one faulty node.

In other words, although we cannot force faulty nodes to cooperate, SPP will always generate provenance that reflects the actual execution of all correct nodes, and SPP can correctly expose at least one faulty node with non-repudiable evidence.

In terms of privacy guarantees, a node is not allowed to audit or otherwise learn about packets it has not processed:

- G_e is **private**. Given an evidence e collected through recursive QUERY, G_e constructed by node v contains only SEND and RECEIVE vertexes for packets that are *visible* to v. We say a packet p is *visible* to a node v, if 1) p is received or sent by v, 2) p is mutated and forwarded as p' that is visible to v, or 3) a visible packet p' is mutated and forwarded as p.

3.3.7 Limitations

SPP is designed for diagnostics and forensics on the Internet's data plane, and there are at least three classes of problems that SPP cannot diagnose directly: a) faults of a remote node that do not affect any external messages, such as CPU overload; b) faults that happen outside of the Internet data plane, such as BGP prefix hijacking; and c) faults that need aggregate information about the packets, such as high per-flow latencies. Next, we explain these categories in more detail, and discuss potential ways of addressing some of them.

Non-observable faults: Not all problems on a node can be detected from only its externally visible inputs and outputs. For instance, if a bit flips in a node's memory, its CPU load is high, or its disk has failed, the network packets that the nodes sends may not be affected initially (or ever). Even if the problem does affect a network packet, detection may still be impossible if the nodes that receives the packets is also faulty. This limitation is inherent [83] and also affects other systems that attempt to detect or diagnose faults based on network events.

Control-plane diagnostics: SPP, as described here, generates an IP-level provenance graph on the Internet's *data plane*; it does not provide visibility into control-plane events. Thus, SPP's QUERY primitive cannot detect faults that manifest entirely on the control plane, such as BGP prefix hijacking, routing policy violations, and the like. This limitation is not inherent, and it should be possible to remove it by extending the provenance model to capture control-plane events, as discussed in Section 3.2.4.

Aggregate information: SPP's QUERY primitive returns information about individual packets, so it cannot directly diagnose problems that are related to aggregate properties of multiple packets or entire flows. For instance, if a flow is experiencing low throughput, this cannot be detected based on what happened to individual packets in that flow. One way to get around this would be to implement the summarization extension from Section 3.2.4; an even simpler way would be to query multiple packets and to do the analysis as a post-processing step. The precision of the second approach would be limited by the accuracy of the nodes' timestamps (recall that SPP does not assume synchronized clocks); however, previous work has shown [38] that useful performance measurements are possible even when the clocks are only loosely synchronized.

3.4 Case studies

Next, we describe four classes of common diagnostic and forensic tasks for which specialized solutions already exist. We explain how SPP can approximate these solutions, and how they could be re-implemented on top of SPP. Table 3.1 provides a summary.

Traceback: Traceback is the process of identifying the sender of a (potentially spoofed) packet. This is difficult in the current Internet because packets contain no secure data about their source or the paths they have traversed. Source authentication systems like AIP [34] and Passport [113] aim to prevent spoofing using cryptographic signatures. Path verification systems aim to reconstruct a packet's

path, e.g., by securely recording it in a header [135], by probabilistically marking packets [148], or by keeping digests of packets at each router [157]. Both types of traceback essentially require access to the *path a received packet has taken*, which is a part of the packet-level provenance that SPP offers (although SPP does not proactively prevent spoofing).

Routing and performance problems: A common diagnostic task is to determine why a particular path has unusually high packet loss, delay, or has become unavailable [110, 41, 159]. To overcome the limited visibility deep within the network, proposals have been made to extract more diagnostic information, e.g., by using more vantage points [98], adding network extensions [118, 38, 143], or using historical data [59, 157]. NetSight [87] even remembers packet "histories" that are similar to the provenance in SPP (though in an intra-domain setting). In essence, those systems want to know the *path a transmitted packet has taken*, along with some timing information for each hop. SPP exposes a superset of the information needed: packet-level properties are visible directly; flow-level properties can be extracted by some post-processing on a set of frozen packets.

Intrusions and misbehavior: Internet-related evidence is appearing in many court cases, but forged packets and IP addresses can lead to judicial errors [82] and bogus actions [142]. One possible solution is to enable the use of packet traces as secure *evidence* using source or packet authentication. AIP [34] and Passport [113] provide the former, and HAL [82] provides the latter; ICING [135], Clue [27], and DRKey [105] support both. In some cases, the ISPs themselves have an incentive to manipulate unwanted traffic [58] or to inject advertisements [145]. Systems to detect such misbehaviors include, e.g., Glasnost [58] and NetPolice [178] that detect traffic differentiation, and Web Tripwires [145] that detects in-flight packet modifications. However, to ultimately resolve such situations, one also needs *evidence*: since the recipient of the packet (the victim) is usually different from the entity that takes action (e.g., a judge), it is necessary to verify that a particular evidence is au-

thentic. SPP's authenticators are designed for this purpose.

Topology discovery: Topology mapping is useful for latency prediction and modeling [153]. However, in the absence of direct support, people must generally rely on low-quality data, e.g., from traceroutes or IP record-route options, which require great ingenuity to collect and clean up. It would be much easier if the network provided *explicit and unambiguous information*, so that there would be no need for "guesswork" based on subtle idiosyncrasies of network hardware.

3.5 Implementation

We have implemented two prototypes of SPP: a software-only implementation of the entire system, and a NetFPGA prototype of the parts that would need to run at line speed.

Software prototype: Our software prototype is written in C/C++. It can be configured to run 1) as a Click router module [107], or 2) as a standalone program. In Click mode, SPP runs with live traffic forwarding; we performed functionality checks and prototyped six common diagnostic routines in this mode. In standalone mode, SPP still runs the entire protocol but disables traffic forwarding; we used this mode to evaluate SPP's protocol overhead as a very conservative lower-bound. Both modes are trace-based, so they are not limited by the speed of our physical NICs. We used SHA-1 for the hash function[1] and RSA-2048 for the cryptographic signatures, as implemented in the OpenSSL library v1.0.1.f. Our Click mode implementation is based on Click v2.1.

NetFPGA prototype: As we will show in Section 4.5, the dominating cost in SPP comes from packet hashing and MHT construction. To evaluate its performance in realistic deployment, we have built an additional implementation of those two components in hardware, on a NetFPGA-10G [46] platform. Our platform contains

[1] After the recent discovery of a collision [158], SHA-1 is no longer considered secure. Future implementations should use a more recent hash function, such as SHA-256.

a Xilinx Virtex 5 (65nm) FPGA (xc5vtx240tffg1759-2 [90]), as well as four SFP+ modules that can each support 10 Gbps traffic. We have implemented SPP as part of the Output Port Lookup module (somewhat analogous to the design in [134]), so that it can run in parallel with the traffic forwarding path. Our logic is divided into 13 fully pipelined stages. The first stage contains a state machine that parses packets from NetFPGA's AXI4-Stream interconnects; the second stage computes per-packet hashes; and the remaining 11 stages construct the MHTs. Our implementation builds on NetFPGA and open-source hashing libraries, and consists of 2,588 lines of Verilog code.

The first stage routes 64 bits of packet data per cycle from the AXI4-Stream interconnects, so it can send a minimum-sized packet to the hashing stage every eight cycles. Our hasher also accepts 64 bits per cycle, but it incurs a 14-cycle delay after the packet's last-bit signal is asserted. To nevertheless keep up with the incoming data rate, the hashing stage contains four separate instances of the hasher and uses them in a round-robin fashion. Each of the MHT stages consists of a buffering phase and a hashing phase: the buffering phase uses a fallthrough FIFO in SRAM to hold the hashes produced by the previous stage, and the hashing phase dequeues hashes from its FIFO, hashes them in pairs, and then enqueues the new hash at the next FIFO. The last stage's hasher produces the MHT roots. Since the data rate decreases as hashes pass through the MHT stages, we are able to do rate matching using 15 hashers: four for the first MHT stage, two for the second MHT stage, and one per each of the remaining stages. We have used SHA-3 (Keccak) in the hardware implementation for its good performance; to make the results comparable to those from the software prototype, we use only the last 160 bits to match the length of SHA-1.

3.6 Evaluation

In this section, we evaluate SPP's performance overhead and demonstrate how common diagnostic functionalities can easily be built with it. We first evaluate SPP's

protocol overhead with our software prototype, including storage, bandwidth, and computation costs, both with *real high-speed traffic*, and in *worst-case scenarios*; in addition, we report our hardware microbenchmarks to show that the seemingly high computation cost in software could be easily handled by off-the-shelf hardware technology. (Note that the storage and bandwidth overheads, unlike the computation cost, would not differ across hardware and software platforms.)

We obtained our real traffic from CAIDA's live capture on a 10 Gbps OC-192 link on Jan. 19, 2012, in which 4.6 million packets were sent with an average rate of 2.46 Gbps. For the worst-case scenarios, we synthesized traffic at 100 Mbps, 1 Gbps, and 10 Gbps in which all packets have the minimum size, and thus the traffic has the maximum packet rate (which is unlikely to occur with real traffic). We also used an epoch length of $T = 100$ms, and 10-bit sequence numbers in the link-layer headers, allowing the numbers in the header to wrap: the full sequence number can be reconstructed as long as loss bursts are below 2^{10}. Our software experiments were run on a Dell OptiPlex 9020 workstation, which has a 3.40 GHz Intel i7-4770 CPU (with 8 cores), 16 GB of RAM, and a 500 GB hard disk. The OS was Ubuntu 14.04 with kernel version 3.8.0.

3.6.1 Recording: Computation cost

SPP requires each network component to regularly generate commitments for the traffic it sends and receives, and to verify its neighbors' commitments. We first used our software prototype to quantify this cost. We generated synthetic traces that consisted entirely of 40-byte packets (the smallest valid TCP packet, 84 bytes on the wire [96], and thus the worst case for SPP) with rates of 100 Mbps, 1 Gbps, and 10 Gbps. We then ran all four traces through our software prototype, measured the computation time to generate and verify the commitments, and normalized the cost to the performance of an individual core. For instance, if one core took 2 seconds to process the commitments for packets sent in 1 second, we report this as 2 cores.

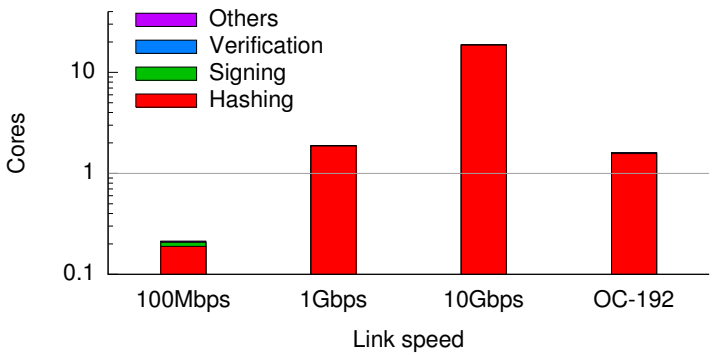


Figure 3.2: Computation cost of SPP's commitment protocol, normalized to the power of one core. The cost of hashing dominates. (The other bars are too low to see.)

(SPP trivially scales to multiple cores, as the cores can work on different epochs independently.) We report a decomposition of the cost of hashing, signature generation, and signature verification.

Figure 3.2 shows that the dominant computational cost of SPP is hashing, especially at higher link speeds. This is good news because hashing is easy to do in hardware [68, 136], and it is also the reason why our NetFPGA prototype focuses on hashing: the remaining computations have a moderate cost, so routers should be able to perform them in software. Figure 3.3 shows results from a similar experiment where the two highest-cost traces still maintain the same rates, but have different packet sizes (and 14-byte Ethernet headers). The figure shows that the overhead drops quickly as the packet size increases. This is because the number of internal hashes in the MHT depends only on the number of packets, but not on their size. At a more typical packet size of 300 bytes [156], the cost is 54% lower.

Hardware prototype: For our NetFPGA prototype, computation cost is not a good metric; instead, we quantify the maximum supported bitrate and the number of hardware elements that it requires. Our NetFPGA prototype can be synthesized to run at 200 MHz (5 ns per clock cycle), which achieves a theoretical throughput of 12.8 Gbps, and an effective throughput of 10 Gbps with the existing SFP+ modules. We note that these results are consistent with other benchmarking ef-

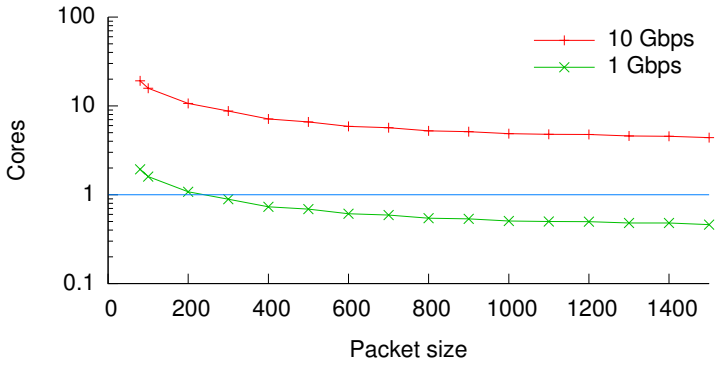


Figure 3.3: Computation cost for different packet sizes in the two traces with the highest costs.

Resource	Used	Total available	Utilization
Slice registers	53,964	149,760	36%
Slice LUTs	109,040	149,760	72%
Block RAMs	28	324	8%

Table 3.2: Hardware cost for hashing and building MHTs.

forts [67, 122, 42]. We have listed the hardware utilization in Table 3.2, in terms of LUTs (LookUp Tables), registers, and Block RAMs used. They are well within the resources available on Virtex-5 FPGAs, and would be only a fraction if mapped on more recent FPGAs: for instance, NetFPGA-SUME's Virtex-7 has nearly three times as many logical elements [185]. We also note that 10 Gbps is not the limit: for 100 Gbps routers, there are optimized hashers that could achieve 34.27 Gbps per hasher on Virtex-5 FPGAs [128], which is about ten times faster than the hash module we have used. The performance of our hardware prototype represents a lower bound on the performance that a hardware implementation of SPP can achieve; in a real-world deployment, the packet processing would be performed on ASICs in high-speed routers, which are much faster than FPGAs.

3.6.2 Recording: Bandwidth cost

Since SPP's bandwidth and storage overheads do not vary with the underlying hardware or software platforms - unlike the computation speed - we evaluated them on our software prototype. SPP requires an extra link-layer header, as well as some new control messages for exchanging commitments. Both consume some fraction of the raw link capacity that is no longer available for sending traffic. To quantify this effect, we measured the fraction of the raw link capacity that was used by SPP. We sent $R = 3$ replicas of each control message, to conservatively account for message loss, and we assumed a link-level packet loss rate of 1%, which is orders of magnitude above typical rates today [20, 14]. We show results for 40-byte packets (the worst case) and a more typical packet size of 300 bytes.

Figure 3.4 shows our results. For the 100 Mbps trace with 40-byte packets, SPP only consumes about 2.06% of the available link capacity. Moreover, the overhead drops with increasing link speeds and increasing packet size. This is because the overhead has two components: one consists of three fixed-size messages (NEWEPOCH, ENDEPOCH, and CONFIRM) that are sent once per epoch, regardless of the link speed and the number of messages, and the other consists of the link-layer headers and the entries in the missing packet list (COMMIT), which are both proportional to the number of packets. At 1 Gbps and with the more typical 300-byte packets, the overhead is only 0.42%. For the most realistic case of the OC-192 link, the overhead is only 0.16%.

3.6.3 Recording: Memory

SPP requires a certain amount of RAM for epoch buffers, action buffers, and the list of lost packets. Next, we quantify how much memory these data structures require.

In our implementation, an entry in the epoch buffer requires 20 bytes (the size of a hash value), an entry in the action buffer requires 30 bytes (the size of a timestamp, a destination port number, and up to 3 mutation records), and an entry in the loss

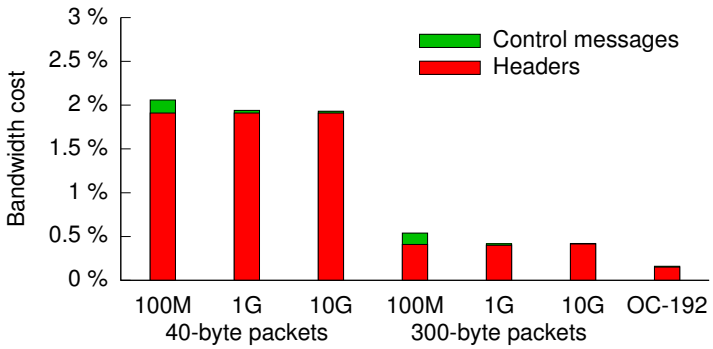


Figure 3.4: Bandwidth consumption of SPP's commitment protocol, as a fraction of the raw link capacity.

buffer requires 10 bits (the size of a sequence number). For each received packet, SPP adds one entry to each of the first two buffers, and for each missing sequence number, it adds an entry to the third buffer. We also dimension the buffers for the worst case. If we assume a 1 Gbps link, an epoch length of $T = 100$ms, and a minimum packet size of 40 bytes (i.e., up to 312,500 packets per epoch), the epoch and action buffers would require 6.25MB and 9.38MB of memory, respectively; with a link-level loss rate of 1%, the loss buffer would require 3.91 kB. Since all sizes are proportional to the link speed, a 10 Gbps link would require ten times as much.

The number of buffers depends on the number of ports the node has, and on the latency that is needed to finish the commitment protocol, which depends on the link's RTT. (Recall that the sender must retain the hashes until the receivers' COMMIT message arrives.) If we conservatively assume a per-link RTT of up to 100ms, $2 * (100/T) = 2$ buffers would be needed per port, so a node with twenty 1 Gbps ports would need 625 MB of RAM. Note that the hashes are written at much lower rates than the links' bitrates, so expensive SRAM is *not* required – commodity DRAM is enough, e.g., NetFPGA-SUME has 8 GB of DDR3 synchronous DRAM with a 238.8 Gbps peak memory throughput [185].

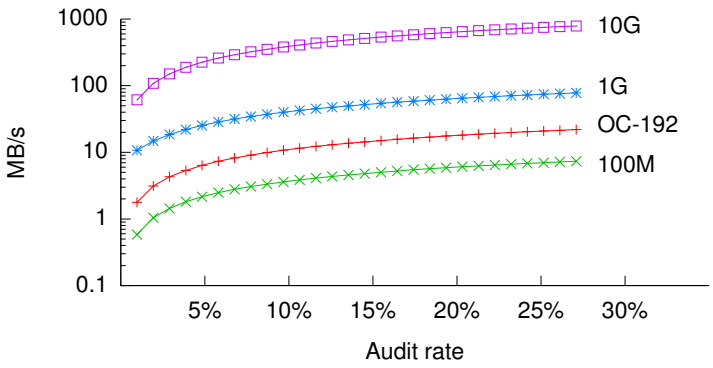


Figure 3.5: Data rates for different auditing rates ϕ.

3.6.4 Recording: Disk space

SPP requires disk space to store the packet-level evidence that has been "frozen" by queriers, as described in Section 3.3.5. To quantify how much storage is needed, we ran the traces through SPP and randomly froze a certain fraction ϕ of the packets. Figure 3.5 shows the amount of data written to disk due to audits. It is expected that the amount of frozen evidence increases with ϕ. However, the increase is not linear: for small ϕ, SPP must store not only the hash of each frozen packet but also the hashes of internal nodes along the path to the root. But, as ϕ increases, there is more and more overlap between the paths, which reduces the number of additional hashes that need to be stored for each new frozen packet. Note that the next-hop authenticator needs to be stored only once per epoch, so the necessary space is comparatively small. From the figure we can see that, an auditing rate of $\phi = 1\%$ can be well supported by the throughput of a hard disk, and $\phi = 15\%$ with a commodity SSD.

Summarizations: Summarizations (Section 3.2.4) can further reduce the storage consumption. To demonstrate this, we have designed a flow-level summarization that contains less detail but can be retained longer. Analogous to NetFlow, this summarization describes the flows the router has seen, the number of packets in each flow, and the size of each flow (but *not* per-packet information, such as the content

hashes). We note that naïvely extracting flow-level information by examining every packet in the epoch buffers would be prohibitive. Therefore, we design an approximate yet efficient summarization method by exploiting the heavy-tail distribution of Internet flow sizes: a flow-level summarization will only achieve a high compression ratio on large, "elephant" flows, but not on small "mice". So our approach attempts to recognize "elephant" flows and summarize their packets only, leaving packets in "mouse" flows as is. Elephants could be efficiently identified with an algorithm such as ElephantTrap [117], or simply by sampling a small fraction of the packets; if both endpoints of a link use the same method for summarization, they will arrive at the same result, and can thus use the commitment protocol to agree on it.

We have implemented a simple, sampling-based flow-level summarization in our SPP prototype, and demonstrated that this is indeed practical. We make two passes over each epoch to summarize the packets. In the first pass, we randomly sampled 20% of its packets and record their flow identifiers as summarization targets. In the second pass, we summarize target flows into flow-level summaries that only include i) number of packets in a flow, ii) size of a flow, and iii) the flow identifier itself. We then write i) the flow-level summaries for the target flows, and ii) packet-level evidence for the unsummarized flows into disk. The computation cost for summarization is 0.23 cores. The data rate is only 8.8 MBps for the OC-192 link; therefore, a 100 GB disk would be able to store the flow-level summaries for 25.3 hours, or more than a day.

3.6.5 Querying

Computation: Upon a query, SPP must freeze and retrieve the evidence that is needed to answer it. The evidence can be constructed by building a MHT with packets in the queried epoch, and tracing the relevant paths from the root to the queried packets. Querying multiple packets in the same epoch only costs marginally more than querying only one packet from that epoch, because queries for packets in

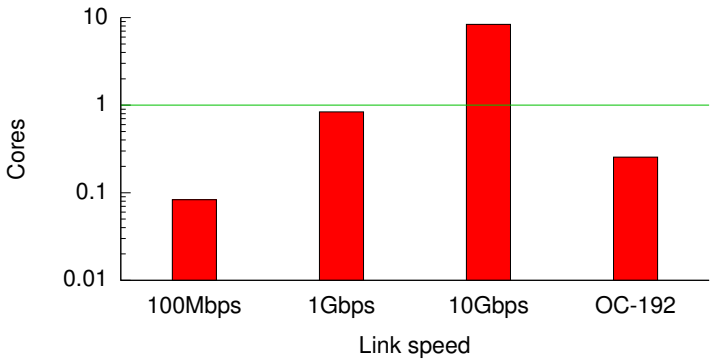


Figure 3.6: Computation cost for answering queries.

the same epoch can be buffered until the end of the epoch and answered altogether. Therefore, the worst-case cost is when MHTs for all epochs need to be reconstructed. Note that this is a simple repetition of the MHT construction at recording time (Section 3.6.1), only this time we do not need to hash the packets again. We show the computation cost for different link speeds in Figure 3.6, and note that our NetFPGA prototype could achieve this at a 10 Gbps rate.

Bandwidth: The bandwidth needed for freezing is low (a single 40-byte packet), so we focus on the bandwidth for retrieving the evidence. Recall that to query the provenance of a packet p on a node n, the querier provide n with p's hash, the number of the epoch p was sent in and the corresponding authenticator, p's index in the epoch buffer, and a containment proof that links the authenticator to the hash; the response contains the same information for the next hop, along with the relevant entry from the action buffer. For a given choice of hash function and signature algorithm, the size of all fields is fixed, except for the containment proof, which grows with the height of the per-epoch MHT. For a 1Gbps link with $T = 100$ms, the size of a single query in our implementation is 680 bytes; for a 10Gbps link, the MHT grows by four levels, and thus the size of a query grows to 760 bytes. Responses are 30 bytes larger because of the additional action buffer entry. Both queries and responses are small enough to fit into a single packet.

We now estimate the *worst-case bandwidth cost* of querying. The cost for a sin-

Core functionality		LoC
Trace a transmitted packet's path	[92]	8
Trace a received packet's traversed path	[157]	8
Identify node on path that drops a packet	[118]	8
Attest to the transmission of a packet	[82]	9
Identify link on path with highest delay	[110]	24
Compute a link's average throughput	[119]	26

Table 3.3: Several applications we built with SPP, and the lines of code (LoC) they required. The code can be found in the appendix.

gle query (with freeze packet and headers for request and response) is 40+680+ 710+2×28=1486 bytes, or 4.95 times the average packet size of 300 bytes. Thus, if a node allows up to 1% of its traffic to be queried, query-related packets would account for 4.7% of its traffic.

3.6.6 Comparison with SNP

We next compare SPP with SNP [180], the state-of-the-art system for secure provenance. SNP has been applied to BGP, Chord, and Hadoop, but it is not designed to handle the high data rates on the network data plane. To demonstrate this, we ran SNP and SPP side by side, and we streamed packets through both of them; we report the results we have obtained on the 1Gbps trace with 40B packets.

Disk space: At an auditing rate of 1%, SPP wrote 65% less evidence on disk than SNP. This is because SPP's evidence mostly consists of 20-byte hashes and not the longer RSA signatures that SNP requires.

Bandwidth: SPP consumes 98.2% less bandwidth than SNP. This is because SPP's can commit to a batch of packets using a single root hash, whereas SNP has to commit to each packet one by one.

Computation: On the same trace, SPP runs 1378.5 times faster than SNP. This is because SPP only performs two hashes per packet and one RSA signature per batch, whereas SNP needs to sign every single packet. At this speed, SNP would require the equivalent of more than a thousand CPU cores to process 1 Gbps of traffic in

```
void tracert(Packet *p, Evidence *e) {
  IP *nextHop = gatewayIP;
  Packet *p0 = p;
  do {
    query(&p, &e, nextHop);
    print(nextHop+" "+(e.time-p0.time));
  } while (nextHop != END_OF_PATH);
}
```

Figure 3.7: Code for tracing a packet's traversed path.

software, whereas SPP can do the same with a single core.

3.6.7 Building applications with SPP

To determine whether SPP can fulfill its key promise of supporting a wide variety of diagnostic and forensic tasks, we implemented the core functionalities of six diagnostic and forensic systems from the literature on top of the QUERY primitive that SPP provides. Table 3.3 shows a list of the six systems, along with the lines of code (LoC) in our implementations. The LoC numbers are very low: our most complex application consists of 26 LoC, and four of the six applications have less than 10 LoC. For concreteness, Figure 3.7 shows the slightly simplified code for tracing the path a transmitted packet p has traversed; the code simply iterates through the sequence of hops, starting with the evidence it received when p was originally sent, and outputs the IPs and latencies it encountered along the way.

The low number of LoC may seem surprising, but the reason is that most of the complexity in the original applications was in the special-purpose network primitives they proposed, or in smart techniques for leveraging and working around existing primitives (such as ICMP TTL Exceeded) that were originally introduced for some other purpose. With SPP in place, the applications we tried reduce mostly to gathering the relevant evidence and/or performing some simple post-processing. Therefore, SPP does deliver its key benefit: a *single* primitive that can handle most existing – and hopefully future – diagnostic and forensic tasks.

3.7 Deployment

As with most extensions to the Internet architecture, getting SPP deployed at scale would not be easy, so it would be unrealistic to expect that *all* ASes would immediately install SPP on *all* their switches, routers and middleboxes. However, SPP has a number of properties that could help facilitate its deployment. Below, we discuss possible strategies for deploying SPP partially and the guarantees a partial deployment would provide, ways of using SPP with existing router hardware, and incentives for ISPs to deploy SPP.

3.7.1 Partial deployment

SPP can be usefully deployed at an individual ISP to diagnose ISP-local problems, so there is no need for a global "flag day". Its benefits increase gradually with the size of the deployment: the more ISPs have support for SPP, the more coverage SPP would have, and the higher the chances that a problem will occur on a path segment that is SPP-enabled (and can thus be diagnosed with SPP). This is very different from a protocol like S-BGP, which must be deployed almost universally to be useful.

For a strategic first deployment, one possible approach would be to first secure cross-domain links with SPP-enabled NICs, and retain legacy NICs for internal links. Similar to the strategy in NetReview [80], an AS can periodically disseminate the authenticators its border routers have received to its neighboring ASes. In such a partial deployment model, we would lose the capability of tracing a packet all the way down its path, as a recursive invocation of QUERY would terminate at the first hop without an SPP deployment. Nevertheless, we still gain useful guarantees with regard to neighboring ASes with cross-domain SPP links. The generated evidence at border routers can be used to detect problems and resolve dispute as to which AS misbehaved. More concretely, this would be sufficient to localize problems to a particular AS (or, from the perspective of adjacent ASes, sufficient to show that

the problem was *not* caused by them). It would not help with AS-local diagnostics, but this capability could be added gradually by enabling SPP on additional devices within the AS.

Impact on guarantees: If a path contains both SPP-enabled and SPP-agnostic devices, provenance information is only available for the segments of the path that are SPP-enabled. In other words, the provenance graph has "holes". Vertexes at the rim of a "hole"—say, a SEND from an SPP-enabled switch to an SPP-agnostic one—are not verifiable because SPP-agnostic devices do not provide evidence. We call such vertexes *rim vertexes*. However, all other vertexes are verifiable as before.

Within a contiguous SPP-enabled path segment, QUERY can be used as previously described. However, in a sparse deployment, there could be multiple short segments, so it would be useful for queriers to continue their query across the rim vertexes and onto the next segment. This is possible, though it would require some additional mechanism. The two key challenges are 1) finding the next SPP-enabled node on the path, and 2) handling packet mutations, such as decremented TTL values or other header option changes. One possible way to handle this, at least within a single AS, would be to use tunneling; it should also be possible to guess a small set of possible next-hop nodes based on the packet's destination IP, and to try all of these to find the one that the packet actually went through. Simple mutations could be handled by excluding the relevant fields from the packet hash, as, e.g., in SPIE [157],

In terms of the accuracy, completeness, and privacy guarantees discussed in Section 3.3.6, the accuracy and privacy guarantees would still hold, but we would lose the completeness guarantee on SPP-agnostic nodes because the evidence pertaining to them cannot be collected in a verifiable manner.

More concretely, the provenance graph G_e is still *accurate*, as the integrity of evidence is protected by the cryptographic signatures in the commitment scheme: correct nodes can still be trusted to generate evidence pertaining to them, while faulty nodes still cannot implicate correct nodes by generating false evidence on their own.

The provenance graph is still *private* because queriers still cannot use SPP's QUERY interface to learn information about packets that are not already visible to them – queriers cannot correctly guess the authenticator that is required to initiate a query.

However, the *completeness* guarantee only holds for links on which both endpoints implement SPP. Recall that the first condition in the completeness guarantee says that each vertex in G will also appear in G_e, the graph generated by the collected evidence. Clearly, in a partial deployment, G_e is restricted to capturing only those events that are secured by SPP. The second condition says that detectably faulty nodes will be exposed by a recursive invocation of QUERY. But since nodes that do not support SPP would break the capability of recursively invoking QUERY through them, queries will have to stop at the first hop without SPP support. As a result, there might exist faulty nodes that "hide" beyond the "broken" SPP chain that can otherwise be detected by a full SPP deployment.

Therefore, in a partial deployment where SPP is only enabled on the border routers, we would have the following properties. Any events that happen on those border nodes will retain the desirable properties of completeness, accuracy, and privacy. However, if faults happen deep within a network (where SPP has not been deployed yet), or recursive tracing is necessary to trace faults down a path, fault detection would be more difficult. We note that this still seems to be a useful guarantee, given that a) events within the same domain fall under the administration of a single trust root, so they can be examined relatively easily by an operator; and b) events that cross neighboring trust domains (where most of the peering contracts take place) can still be verified and attributed by the use of the SPP protocol.

3.7.2 Using existing routers

Deploying SPP does not necessarily require new equipment. Consider a cross-domain link on which neither side has implemented SPP in their routers, but they would still like to use SPP. This is still achievable by attaching a separate machine

or FPGA, i.e., an SPP proxy, to each endpoint of the link, and mirroring all traffic to them. The proxies can make commitments and enable audits according to the SPP protocol, while the routers themselves forward traffic as they would with today's routing fabric. Similarly, if a cross-domain link has one SPP-enabled router on one side and one conventional router on the other, a similar standalone proxy could be used to pair with the SPP-enabled node.

This approximation may cause some inaccuracy when the traffic mirroring process causes packets to be dropped or garbled. For instance, for a link $A \rightarrow B$ where A's mirroring has caused packet loss or corruption that is not visible by B, the SPP proxies could generate commitments mismatch when in fact no fault has happened. But we could handle this 'false alarm' by allowing a certain fraction of commitment mismatches on links with a box-facilitated SPP deployment.

Impact on guarantees: We now discuss the guarantees that can be provided by links with SPP proxies. There are two cases: a) only one side of the link installs SPP but the other side uses a proxy, and b) both sides use proxies. Since case b) is essentially a simple extension of case a), we primarily focus on case a) in the ensuing discussion.

On a link $A \rightarrow B$ where A is SPP-enabled but B uses a proxy, the *accuracy* guarantee holds only if the traffic mirroring process successfully forwards every packet from A to its proxy. In this case, effectively, the commitment protocol between A's proxy and B will work as if both sides had deployed SPP.

However, if routers A and B work correctly but the traffic mirroring process could drop or garble some packets from A to its proxy, the accuracy guarantee will be weakened. In this case, since the SPP module on A's side works on a separate proxy, the packets that have been dropped or garbled during traffic mirroring will not be committed on A's side (though the packets arrived at B without error). So when A's proxy and B try to establish agreement, it may appear as if B had generated some extra packets in the epoch but as a matter of fact this is due to the inaccuracy introduced by the proxy.

This translates into the following accuracy guarantee: in the epochs where *A* and *B* are able to establish agreement, SPP's accuracy holds as with the case without a proxy; in the epochs where the traffic mirroring malfunctions, SPP will lose accuracy *for those epochs only*. Since the traffic mirroring malfunction is expected to be rare, links with proxies could allow a small number of commitment mismatch to happen, and record these epochs where SPP cannot provide guarantees.

The analysis for *completeness* guarantee is similar: Completeness still holds when agreement can be established; but for the epochs where agreement cannot be established, the completeness guarantee will be weakened since SPP cannot distinguish between the case where *A*'s proxy dropped packets and the case where B injected extra packets.

Privacy still holds because neither side can query packets from each other if they have no prior evidence, i.e., the packets have to be visible.

3.7.3 Incentives for deployment

One primitive, many applications: As we have argued, it should be possible to implement a variety of existing diagnostic and forensic systems on top of SPP. Thus, although deploying any new feature in the data plane would not be cheap, at least this effort would have to be spent only once (rather than once for each specialized solution), and it would yield a solution for a wide range of problems.

Few changes to the protocol stack: SPP leaves the current protocol stack (almost) untouched; it mostly "sits on the sidelines" and collects information about the traffic it observes. (The one change it does require is the additional link-layer header for the commitment protocol, which is only visible to the routers on that particular link.) Thus, SPP requires much fewer changes than a design that introduces a new kind of addresses [34] or major packet header changes [135].

Given the above reasons, we believe that ISPs can benefit from deploying SPP. The large ISPs, such as tier-1 ASes, tend to adopt new technologies and best prac-

tices early to increase their competitiveness. Given the diagnostic complexities and security issues in today's Internet, SPP seems to be an attractive value-added service to provide to their customers: SPP is designed to support a wide variety of existing diagnostic and forensic primitives [110, 34, 38, 41, 58, 59, 82, 87, 113, 98, 118, 130, 135, 145, 148, 157, 159, 178], so the rationale for deploying these primitives should, at least to some extent, apply to SPP as well.

More concretely, since SPP can handle a wide range of diagnostic tasks, it can, in particular, also handle troubleshooting tasks that are directly useful to an individual ISP (analogous to NetSight [87]), even without considering the rest of the Internet. For instance, SPP could serve as a troubleshooting tool, analogous to NetSight [87]. Thus, each ISP would initially have at least some incentive to deploy SPP in its own network, independent of what everyone else is doing. SPP is perhaps a bit heavyweight for any single purpose, and, if this were the *only* usage scenario the ISP cared about, it might be better off with a specialized solution, such as the original NetSight. However, SPP has a variety of other uses (e.g., detecting compromised routers), so an ISP might be willing to shoulder some additional cost for the extra flexibility.

Moreover, SPP can help with cross-domain fault localization, which is notoriously difficult. For instance, suppose two adjacent ISPs cannot agree whether a path performance problem lies in one ISP or the other. Today, this situation might involve long phone calls between the ISP operator teams, and potentially some customer dissatisfaction on both sides, even at the ISP who is not at fault. This creates a triple incentive to deploy SPP: 1) ISPs might prefer to peer with networks that support SPP, to better diagnose problems in these networks; 2) ISPs might adopt SPP in their own network to distinguish themselves from competitors and to highlight their own reliability; and 3) ISPs might adopt SPP to quickly establish that the problem an angry customer is reporting is *not* on their side.

Ideally, these incentives would initially lead to the formation of "deployment

islands", which would then slowly grow until they start to merge. In the process, the fraction of a typical path that would be covered by SPP would slowly increase, leading to better and better end-to-end diagnostics.

3.8 Related Work

Here, we review existing literature on network diagnostics and forensics that are particularly related to SPP, and note that related work on network provenance has been described in Chapter 2.

Specialized primitives: As discussed in Section 5.1, there is a rich literature on systems that solve a *particular* diagnostic or forensic problem [110, 34, 38, 41, 58, 59, 82, 87, 113, 98, 118, 130, 135, 145, 148, 157, 159, 178]. To address all of the underlying problems, it would be necessary to deploy all of these systems in combination. In contrast, SPP aims to provide a single primitive that can be used for a broad variety of tasks.

Packet-level diagnostics: SPIE [157], HAL [82], and NetSight [87] resemble SPP in that they all "remember" every single network packet. However, SPIE cannot reliably identify a specific packet due to the use of Bloom filters, and HAL only collects per-packet evidence but does not perform diagnosis. NetSight [87] is closest to SPP: it assembles a "history" of each packet for SDNs. However, NetSight provides no security guarantees in the presence of compromised nodes, and it is designed for a data-center setting, where packet traces can be recorded without privacy concerns and data does not need to be shared with other domains. UnivMon [114] and OpenSketch [172] are recent proposals for flow-level monitoring counters based on sketches. These approaches are useful for gathering traffic statistics, but, unlike SPP, they do not provide packet-level provenance data.

Accountability: SPP is similar in spirit to previous proposals for network-level accountability, e.g., "packet obituary" [36] that reports packet drops, or AudIt [37] that provides secure records of delay and loss rates. Network Confessional [38] uses

a similar retroactive sampling approach to prevent special treatment of the sampled packets; however, it focuses on forwarding performance verification, not direct support for diagnostics or forensics. PAAI [177] also uses retroactive sampling to track lost packets, but it assumed that end hosts are always honest.

3.9 Conclusion

As the large number of proposed extensions shows, the current Internet architecture does not support diagnostics and forensics very well. However, most existing proposals are specialized solutions; thus, a comprehensive solution would require deploying several of them concurrently, at a substantial cost – in terms of both overhead and complexity. In this chapter, we have made a case for a network-level primitive that can support a variety of different diagnostic and forensic applications, and we have also presented SPP as a concrete proposal. Our evaluation shows that SPP can be implemented efficiently in hardware and can approximate a variety of common diagnostic and forensic tasks.

4

Differential Provenance

Distributed systems are not easy to get right. Despite the fact that researchers have developed a wide range of diagnostic tools [101, 169, 171, 120, 151, 167, 60], understanding the intricate relations between low-level events, which is needed for root-cause analysis, is still challenging.

Recent work on *data provenance* [183] 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 [183], SNP [180], and Y! [169].

However, while such a comprehensive explanation is useful for diagnosing a prob-

lem, 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 diagnosing 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" [77]); 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 [45]. 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 4.1.) A good debugger should be able to pinpoint just the broken flow entry and leave out the irrelevant consequences.

In this chapter, 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 run-time 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 4.2);
- DiffProv, a concrete algorithm for generating differential provenance (Section 4.3);
- a DiffProv debugger prototype (Section 4.4); and
- an experimental evaluation in the context of SDNs and Hadoop MapReduce (Section 4.5).

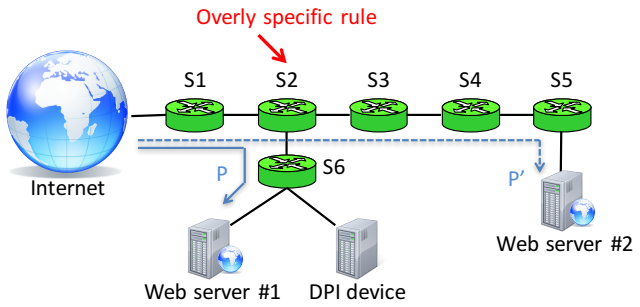


Figure 4.1: Example scenario (SDN debugging).

We discuss related work in Section 4.6, and conclude the chapter in Section 5.6.

4.1 Overview

Figure 5.1 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`.

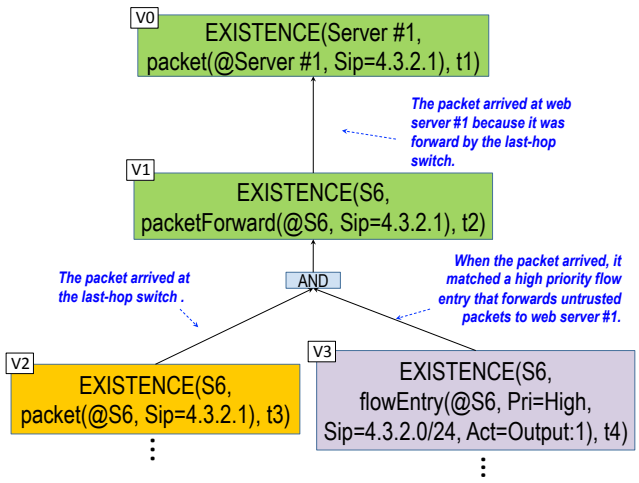
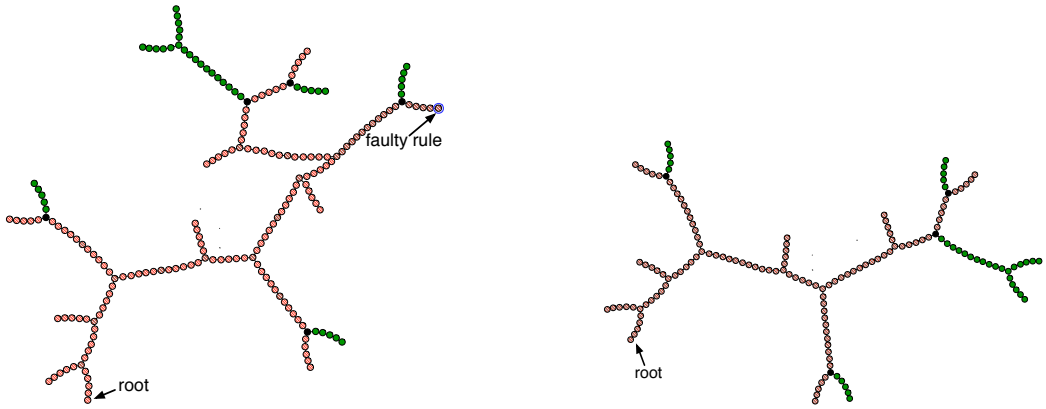


Figure 4.2: An example provenance tree



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

Figure 4.3: Provenance trees for P' (a) and P (b) from Figure 5.1. Each circle corresponds to a box in Figure 4.2, 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 (a) that represents the root cause of the routing error that affected P'.

4.1.1 Background: Provenance

Network provenance [183] 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 4.3(a) sketches the provenance of the packet P

from Figure 5.1 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 [47, 69, 169].

4.1.2 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 4.3(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 [169]. Hence, extracting a concise root cause from a complex causal explanation remains challenging.

4.1.3 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 4.3(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 4.1.5.

4.1.4 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.

4.1.5 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 [45] – 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 4.3a and 4.3b 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.

4.1.6 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".

4.2 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 [116], few deployed systems are written that way today. However, DiffProv is not specific to the declarative model: in Section 4.4, 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 [131], an imperative language.

4.2.1 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.

4.2.2 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 [182]. 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 [182] 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.

4.2.3 Towards a definition

We are now ready to formalize the problem we have motivated in Section 4.1. 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 4.1, 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

```
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 4.4: Pseudocode of the DiffProv algorithm. The FINDSEED, FIRSTDIV, MAKEAPPEAR, and UPDATETREE functions are explained in Sections 4.3.2, 4.3.4, 4.3.5, and 4.3.6 respectively. The CREATETAINT, PROPTAINT, and APPLYTAINT functions are introduced to establish equivalence between corresponding tuples in T_G and T_B (Section 4.3.3).

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 4.3.3.

With these refinements, we arrive at our final definition:

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 an equivalent of T_G, and 2) preserves s_B.

Figure 4.5 illustrates this definition with a simple derivation rule `C(x,y²,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.

4.3 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.

4.3.1 Roadmap

The DiffProv algorithm is shown in Figure 4.4. 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 4.3.2). 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 4.3.3). 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

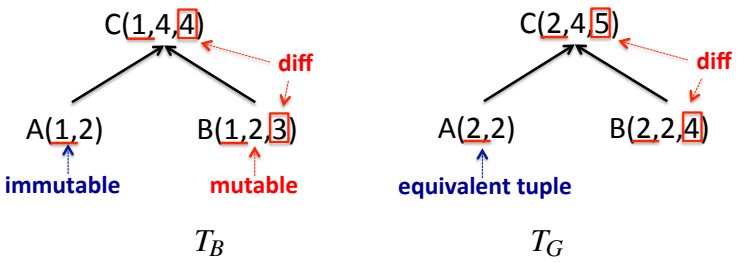


Figure 4.5: A simplified example showing the differential provenance for a one-step derivation. `A(1,2)`, `A(2,2)` are the seeds; equivalent fields are underlined, and differences are boxed. Differential provenance transforms `B(1,2,3)` into `B(1,2,4)` to align this derivation.

same under the equivalence relation defined earlier (Section 4.3.4). 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 4.3.5–4.3.6). 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.

4.3.2 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 4.2.2) 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.

4.3.3 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 4.2.3); 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 `portAnd LastOctet(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.

4.3.4 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 4.3.3), 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 4.3.3). 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.

4.3.5 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 instance, 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 4.3.6). 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.

4.3.6 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.

4.3.7 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 4.2.3, 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.

4.3.8 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 [182], 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.

4.3.9 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 5.1.3), 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 [174] and the "guided probes" idea in Everflow [184].

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 [33] that considers provenance for explaining aggregation results.

Non-determinism: Replay-based debuggers such as DiffProv, ATPG [174], 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.

4.4 Implementation

Next, we present the design and implementation of our DiffProv prototype. We have implemented a DiffProv debugger in C++ based on RapidNet [18], 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 [116]. 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 [132]. 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 [180]. We applied this to the complex SDN scenario in Section 4.5.7, where the recorder tracks packet-level provenance in Mininet [13] 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 pro-

grams that accepts programs written either in native NDlog or in NetCore (part of Pyretic [131]). When a NetCore program is provided, our front-end internally converts it to NDlog rules and tuples using a technique from Y! [169].

Logging and replay engines: The logging and replay engines are needed to support temporal provenance as described in Section 4.3.8, 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 4.3, and then issues replay requests to update the trees.

4.5 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?

4.5.1 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 5.1, 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 [174] 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 [13] with a Beacon [23] controller), we defer the discussion of this experiment to Section 4.5.7.

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.

4.5.2 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 [139].** 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 4.1.

- **SDN2: Multi-controller inconsistency [60].** 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 [146].** 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 ac-

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 4.1: 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.

cidentally 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.

4.5.3 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! [169] 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 4.1.5, 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 4.1 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 4.1.5.) 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.

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.

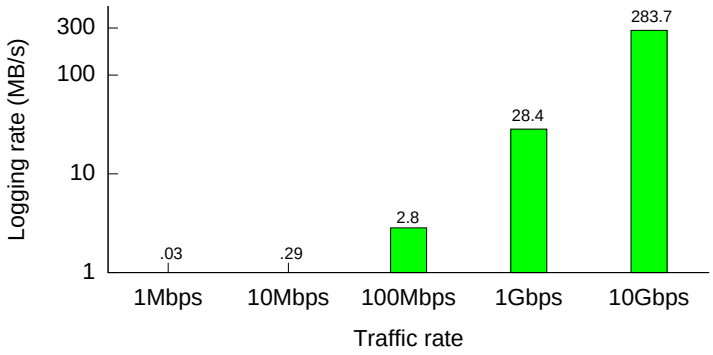


Figure 4.6: Logging rate for different traffic rates.

4.5.4 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 [169] 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%.

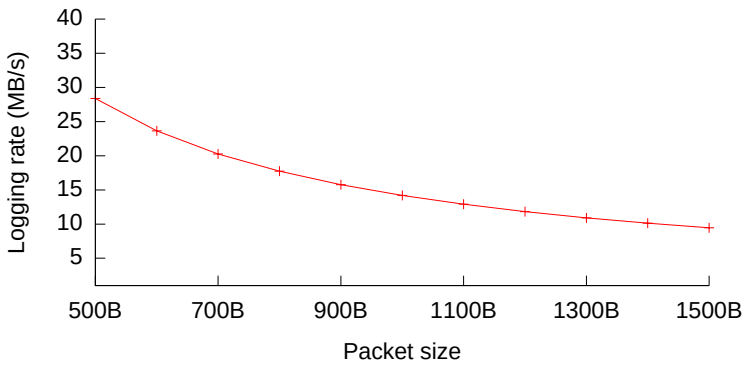


Figure 4.7: Logging rate with different packet sizes at 1Gbps.

4.5.5 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 4.6 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 4.4). 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 4.7 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 [87] or Everflow [184]: 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.

4.5.6 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! [169] 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 4.8 shows our result: except for SDN4, all other queries were answered within one minute; 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

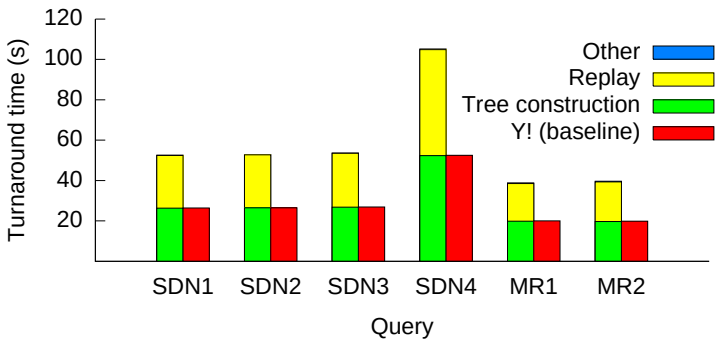


Figure 4.8: Turnaround time for answering differential provenance queries (left), and Y! queries (right). DiffProv's reasoning time (shown as "Other") is too small to be visible.

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 4.8, 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 4.9. 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.

4.5.7 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

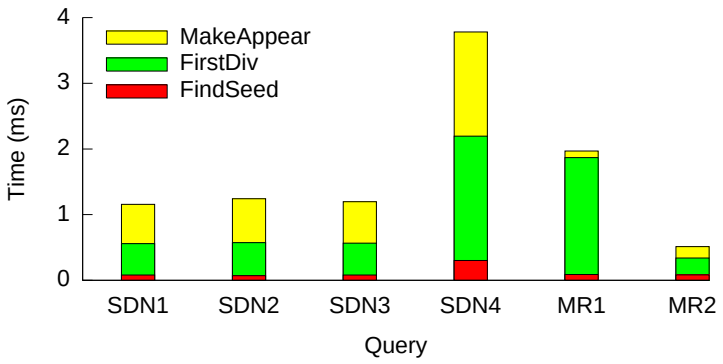


Figure 4.9: Decomposition of DiffProv's reasoning time. For SDN4, we have stacked its two rounds together.

real-world configurations and realistic background traffic.

Basic setup: Our scenario is based on the Stanford University network setup obtained from ATPG [174]; 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 [13], and controlled by a Beacon [23] 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 [174] 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 former. 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 [150] that is based on fuzzy testing, PeerPressure [167] that uses statistical analysis to find the likely value of a configuration entry, NetMedic [97] 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.

4.5.8 Experience

To check whether DiffProv's diagnostics indeed translates to significant time savings, we have performed a study with eight graduate students from our department who specialize in networking research. All participants specialize in networking research, and have prior exposure to provenance. We presented each participant with an illustration of the SDN1 scenario in Figure 5.1, the controller program, the network's intended operation, and we answered any questions they had. We then presented them with a description of the fault (a packet was routed to the wrong server) and the corresponding provenance tree. Since the original tree has 156 vertexes and is thus quite complex, we applied some of the heuristics from Y! [169] (e.g., by summarizing derivation chains in a single super-vertex), yielding a much simpler tree with 56 vertexes. We asked each participant to diagnose the problem and suggest a root cause.

Seven of the participants took 20, 20, 24, 35, 40, 45, and 71 minutes (averaging 32 minutes) to diagnose the problem; one gave up after 30 minutes. We then showed each participant the output from DiffProv, and each quickly agreed that this was consistent with their understanding of the root cause. We note that this is not a formal user study; there was no control group, the number of participants is very small, and the participants were not representative of the likely users of DiffProv (network operators). Nevertheless, our results suggest that diagnosing faults with a complex provenance tree can take a substantial amount of time, which DiffProv's much simpler output can help to reduce. We plan to carry out a more detailed user study as our future work.

4.6 Related Work

Network diagnostics: A variety of diagnostic systems have been developed over time. For instance, Anteater [120], Header Space Analysis [101], and NetPlumber [100] rely on static analysis, while OFRewind [171], Minimal Causal Sequence analysis [151], DEMi [150], and ATPG [174] 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 [167], EnCore [176], ClearView [141], and Shen et al. [152] 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 [40] does take causality into account, but it can only compare equivalent systems (e.g., "sick" and "healthy" computers), not events. NetMedic [97] 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 [103], which diagnoses liveness violations by finding a critical state transition. However, [103] 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 [87] records traces of packets as they traverse the network, and Everflow [184] 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.

Moreover, we note that related work on network provenance has been described in Chapter 2.

4.7 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.

5

Causal Networks

So far in this dissertation, we have presented systems that can secure high-speed provenance data in adversarial environments and use the provenance data to identify root causes of network problems. In this chapter, we take the next step and leverage the collected provenance data to generate network repairs. This is challenging because implementing a repair can have network-wide effects – it is important to ensure that a repair not only fixes the symptom at hand, but also avoids undesirable side-effects in the network.

Existing approaches [86, 183, 87, 169] have proposed "network debuggers" to help operators with diagnostics. Analogous to conventional debuggers, network debuggers (e.g., `ndb` [86]) accept a buggy behavior (e.g., an unexpected event or state) as input, produce a "backtrace" that chronicles what happened in the network, and link the buggy behavior to certain parts of the network execution and/or its root causes. But most debuggers only explain how a problem surfaced, but do not consider how to repair it – the latter is an operator's ultimate goal, but it is also more challenging. Recently, researchers have proposed another kind of debuggers [168, 147, 50]

to generate network repairs in an automated fashion, which can potentially offer operators a lot more help. However, [147] can only repair static data-plane configurations; although [168] and [50] are applicable to distributed systems in general, they only produce repairs that rectify a *single* buggy behavior without considering the side-effects on other parts of the network. As a result, repairs generated in this way may break the network elsewhere.

Consider a scenario where Alice sees her DNS server S receive a spurious HTTP packet, and she asks her debugger to generate a repair that prevents the packet from arriving at S. Since Alice's request only mentions one goal (removing a packet from server S), her debugger may produce a repair that simply drops all HTTP packets at the network ingress, or a repair that disconnects S from its last-hop switch. Both repairs faithfully remove the packet from S, but both would cause new problems elsewhere. The ideal repair, on the other hand, may be "change the flow entry at S's last-hop switch to forward HTTP packets to a different port". In network operations, the risk of "death by failure recovery" [79] is non-trivial – some of the biggest network incidents have resulted from applying an improper repair [62, 71, 72, 129]. Therefore, it would be valuable to have a debugger that not only rectifies the problem at hand, but also minimizes undesirable side-effects elsewhere.

We argue that the problem at its core is that *existing debuggers have a very restrictive interface*, which accepts one single buggy behavior as input. As a result, although Alice has many constraints about what the correct network ought to look like, she can only convey to her debugger part of the story (i.e., the correct network should not contain the input event or state). If Alice were able to tell her debugger all of her constraints – e.g., that HTTP traffic should not go to server S, but S′ instead, and that DNS traffic should still be processed by server S – the debugger would have a lot more information that it can use to generate a successful repair.

We observe that this is analogous to *automated program repair* [73]. For some time, the software engineering community has been considering the problem of gen-

erating *program* repairs with few side-effects. One effective solution this community has developed is to tell the debugger not just about the bug itself but about *multiple* desirable properties. This extra information can then guide the automated debugger towards a better repair. Our key insight is that it should be possible to adapt this idea and apply it to the networking domain as well: if network debuggers had a more expressive interface, operators would be able to articulate more clearly what they want, and thus obtain better repairs.

In this chapter, we make a case for this new approach, which we call *intent-based diagnostics*. We propose Aladdin, a language for describing diagnostic intents. When an operator sees a network problem, she can use Aladdin to describe in a *diagnostic intent* what she wishes to happen instead. This interface is far more expressive than just one buggy event, because an operator can describe in an intent many events and how they relate to each other, e.g., HTTP traffic should be forwarded to server S′ *and* DNS traffic should be forwarded to server S. Our debugger, NetGenie, can then leverage all expressed constraints to find a high-quality repair. NetGenie needs to find a small repair that achieves *multiple* goals; this can involve a huge search space, so simply trying random changes would not work. We address this using *causal networks* [85], a generalization of network provenance [169] that encodes causality between many network events. NetGenie then leverages this causality information to perform *targeted repair*.

We proceed with an overview in Section 5.1. We then formalize the network repair problem in Section 5.2, design the NetGenie algorithm in Section 5.3, and present the evaluation in the context of SDNs in Section 4.5. Finally, we discuss related work in Section 5.5, and conclude this chapter in Section 5.6.

5.1 Overview

We begin with a very simple scenario in Figure 5.1 that illustrates the type of problems we are considering. It shows an SDN with three switches and three servers.

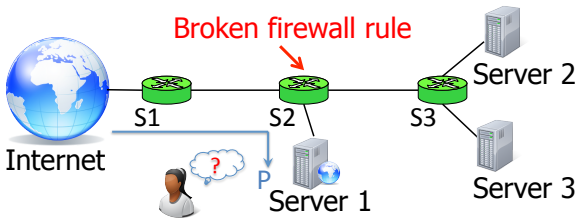


Figure 5.1: An example SDN network.

The operator, Alice, would like requests from a trusted origin (e.g., traffic from 10.0.0.0/24) to be load-balanced between servers 2 and 3, and requests from untrusted origins (e.g., traffic from 1.2.3.0/24) to be processed by server 1 (e.g., because it is in a DMZ). To implement the security policy, she configured two firewall rules on S2: a) R_1, a rule that forwards the trusted traffic to S3, and b) R_2, a wildcard rule that forwards the untrusted traffic to server 1. To implement the load-balancing policy, she configured two load-balancing rules R_3 and R_4 on S3 to match traffic from 10.0.0.0/25 and 10.0.0.128/25, and forward it to servers 2 and 3, respectively. However, Alice made a mistake in R_1, which was misconfigured to match the subnet 10.0.1.0/24 instead of 10.0.0.0/24. As a result, Alice sees a mix of trusted and untrusted traffic at server 1, and she would like to diagnose the problem.

5.1.1 Why are existing debuggers not enough?

Most existing debuggers [87, 86, 28, 169, 66, 183, 161, 180] can explain how an event in a distributed system came about, in the form of a "backtrace". For instance, if a packet p from 10.0.0.1 was misrouted to server 1, `ndb` [86] would produce a backtrace like the following: p was received by server 1 at 10:51pm, because 23ms ago, it was transmitted from switch S2 at port #1 due to a flow entry match; this in turn was because 40ms ago, S1 transmitted p at port #0 due to another flow entry match, etc. However, producing a backtrace is only the first step – the operator still needs to identify the root cause in the backtrace, and decide on a repair.

Recently, researchers have proposed another kind of debuggers that can generate repairs for distributed systems [50, 168]. Given a buggy behavior, e.g., the misrouted

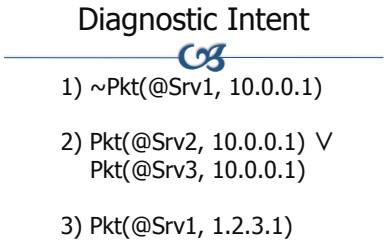


Figure 5.2: An example diagnostic intent.

packet p in Figure 5.1, they can propose changes to the network to remove the bug. However, repairs produced in this way are narrowly targeted at a single event and thus may break the network elsewhere. For example, in order to remove p, a debugger might propose to disconnect server 1 from S2, or to drop all packets at S1. Both are *valid* repairs in that they successfully remove p, but both have undesired side-effects.

We observe that this is because the operator has many constraints on what a correct network should look like, but existing debuggers do not have an interface for her to express them. Suppose that we had a debugger that accepts *multiple* constraints of what a successful repair should achieve – e.g., packets in 10.0.0.1 should not be routed to server 1, but to server 2 or 3 instead; moreover, other packets should still be routed to server 1 – the debugger would be much more likely to identify the broken entry at S2 to be the culprit.

5.1.2 Approach: Intent-based diagnostics

We propose to build a new debugger that can generate repairs based on a *diagnostic intent*, which is far more expressive than a single event. With this debugger, an operator's main duty is to describe what she wishes to happen in her network, using a language that we call *Aladdin*. Aladdin supports three simple operators $\wedge$ (`and`), $\vee$ (`or`), and $\sim$ (`not`), which can be used to compose complex intents from simple ones. For instance, to generate a repair for the scenario in Figure 5.1, an operator can write down her intent, as shown in Figure 5.2: 1) packets from 10.0.0.1 should not have been processed by server 1; 2) they should be processed by servers 2 or 3 instead;

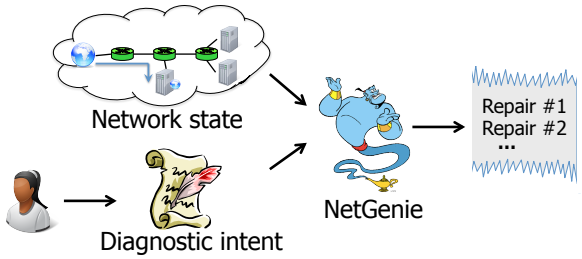


Figure 5.3: The workflow of intent-based diagnostics.

and 3) packets from 1.2.3.1 should still be processed by server 1. Our automated debugger, which we call *NetGenie*, then takes into account all constraints expressed in the intent, and suggests repairs that satisfy all of them, as shown in Figure 5.3.

Goals and non-goals: At first glance, the use of a high-level intent might be reminiscent of network verification or synthesis (e.g., HSA [101], NetGen [147], NetEgg [173], etc.), which are still restricted to certain types of network [165, 140], not general distributed systems. But the goal of NetGenie is not to perform full verification, i.e., to guarantee the correct behavior for *all possible packets* in the header space, but rather to repair specific problems that have manifested in the network. This resembles the difference between program verification (which is universal but expensive) and program debugging (which is case-specific and much cheaper). In other words, a generated repair may not guarantee that `Pkt(@Srv1,10.0.0.1)` will never occur again, regardless of network topology, packet history, etc.; however, it will fix any problems that have affected the packets in this space so far. This is less ambitious than verification, but in return, NetGenie gains the ability to repair much more complex networks, and even other kinds of distributed systems (analogous to [168] and [50]), which is still beyond the reach of current network verification or synthesis techniques [165].

5.1.3 Would NetGenie be practical?

It is reasonable to ask how common it is for a repair task to have multiple goals. To answer this question, we reviewed 62 incident reports from Google Cloud Status

Dashboard [11], which documented major network faults in Google's data centers, from April 2014 to March 2016. We found that 46 (74%) of them require two to ten constraints to hold at the same time in a successful repair. We then formulated an intent for each scenario, and found that a typical intent can be complex enough to involve 2 to 17 Aladdin operators.

One might also wonder if NetGenie would be perceived as difficult to use, as it requires some additional work from the network operators (writing down their diagnostic intent in Aladdin). To understand this, we studied operators' common practices from several sources, including mailing lists [15, 25], surveys [138], online discussions [24, 163, 106, 93, 112], and made two observations. First, operators are masters in simple CLI languages, e.g., router configuration languages. As the Cisco manual indicates, configuring the BGP protocol could involve 59 different commands [9]. Moreover, even a medium-sized network could be configured with 1,500 ACL rules and 757,000 forwarding entries [101, 174]. As a survey with 217 network operators shows [138, 104], operators prefer languages that are "concise, intuitive, easy to understand". Second, due to the rise of SDNs, more and more network operators are *becoming* programmers [138, 106, 166]. We see many operators transition from NetOps to DevOps [112, 93], and they are frequently learning new languages [163, 112, 24] such as Puppet [17], Chef [8], Ansible [4], etc.

Given that Aladdin is no more complex than the languages already in use, and that network repairs can be tricky to get right, we believe that operators' benefits in using NetGenie should far outweigh the effort of writing down their intent.

We note that NetGenie does not require operators to write down her intent exhaustively, which will likely involve a huge number of constraints, but only a subset of them. This is similar in spirit with other systems that reason with user-provided examples, including NetEgg [173] on SDN policies, Foofah [94] on semi-structured data transformations, QuickCode [78] on string transformations in spreedsheets, Storyboard Programming [155] on pointer manipulations on data strucutures such

as linked lists and binary trees, and [154] on number transformations such as formatting and rounding. A small number of examples often suffices in practice, because each example tends to cut down the search space quite significantly. For instance, in FooFah [94], one or two input-output examples are already enough for the system to find the correct transformation; similarly, in NetEgg [173], each scenario also only contains one to three examples.

In fact, we have similar observations: in Section 5.4, we will see that relatively simple intents are also sufficient for NetGenie to generate high-quality repairs. That said, since NetGenie is restricted to work with an incomplete intent, any *guarantees* it can provide would only hold on the expressed goals. One implication of this is that, to be on the safe side, operators are still recommended to manually inspect the generated repairs before applying them to the running network.

5.1.4 Challenge

Finding a repair that satisfies all constraints could involve a huge search space, so naïvely trying change combinations would not work. We observe that tracking causality, as in network provenance [183], can be a useful starting point: if NetGenie tracks the causal connections between events and state, it can then pinpoint just a few places in the network that have affected the event, and perform *targeted repair*. However, reasoning about each event separately would not work: a fix that is effective for one event could cause problem for another. To address this, we use a data structure called *causal networks*, a concept in classic causality theory [85] that can be seen as a generalization of provenance. With causal networks, NetGenie can find repairs that fix all problems simultaneously.

5.2 Intent-based network repair

In this section, we formalize the problem of intent-based network repair, and we describe a solution that is based on the concept of causal networks.

```
function CausalNetwork(ψ, S)
    N ← ⟨∅,∅⟩
    for e in ψ do
        if e ∈ S then
            ⟨V,E⟩ ← PosProv(e,S)
        else
            ⟨V,E⟩ ← NegProv(∼e,S)
        N ← ExpandNetwork(N, ⟨V,E⟩)
    return N

function FindRepair(N,S)
    PC ← PrsvConstr(O)
    OC ← ObjConstr(O)
    for each o ∈ O do
        DC ← DC∪ DrvConstr(o)
    Δ ← ∅,k ← 1,C ← PC∪OC∪DC
    while Δ = ∅ & NotTimedOut do
        Δ ← FindKChanges(I,k,C)
        k ← k+1
    return Δ

function FindKChanges(I,k,C)
    I^k ← NextKTuples(I,∅)
    while I^k ≠ ∅, Δ = ∅, NotTimedOut
do
        PC ← PC∪ PrsvConstr(I^k)
        Δ ← FindAssignment(C)
        I^k ← NextKTuples(I,I^k)
    return Δ
```

Figure 5.4: Algorithms for constructing causal networks, and for finding repairs.

5.2.1 System model

For ease of exposition, we use a declarative model in Network Datalog (NDlog) [116], because causality is particularly easy to see in a declarative syntax. We note that provenance is not restricted to declarative languages; it has been widely applied in many imperative systems [89, 69, 57, 132, 43], and provenance can be extracted from imperative systems and even black-box applications [180].

In this model, network state and events are modeled as *tuples* that can either be externally inserted into the network (i.e., base tuples, such as incoming packets at the ingress router, initial configuration state, etc.), or generated by the network protocol (i.e., derived tuples, such as forwarded packets, derived configuration state, etc.). The network protocol can be seen as a set of NDlog rules in the form of `X:-A,B,C`, where `X`, `A`, `B`, and `C` can be either configuration state or network events, which means that an `X` tuple should be generated whenever the network has `A`, `B`, and `C`. Moreover, tuples are organized into *tables*, e.g., a packet could be represented by a tuple in the `Pkt` table.

The provenance of a tuple can be easily defined based on the declarative rules. A base tuple's provenance is itself, since it cannot be explained further. A derived tuple `X`'s provenance is its derivation rule (e.g., `X:-A,B,C`), the tuples that are used to derive it (e.g., `A`, `B`, and `C`), and, recursively, their provenance as well. Therefore, the provenance of an event e would be a tree of events and state, where e is the root of the tree, and e's children are its immediate causes; each node in the tree can in turn be

recursively explained by the subtree rooted at it, until we reach a set of leaves, or base tuples. We can also reason about why an event *failed* to occur, using a variant called negative provenance [169]. This is achieved by tracing along the causal connections that *almost* derived the desired event, and examining which pre-conditions failed to hold (e.g., a packet was received, but it never matched any flow entry).

5.2.2 The network repair problem

We now formalize the problem of network repair, using a "possible worlds" semantics [54].

Intents: We distinguish four kinds of atomic repair intents. The first two are used to describe the desired effects of the repair: the operator can ask for a new tuple to be created (`add`) or an existing tuple to be removed (`delete`). The others are used to avoid side effects: the operator can specify that an existing tuple should not be removed by the repair (`preserve`) or that a non-existing tuple should not be created (`suppress`). More complex intents can be described by disjunctions, conjunctions or negations of these events.

Possible worlds: A network protocol, as modeled by a set of derivation rules R, induces a set of *worlds* $\mathscr{S}$ that are permitted under this protocol. Each world $S \in \mathscr{S}$ describes a particular state that the network could be in, and can be further divided into $S = S_b \cup S_d$, where S_b is the set of base tuples in the network state, and S_d the set of derived tuples. S_b represents the set of initial configurations, incoming packets, and other external inputs to the system. S_d represents the set of derived configurations, forwarded packets, etc., that can be obtained by recursively applying R to S_b until S stabilizes.

We say that a world S is *sound* iff, for each tuple $\tau \in S$, we have either a) $\tau \in S_b$, i.e., it is a base tuple, or b) $\tau \in S_d$, and there exists at least one derivation rule $\tau :\!- \tau_1, \cdots, \tau_k$, where $\tau_k \in S, \forall i \in [1..k]$. In other words, soundness requires that derived tuples cannot appear out of thin air, when none of the corresponding rules have fired. We say that

a world S is *complete* if we have $\tau_1, \cdots, \tau_k \in S$, and τ :- $\tau_1, \cdots, \tau_k$, then we also have $\tau \in S$ – in other words, if every derivable tuple has been derived. If a world is both sound and complete, we call it a *possible world*.

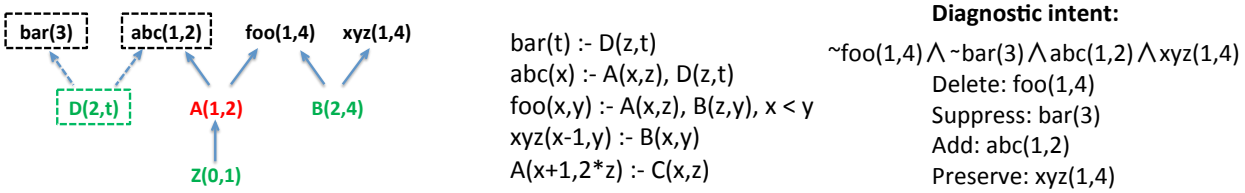
Network repair: Suppose $S^0 = S^0_b \cup S^0_d$ is the state of the network at the time the operator requests a repair. Then the goal of network repair is to find a set of changes to base tuples $\Delta := \{\tau_1 \rightarrow \tau'_1, \cdots, \tau_k \rightarrow \tau'_k\}$, where $\tau_1, \cdots, \tau_k \in S^0_b$, that transforms S^0 into a possible world S that satisfies the diagnostic intent. We call such a Δ a *valid* repair. If there are multiple valid repairs, the goal is to find the one that changes the fewest tuples.

5.2.3 Causal networks

Next, we introduce the concept of causal networks, and discuss how we can use this concept to generate network repairs.

If we only needed to change a *single* tuple, we could track the tuples's preconditions in the form of a provenance tree, and either make the tuple appear by adding all the required base tuples, or make it disappear by removing at least one of them. For instance, if we know that `X:-A,B`, then removing either `A` or `B` would make `X` disappear. However, since we need to produce a repair that accounts for *all* events *at the same time*, it cannot consider each event in isolation. The reason for this is that breaking the pre-conditions of one event might affect other events in unexpected ways. For instance, suppose we need to remove `X` *and* preserve `Y`, where `Y:-A,C`; we cannot do this by first deleting `A` to remove `X`, and then adding it back to obtain `Y`.

To address this, we use a concept that was originally proposed in classic causality theory: *causal networks* [85]. A causal network can be seen as a generalization of provenance. Its vertexes are classified as $V = O \cup I \cup C$, where a) O is the set of *outcome vertexes*, which correspond to the diagnostic events in the intent, b) I is a set of *input vertexes*, which correspond to the base tuples in the network state, and c) C is a set of *causality vertexes*, which capture the derivations from the inputs I to the outcomes



(a) An example causal network. (b) Derivation rules. (c) The repair intent.

Figure 5.5: An example causal network constructed from a repair intent.

O. Given a set of intents (outcome vertexes), we can construct a causal network for them by gradually adding the dependencies for each outcome vertex, as shown in the CausalNetwork procedure in Figure 5.4.

The key benefit of causal networks is that they can encode all dependencies for all diagnostic events. We have also sketched an example causal network in Figure 5.5.

5.2.4 Generating repairs

With a causal network that encodes all events of interest and their inter-dependencies, we are now ready to generate network repairs that account for all constraints. This is done by performing *counterfactual reasoning* on the causal network, which asks the question, "what *could have* happened instead so that the network state would fulfill the intent?", and produces a set of network changes as a candidate repair. Below, we present the algorithm in more detail.

Preservation constraints: We begin with the properties of the network that the operator wishes to preserve. These are described with `preserve` intents, which identify existing tuples in the network that need to be preserved, and `suppress` intents, which identify undesirable tuples that need to be suppressed. Therefore, when attempting to find a repair, the algorithm needs to ensure that the changes satisfy these requirements.

The algorithm achieves this by performing a recursive descent over the causal network, starting from the `preserve` and `suppress` outcome vertexes. For each step, we collect a set of positive constraints C_1 that describe conditions that must hold to

prevent the `preserve` tuples from disappearing, and a set of negative constraints C_2 that describe conditions that will prevent the `suppress` tuples from appearing. For instance, in the example in Figure 5.5, C_1 would contain `xyz.x==1`$\wedge$`xyz.y==4`, which are collected over the outcome vertex `xyz(1,4)`, and C_2 would contain $\sim$`bar.x==3`, which is collected over `bar(3)`. The algorithm then follows the causality chains in the causal network, and recursively explores the set of conditions that need to hold for the current vertexes' children; it stops until a set of input vertexes have been reached. Then, when we are generating repairs to the network, we need to ensure that C_1 and C_2 always hold for the repaired state.

Objective and derivation constraints: Next, we discuss how the algorithm handles `delete` and `add` intents. This step also heavily relies on the causal network as a guide, and uses a similar recursive descent from the `delete` and `add` outcome vertexes.

For a `delete` outcome vertex, we collect a set of constraints that describe the corresponding tuple and its dependencies, and then negate these constraints to reflect the fact that this tuple should be deleted. This can be done by examining the constraints and dependent vertexes for this target tuple. For instance, consider the outcome vertex `foo(1,4)` in Figure 5.5, which has been derived by `foo(x,y):-A(x,z), B(z,y), x < y`. Then, the set of objective constraints would be $\sim$(`foo.x==1`$\wedge$`foo.y==4`).

For an `add` outcome vertex, we collect a set of objective constraints that need to hold in order for the corresponding tuple to appear in the network state. Since this tuple does *not* yet exist, the algorithm must analyze the constraints in the derivation step *counterfactually*, somewhat analogous to negative provenance [169]. For instance, consider a missing tuple `abc(1,2)`, and a potential derivation rule `abc(x,y):-D(z,t),A(x,z)`, the algorithm may find that a dependent tuple `A(1,2)` already exists in the network, but an appropriate `D` tuple is missing. Then, we would collect a set of objective constraints `abc.x==1`$\wedge$`abc.y==2`.

In order to make the objective constraints hold, it may be necessary to make their dependencies become true. For instance, to delete an unwanted event from the network, we must delete one or more of its children; to make a missing event appear, we must add all its missing dependencies. Therefore, to satisfy a set of objective constraints on the outcome vertexes, we need to, recursively, satisfy a set of derivation constraints that describe how the desirable outcomes can be achieved. To this end, our algorithm descends from the set of outcome vertexes along the dependency chains, and collects a set of derivation constraints on each of the causality vertexes.

This can be achieved by propagating the set of objective constraints down the causality chains. For instance, in order to delete the undesirable tuple `foo(1,4)`, the algorithm needs to additional create derivation constraints $\sim$(`A.x==1`$\wedge$`A.y==2`$\wedge$`B.x==2` $\wedge$`B.y==4`); in order to make the missing tuple `abc(1,2)` appear, the algorithm needs to additionally create derivation constraints `D.x==2`$\wedge$`A.y==2`. The algorithm recursively walks down the causal chains while propagating the constraints, until it reaches the set of input vertexes, and return a set of objective and derivation constraints over the causal network. Finally, after collecting the preservation, objective, and derivation constraints, the algorithm can find a repair by jointly solving those constraints.

Notice that, for each step, instead of naïvely trying out all combinations of possible changes, the algorithm relies heavily on the causality chains in the causal network to make *targeted* changes to the current state.

5.2.5 Generating the minimal repair

Now, another challenge arises: since *any* repair that satisfies the set of collected constraints could satisfy the diagnostic intent, the number of possible repairs can be very large, or even infinite. Ideally, we would like to find a *minimal* repair, with a very small number of changes to the network – that is, we would like the repaired network state to be as close as the original state as possible, except for necessary changes

to fulfill the diagnostic intents.

Therefore, we take an iterative deepening approach [108] to finding the minimal network repair, as shown in the FindRepairs procedure in Figure 5.4. In the k-th round of the search (initially, $k = 1$), the algorithm attempts to find a repair with exactly k changes to the base tuples, or input vertexes; it will only proceed to the $(k+1)$-th round if the current round cannot produce a repair. If any round finds a repair successfully, the algorithm terminates and output the Δ it has found; if the current k does not yield a repair, then the algorithm increments k and proceeds to the next round. By so doing, we can guarantee that any repair found would contain the minimal number of tuple changes possible.

For the k-th round, the algorithm uses a close variant of that in Section 5.2.4 that is parameterized by the number of changes to input vertexes that are allowed. To do this, it picks k input vertexes from the causal network, and attempts to find a repair that only involves changing those k input vertexes (besides the changes to the causality and outcome vertexes as a result). Our algorithm achieves this by expanding the set of Preservation Constraints (*PC*) on the input vertexes that are *not* picked in this round, analogous to how it handles `preserve` and `suppress` events. For instance, if an input vertex `B(2,4)` is not picked in the current round, the *PC* would be expanded to include `B.x==2`$\wedge$`B.y==4`.

For the input vertexes that are picked in a round, e.g., `Z(0,1)`, we create free variables associated with each of its fields, i.e., `Z.x` and `Z.y`. Notice that, it is possible that we need multiple copies of `Z` tuples in order to repair the network. Therefore, our algorithm also accounts for this, by creating multiple copies of a tuple if necessary. Nevertheless, in the k-th round, the number of copies will be exactly k. For instance, in the second round, the algorithm will not only try to generate repairs by changing two tuples in `D`, `Z`, and `B`, but also two copies of the same tuple as well. If the algorithm attempts to change two copies of `Z`, it will create free variables for each copy, e.g., `Z1.x`, `Z1.y`, etc.

For instance, in the example in Figure 5.5, the algorithm would suggest a repair with two changes: a) changing `Z(0,1)` to `Z(0,y),y≠1`, which can delete `foo(1,4)` while preserving `xyz(1,4)`, and b) inserting `D(2,t),t≠3`, which can make `abc(1,2)` appear while suppressing `bar(3)`.

5.2.6 Properties

Soundness and completeness: The repaired network state is always sound, because for each tuple τ that is added to the network, the algorithm will also add its dependent tuples. Likewise, the repaired network state is also complete, since the algorithm only makes direct changes to the input vertexes, and the causality and outcome vertexes are only changed indirectly.

Validity: If the algorithm outputs a repair Δ, then it always meets the diagnostic intent. This is guaranteed by the step where the algorithm collects the objective constraints. If the algorithm does find a repair, then the repair will satisfy the *OC* constraints, which in turn guarantee that the repair is valid.

Minimality: The repair Δ found in this way is always minimal, because the iterative deepening procedure will make sure that repairs of size k are all explored before moving on to explore repairs of size $k+1$.

5.2.7 Limitations

There are also several limitations of the algorithm, which we discuss below.

Handling user-defined functions: The step where we collect the derivation constraints may fail, due to the use of user-defined functions in the NDlog rules. When we have such functions, the algorithm would not be able to recognize which dependent tuples are needed in order to make a target tuple appear. In such cases, the algorithm would abort early, reporting the tuples on which it is no longer able to make progress.

Uniqueness: The repair that the algorithm finds may not be unique. Although the

```
function NetGenie(S, Ψ)
    ψ1 ∨ ψ2.. ∨ ψt ← Partition(Ψ)
    for ψi, i ∈ [1..t] do
        Ni ← CausalNetwork(ψi, S)
        Δi ← FindRepair(Ni, S)
    return Δ = minsize{Δ1, Δ2 ··· , Δt}

function Immutability(τ)
    PC ← PC∪ PrsConstr(τ)
    for τ′ ∈ Children(τ) do
        Immutability(τ′)
function Partition(Ψ)
    ψ1 ∨ ψ2.. ∨ ψt ← DNFConversion(Ψ)
    return ψ1 ∨ ψ2.. ∨ ψt
```

Figure 5.6: Pseudocode of the NetGenie algorithm.

algorithm aims to find the minimal repair, there may still exist multiple repairs that have the same size. In such cases, the algorithm cannot recognize which is the "best" candidate repair, since all of them satisfy the diagnostic intent.

Wrong intent: If an operator supplies a wrong intent, then the algorithm would not be able to recognize them. The repair will still satisfy the supplied intent, but it may not achieve what the operator actually has in mind.

5.3 The NetGenie algorithm

So far, we have introduced the problem of intent-based network repair, and how we can use the concept of causal networks for this purpose. In this section, we start with describing Aladdin, a language that can describe diagnostic intents. We then present several refinements to the basic algorithm in Section 5.2, leveraging unique properties in the SDN context, arriving at the final algorithm for the NetGenie debugger.

5.3.1 The Aladdin language

First, we introduce the Aladdin language, which is used to describe an operator's diagnostic intent to NetGenie. Its syntax is presented in Figure 5.7. A *diagnostic event* can be represented by a tuple with a list of typed attributes $a_1, .., a_k$, with types $T_1, .., T_k$, respectively, as defined by the schema of its table; in our system model, this could denote a network event, such as a misrouted packet in the `Pkt` table, or a

$$
\begin{array}{rrcl}
\text{Diagnostic event} & e & ::= & \langle a_1{:}T_1, .., a_k{:}T_k\rangle \,|\, \emptyset \\
\text{Operator} & op & ::= & \wedge \,|\, \vee \,|\, \sim \\
\text{Statement} & st & ::= & e \,|\, \sim e \,|\, (e_1 \wedge e_2) \,|\, (e_1 \vee e_2) \\
\text{Diagnostic intent} & \Psi & ::= & \{st_1, st_2, .., st_n\}
\end{array}
$$

Figure 5.7: Syntax of the Aladdin language.

network state, such a flow entry in the `FlowEntry` table. As a simplified example, a packet located at switch S can be represented by a `Pkt(@S,Sip,Dip,Spt,Dpt,Pro)` tuple, where the location S and the five fields are the packet's attributes. Aladdin supports three *operators* – $\wedge$ (`and`), $\vee$ (`or`), and $\sim$ (`not`), which can be used to compose *statements* from events. The overall *diagnostic intent* is a conjunction of all statements.

5.3.2 Refinement #1: Sub-intents

In an SDN setting, there may exist multiple "versions" of network state that are all considered to be correct. A simple example of this is load-balancing policies, where certain packets can be handled by one of several switches or servers. More generally, such multiplicity can be detected by the use of disjunctions in the intent.

When an intent contains disjunctions, NetGenie can first break it into a set of "smaller" sub-intents, e.g., $\Psi' = \psi_1 \vee \psi_2 \vee \cdots \vee \psi_t$, and then constructs a causal network for each ψ_i. By so doing, the size of the causal networks can be reduced for more efficient reasoning. Since each of the sub-intents can be considered correct on its own, this partitioning does not violate the correctness of the repair. NetGenie can generate a set of repairs for each sub-intent, and return the smallest repair as the final result. This partitioning process can be achieved by a conversion of the intent into Disjunctive Normal Form (DNF) formulas, for which many algorithms are available.

5.3.3 Refinement #2: Immutable tuples

In principle, NetGenie can change any parts of the causal network, as long as the produced repair satisfies the intent. However, not all the base tuples in the network can necessarily be changed. For instance, suppose the operator would like to suppress a single DNS packet that was routed to her web server because of a corner case that was not handled correctly by several aspects of the network configuration. In this case, a naïve algorithm might suggest to simply remove the DNS packet itself (which is represented as a base tuple). However, in practice a repair cannot influence what kind of traffic arrives, it can only change how the network handles that traffic.

Therefore, our second refinement introduces the notion of immutable tuples. When generating a repair, NetGenie only considers changes to mutable tuples, not immutable ones. This is achieved by adding an additional set of "preservation constraints" in the FindRepair procedure in Figure 5.4, based on the set of tuples that are considered immutable. We expect that, in most cases, mutability will be related to the tuple type and not to individual tuple instances (e.g., configuration entries are mutable, and packets are immutable); thus, mutability would only need to be specified once for each network.

5.3.4 The final NetGenie algorithm

With those refinements above, we have arrived at the final version of the NetGenie algorithm, as shown in Figure 5.6. Overall, NetGenie takes in the current network state S and a diagnostic intent Ψ, and it outputs a list of base tuple changes Δ that represents the proposed repair. (Derived tuple changes are captured by a combination of base tuple changes.) The NetGenie algorithm proceeds as follows. First, it partitions Ψ' into a disjunction of sub-intents, each of which describes a "valid" version of the network of its own, and constructs a causal network N_i for each sub-intent ψ_i. Finally, it analyzes each N_i to find Δ_i – a set of changes to the mutable tuples in the network state, such that ψ_i is fulfilled in the repaired network. NetGe-

nie then returns the Δ_i with the fewest number of changes as the "best" candidate repair. Finally, NetGenie injects the generated repair to a clone of the network state, and validates that the diagnostic goals have been met.

5.4 Evaluation

In this section, we evaluate NetGenie with four diagnostic scenarios in the context of SDNs. Our evaluation aims to answer three high-level questions: a) how well can NetGenie generate network repairs that account for an operator's diagnostic intent? b) does NetGenie incur a reasonable overhead at runtime? and c) how fast can NetGenie generate network repairs?

5.4.1 Prototype implementation

We have implemented a prototype of the NetGenie debugger with 8752 lines of code in C++. Our prototype has the following four components:

Front-end: Our front-end accepts Aladdin intents, performs syntax checks, and sub-intent partitions; it then invokes the back-end debugger to generate network repairs. The front-end also accepts program written in NetCore (part of Pyretic [131]), and translates NetCore programs into NDlog rules and tuples using a similar technique from Y! [169].

Logging and snapshot engines: Our logging and snapshot engines are used to reconstruct a historical state of the network upon diagnostic queries. At runtime, the logging engine only writes down *external* inputs to the network in a log, such as packets arriving at an ingress switch. Therefore, it only needs to keep a log for each ingress switch, not for the internal switches; any internal events are later reconstructed by the replay engine. The snapshot engine takes periodic snapshots of the state of the network, so that when it is needed to reconstruct a historical state, we only need to replay a segment of the log starting from a certain snapshot, not in its entirety.

Replay engine: Our replay engine can take in a historical snapshot of the network's state, a segment of the log, and reconstruct the network state using deterministic replay; it is also used to test out a repair by applying it to a network state, and detecting whether the diagnostic goals have been met. (We note that, however, the repairs are not directly applied to the running system, but only a clone of its state.) Since the replay engine is only activated when there is a need to answer diagnostic queries, it does not incur any runtime overhead. It can also selectively reconstruct *part* of the network state to optimize for speed [180].

NetGenie reasoning engine: The NetGenie reasoning engine implements the main algorithm that we described in Sections 5.2 and 5.3, and it has an interface to the Z3 solver [56] for constraint solving. This engine accepts a reconstructed network state from the replay engine, and performs causal network analysis to generate repairs.

5.4.2 Experimental setup

We conducted our experiments in RapidNet `v0.3` [18], a declarative networking engine, on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM and a 128 GB SSD. The OS was Ubuntu 13.04, and the kernel version was 3.8.0. We set up the diagnostic scenarios using a similar OpenFlow model as in Y! [169]. To create realistic background traffic, we replayed a CAIDA packet trace collected from an OC-192 link [7] through our network, in addition to the test traffic we generated in each scenario.

5.4.3 Diagnostic scenarios

We have adapted four diagnostic scenarios in the context of SDNs for our experiments, based on an extended network topology of that in Figure 5.1. Below, we present the (slightly abbreviated) diagnostic intents, and note that these intents cannot be expressed to existing debuggers because they contain multiple goals.

- **SDN1: Load-balancing policies:** Certain requests were received by server 1

Scenario	e_1 only	e_2 only	e_3 only	Naïve	**NetGenie**
SDN1	12/0	5/1	5/1	25%	1/1
SDN2	12/0	5/1	N/A	25%	1/1
SDN3	19/1	19/1	N/A	11%	1/1
SDN4	19/0	4/3	7/2	23%	1/1

Table 5.1: Repairs generated based on individual intents rarely satisfy the operator's overall intent, whereas NetGenie can generate effective repairs for all of our scenarios. An X/Y entry means that X repairs were generated for the intent in that column, and Y of them satisfied the entire intent. Scenarios SDN2 and SDN3 only contain two individual intents. The 'Naïve' column shows the probability that a random combination of repairs for e_1–e_3 individually will satisfy the overall intent.

(e_1=P@S_1), but they should have been processed by either of the load-balanced servers 2 (e_2=P@S_2) or 3 (e_3=P@S_3). An operator writes the corresponding intent as $\Psi = \{\sim P@S_1 \wedge (P@S_2 \vee P@S_3)\}$.

- **SDN2: Misrouted packets:** A subset of HTTP traffic was misrouted to a DNS server S_1 (e_1=P@S_1) instead of the intended HTTP server (e_2=P@S_2). To address this problem, an operator specifies the intent $\Psi = \{\sim P@S_1 \wedge P@S_2\}$.

- **SDN3: Duplicate packets**: Certain traffic was received by server 1 (e_1=P@S_1) and server 2 (e_2=P@S_2), but it should have been processed by one server only, e.g., because of a load-balancing policy. Then, an operator writes the intent as $\Psi = \{(P@S_1 \vee P@S_2) \wedge \sim(P@S_1 \wedge P@S_2)\}$.

- **SDN4: Traffic scrubbing:** Traffic from untrusted origins should have been processed by a traffic scrubber (e_1=P@S_1), and handled by a server in DMZ (e_2=P@S_2), but it was routed to the internal server as a mistake (e_3=P@S_3). The diagnostic intent is written as $\Psi = \{P@S_1 \wedge P@S_2 \wedge \sim P@S_3\}$.

5.4.4 Effectiveness

First, we performed a simple sanity check that is designed to test whether NetGenie can indeed find effective repairs. To test this, we formulated the intents from

our four scenarios in Aladdin, and we ran them through NetGenie; we counted the number of repairs that were generated, and we tested whether the repairs did indeed fix the problem that existed in that scenario. Our results are shown in the last column of Table 5.1: as expected, NetGenie found a working repair for each scenario. For instance, in SDN1, NetGenie constructed a causal network with 94 vertexes, including 21 input vertexes; from this complex network, NetGenie then pinpointed the flow entry for 10.0.0.0/24 at S2, and changed the next-hop field from port 1 to port 2. This is the repair that a human operator would probably expect as well.

Next, we tested how much NetGenie benefits from having entire diagnostic intents, rather than individual atomic intents (like the existing solutions). There are two concerns here. First, it might be the case that the repairs for the individual intents are already "good enough" in many cases, in the sense that they often already satisfy the entire intent, even if it is not specified. Second, it might be the case that the repairs for the individual intents rarely cause any "collateral damage" or are incompatible; in this case, one could simply run an existing repair generator on all the `add` and `add` intents separately, and then combine the resulting repairs. (Existing solutions do not support `preserve` or `suppress` intents.)

To answer this question, we performed additional NetGenie runs using each of the individual `delete` or `add` intents e_i from each scenario. For instance, for SDN1, we ran NetGenie three times: once to remove only `Pkt(@Srv1,10.0.0.1)` (e_1), once to create only `Pkt(@Srv2,10.0.0.1)` (e_2), and once to create only `Pkt(@Srv3,10.0.0.1)` (e_3). We counted 1) how many repairs were generated for each individual intent e_i; and 2) how many of these repairs happened to satisfy the overall intent Ψ. Additionally, we generated all the naïve combinations of the resulting repairs (for each scenario, a full cross product of the two or three sets of repairs) and we counted how many of these combinations satisfied the overall intent Ψ.

Our results are shown in Table 5.1. The first three columns contain entries X/Y,

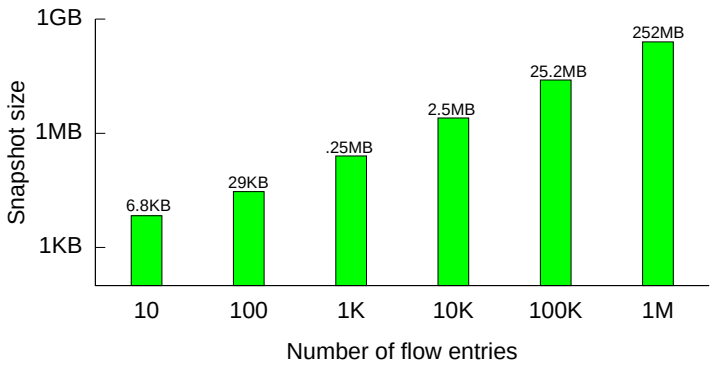


Figure 5.8: The size of snapshots grows (mostly) linearly with the number of flow entries in the network.

where X is the total number of repairs generated for e_i, and Y is the number of these repairs that happened to satisfy Ψ; the fourth column shows the fraction of the cross product that satisfies Ψ. It is clear that the answer to the first concern is no: in the overwhelming majority of the cases, solving an individual problem e_i did not serendipitously solve the other problems as well.

As the second-to-last column shows, the answer to the second concern is also no: in many cases, the repairs for the individual problems are not compatible with the repairs for the other problems. To use SDN1 as a concrete example, one repair that was generated for e_1 deleted the flow entry for 10.0.0.0/24 at S1; this would stop such traffic from being routed to server 1, but neither server 2 or 3 could receive those packets either. Another repair for this scenario simply disconnects server 1. Repairs generated for e_2 and e_3 include flooding packets from 10.0.0.0/24 at S2 to all ports, or installing a flow entry at S2 that forwards all packets to S3. This is not necessarily surprising (the repair generator simply does not know about the operator's other goals and can therefore satisfy them only by accident); at the same time, it clearly shows the benefits that NetGenie's richer interface can provide.

5.4.5 Runtime overhead

Next, we evaluated the runtime overhead of NetGenie in terms of latency and storage overheads, and how the sizes of the snapshots scale with the network size.

Latency: To evaluate the latency overhead of NetGenie, we conducted two sets of experiments. In the first set of experiments, we streamed the CAIDA trace through the network, and measured the average latency for processing one packet without logging. In the second set of experiments, we measured the same metric with the logging engine enabled. We found that logging increased the latency by 3.2%, which seems reasonable. We also note that further optimizations are feasible, which can bring down the latency even further [169].

Storage: Next, we evaluate the storage overhead incurred by logging. For each incoming packet, NetGenie logged its header, timestamp, the ingress switch ID, and several other types of metadata; on average, each packet consumed about 136 B disk space. Therefore, for a 10 Gbps switch with 500 B average packet size, the growth of the log is about 272 MB/s, without any compression or selective sampling. Moreover, it has been shown that packet capture is a common requirement in large-scale deployments, and that compression and sampling can help reduce the amount of storage growth [87], and enable packet logging in data centers on a Tbps scale [184].

Scalability: Next, we measure how the snapshot scales with the complexity of the network. Since the size of the snapshot depends on the number of flow entries in the network, we varied the number of flow entries configured in the network from 10 to 1 million, and plotted the growth of the snapshot size in Figure 5.8. As we can see from the figure, the growth is mostly linear; this is because NetGenie uses a constant size for each type of configurations (e.g., flow entries, server configurations, etc.) Since a typical network could contain 289k [120] to 575k [174] rules overall, NetGenie should be able to scale to that size with a reasonable amount of overhead.

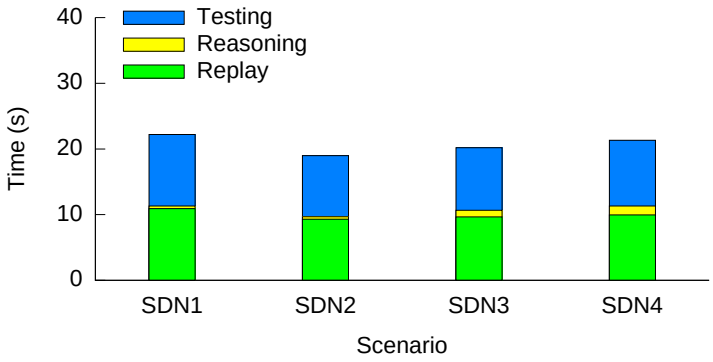


Figure 5.9: The repair generation speed for different scenarios. NetGenie returns an answer within one minute in all cases.

5.4.6 Repair generation speed

Network diagnostics typically does not require a *real-time* response; however, it is still desirable to have a reasonably short turnaround time. To evaluate this, we measured the time it took for NetGenie to generate network repairs for each scenario. Figure 5.9 shows the results. We can see that for all four cases, NetGenie finished within one minute, which seems like a reasonable amount of time for the purpose of network debugging. Moreover, most of the time has been spent in constructing the historical network state using replay, and in testing out the repair afterwards. The actual NetGenie reasoning took much less time.

Figure 5.10 shows a further decomposition of NetGenie's reasoning time. We can see that the provenance queries and the causal network analysis took the majority of the time. Once the causal network is constructed, collecting and solving the constraints took much less time. Overall, the NetGenie reasoning took less than 2.5 seconds in all scenarios.

5.5 Related Work

Our work is related to provenance in its use of causality, but it generalizes the concept of provenance to causal networks, which can keep track of causal connections and

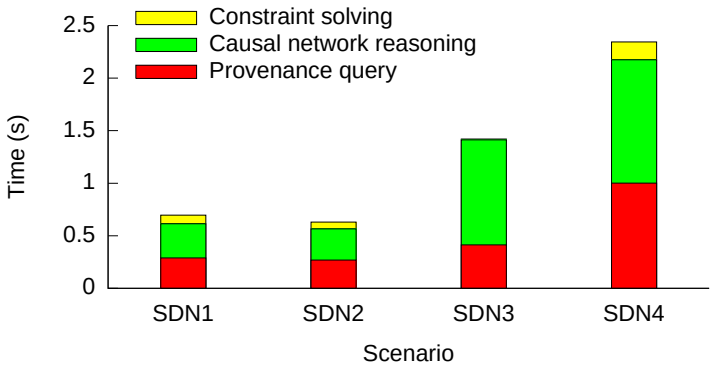


Figure 5.10: A decomposition of NetGenie's reasoning latency.

inter-dependencies among many events. For a more detailed discussion on related work on provenance, please refer to Chapter 2. Below, we discuss other related work of NetGenie.

Network diagnostics: Many debuggers have been proposed for diagnosing distributed systems. Examples include `ndb` [86], NetSight [87], X-Trace [66], CherryPick [161], and SDN traceroute [28], which can produce a "backtrace" for a given event. This is similar in spirit with provenance-based debuggers, such as [43, 169, 132, 183, 69, 180, 57]. However, all of the above solutions are purely diagnostic; they cannot generate repairs. [168] and [50] can produce repairs, but only for a single buggy event. Debugging can also be done by dynamic testing, as in OFRewind [171], ATPG [174], DEMi [150], and MCS [151], or by statistical learning, as in NetMedic [97] or NetPoirot [39]. These approaches do not track provenance or causality; therefore, they can narrow down potential culprits but not pinpoint root causes or generate a repair for a given diagnostic intent.

Verification and synthesis: Verification can eliminate bugs for certain types of networks, such as in Anteater [120], Header Space Analysis [101], NetPlumber [100], VeriFlow [102], Batfish [65], Libra [175], ConfigChecker [31], FlowChecker [30], Flowlog [137], NetKAT [35], Kinect [104], etc. However, verification does not obsolete network repair, as they are orthogonal problems – if the verification process finds a violation, one still needs to (manually) roll out a fix. NetGenie is also

related to network synthesis, including NetEgg [173] that synthesizes SDN policies, Condor [149] that synthesizes network topologies, [123] that synthesizes network updates and their ordering, etc. NetGenie does not synthesize a network from scratch, but a small change to an existing network. This is similar in spirit with NetGen [147], though the latter only repairs static data planes.

5.6 Conclusion

In this chapter, we have proposed NetGenie – a new kind of debugger that can generate network repairs based on an operator's *diagnostic intent*. When an operator sees a network problem, she could describe what she wishes to happen instead using the Aladdin language. NetGenie then reasons about the diagnostic intent, identifies the root causes, and suggests a potential repair. In our case studies in the context of SDNs, we have found that NetGenie can generate high-quality network repairs that account for an operator's entire diagnostic intent, unlike existing repair tools that only fix a single event and may cause undesirable side-effects elsewhere in the network. Moreover, NetGenie incurs a reasonable runtime overhead, and generates network repairs within one minute.

6

Conclusion

The main goal of this dissertation is to investigate whether provenance can be a good candidate for supporting the challenging diagnostic and forensic tasks that we face today. Overall, the results in the above three chapters suggest a positive answer to this hypothesis. In this final chapter, we reflect on the lessons learned in this investigation, and we then look beyond on possible future work.

6.1 The benefits of provenance

In retrospect, provenance offers three properties that have proven to be crucial in supporting diagnostic and forensic tasks.

First, provenance turns out to be a "common core" that underlies many identified diagnostic and forensic tasks. This is because, at least at a conceptual level, provenance can be seen as a complete chronicle of everything that happened in the network; for instance, SPP can capture information about network topology, control plane configurations, as well as data plane events. Therefore, if a diagnostic or

forensic question can be answered by querying such a "chronicle", then provenance would be a natural candidate for providing such support.

Second, provenance comes with a general data model, so users can easily customize the provenance model for different applications. This is evidenced by the variety of case studies we have applied provenance to: SPP applies provenance to the Internet's data plane, NetGenie focuses on provenance in SDNs, and DiffProv uses provenance in both SDNs and Hadoop MapReduce. This versatility has an important implication – research advances in provenance can potentially benefit a wide range of use cases or applications. In fact, DiffProv is an exact case in point – the same differential provenance algorithm has proven to be helpful in debugging very different types of distributed systems.

Last but not least, provenance can help weed out irrelevant factors in diagnosis, which reduces the potential search space as a result. This is because provenance captures causality, not just incidental correlations. NetGenie, for instance, uses causal networks as a close guide to generate *targeted* repairs. Such causality information also offers a second benefit: it enables a kind of "what-if" analysis to reason about potential outcomes of configuration changes – a property that both DiffProv and NetGenie have leveraged to avoid "guesswork", which is often a major source of false positives and false negatives.

6.2 Limitations and future work

At the same time, provenance is not without its limitations. First, capturing full, fine-grained provenance about an entire system can be expensive, as it can require significant storage space. Therefore, in practice, provenance systems may need to expire old data, or selectively capture a subset of events and states. For instance, SPP does not capture provenance on the application layer, and it may need to gradually summarize old data into coarser levels of granularity. As a result, if later, operators discover that a particular diagnostic task requires analyzing provenance data that

already expired or were not captured at all, such a task would be difficult to perform.

Therefore, reducing the storage overhead of provenance systems is an interesting research direction. For instance, can we build provenance systems that *automatically* recognize what "level of detail" is necessary for a particular set of diagnostic and forensic tasks, or even systems that can dynamically tune the provenance model at runtime? Such systems would make it safe to expire certain data without jeopardizing the ability to answer provenance queries later on. Or, would it be possible to develop powerful compression techniques on provenance data, so that data deletion or expiration would not even be necessary?

The second limitation arises when applying provenance to legacy applications. When the applications' source code is available, we would need to manually instrument them to extract provenance information. For blackbox applications where only binary executables are available, we would have to rely on external specifications or models of an application's expected behavior for provenance extraction. Both approaches can be tedious and error-prone; if the instrumentations or external models are incorrect or incomplete, then this would affect the quality of the captured provenance.

Therefore, an interesting question to consider is how we can automate provenance extraction for legacy applications. Here, two techniques could help. First, static analysis could be a good candidate for automatically recognizing where in the program to instrument and how, relieving programmers of such a burden. Second, dynamic tracing may also be helpful – for instance, recent processors have built-in hardware support for instruction-level program tracing with minimum performance overhead (such as Intel PT [12] and ARM CoreSight [5]). Such features can be a fine-grained provenance data source even for blackbox applications.

Third, existing provenance systems assume that the needed provenance data is always available in its entirety. This is a reasonable assumption for many diagnostic scenarios – for instance, when diagnosing problems that happen in an enterprise

network. However, there are also cases where (part of) the provenance data may be private. In SPP, we have already seen a preliminary version of this problem, where certain provenance vertexes are only visible to users with privileged views, but not others. Another example scenario would be multi-tenant data centers – if a tenant's VM has a misconfiguration, this may affect the normal operations of the underlying physical infrastructure (e.g., causing congestion on certain links); or, on the other hand, if the infrastructure misconfigures a tenant's access-control list rules, the tenant's VMs may fail to receive traffic. Both scenarios require performing diagnostics with incomplete provenance data.

To address this, it would be interesting to consider whether we can develop privacy-preserving diagnostics with provenance. The first step for this would be to find the right privacy model, answering questions such as "which part of the provenance data should be considered private?". The second step would be to design privacy-preserving algorithms on provenance data, where secure multiparty computation or other security protocols may be important building blocks. Last but not least, it would also be interesting to explore the theoretical limitations posed by privacy requirements. Since the diagnostic tool only has access to limited data, this would lead to an inherent limitation as to which types of problems are "diagnosable".

Finally, existing provenance systems still adopt a human-in-the-loop approach to diagnostics. For instance, DiffProv requires an operator to provide both a faulty symptom as well as a reference event, and NetGenie requires an operator to write down a diagnostic intent. Involving human operators in the loop has two potential downsides: a) the turnaround time for diagnostics is limited by how fast human operators can identify and respond to network problems, and b) the effectiveness of the diagnostics depends on the quality of the inputs from operators.

Looking beyond, one interesting approach would be to take human operators completely out of the loop. Ideally, there would be a *fault detection* component in the system that performs real-time problem detection. Upon fault detection, a *fault*

diagnosis component would analyze the symptom and identify the root cause of the detected problem. Finally, a *fault recovery* component would generate repairs and inject them to the system to rectify the problem. Here, several interesting questions arise: a) how can we borrow from the formal methods community and develop specifications or invariants of "correct" behaviors of a system?; b) how should we assess the degree of success of fault recovery?; and even c) can we use program synthesis techniques to synthesize networks that are correct by construction? If we are able to address some or all of the above challenges, it could substantially enhance the reliability and security of future distributed systems.

Bibliography

[1] 5-minute outage costs Google $545,000 in revenue. https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/. Accessed: May 2017. [Cited on page 2.]

[2] Amazon and the $150 million typo. http://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo. Accessed: May 2017. [Cited on page 2.]

[3] Amazon mystery solved: A typo took down a big chunk of the Internet. https://www.usatoday.com/story/tech/news/2017/03/02/mystery-solved-typo-took-down-big-chunk-web-tuesday/98645754/. Accessed: May 2017. [Cited on page 2.]

[4] Ansible Playbooks. http://docs.ansible.com/ansible/playbooks_intro.html. Accessed: May 2017. [Cited on page 103.]

[5] ARM CoreSight. http://www.arm.com/products/system-ip/coresight-debug-trace. Accessed: June 2017. [Cited on page 127.]

[6] BGP case studies. http://bgp.us/case-studies/. Accessed: May 2017. [Cited on page 1.]

[7] CAIDA. http://www.caida.org/home/. Accessed: May 2017. [Cited on pages 84 and 117.]

[8] Chef for Junos OS. https://docs.chef.io/junos.html. Accessed: May 2017. [Cited on page 103.]

[9] Command References: BGP Commands. http://www.cisco.com/c/en/us/td/docs/ios/12_2/iproute/command/reference/fiprrp_r/1rfbgp1.html. Accessed: May 2017. [Cited on page 103.]

[10] Cwe-117: Improper output neutralization for logs. https://cwe.mitre.org/data/definitions/117.html. Accessed: May 2017. [Cited on page 2.]

[11] Google Cloud Status Dashboard. https://status.cloud.google.com/summary. Accessed: May 2017. [Cited on page 103.]

[12] Intel PT. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. Accessed: June 2017. [Cited on page 127.]

[13] Mininet. http://mininet.org/. Accessed: May 2017. [Cited on pages 82, 84, and 92.]

[14] NTT SLA. http://www.ntt.net/english/service/sla_ts.html. Accessed: 2013. [Cited on page 44.]

[15] Outages mailing list. http://wiki.outages.org/index.php/Main_Page#Outages_Mailing_Lists. Accessed: May 2017. [Cited on pages 1 and 103.]

[16] Police face £750k bill for false Operation Ore charges. http://www.telegraph.co.uk/technology/news/8422200/Police-face-750k-bill-for-false-Operation-Ore-charges.html. Accessed: May 2017. [Cited on pages 2 and 22.]

[17] Puppet 4.5 reference manual. https://docs.puppet.com/puppet/latest/reference/lang_summary.html. Accessed: May 2017. [Cited on page 103.]

[18] RapidNet. http://netdb.cis.upenn.edu/rapidnet/. Accessed: May 2017. [Cited on pages 82 and 117.]

[19] A rare peek into the massive scale of AWS. http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/. Accessed: Nov. 2014. [Cited on page 1.]

[20] Sprint SLA. https://www.sprint.net/sla_performance.php. Accessed: May 2017. [Cited on page 44.]

[21] Symantec says hackers tried extortion. http://bits.blogs.nytimes.com/2012/02/07/symantec-says-hackers-tried-extortion/. Accessed: May 2017. [Cited on page 21.]

[22] Techie lands in jail due to Airtel, sues it. http://ibnlive.in.com/news/techie-lands-in-jail-due-to-airtel- sues-it/101343-3.html. Accessed: 2009. [Cited on pages 2, 21, and 22.]

[23] The Beacon Controller. https://openflow.stanford.edu/display/Beacon/Home. Accessed: May 2017. [Cited on pages 84 and 92.]

[24] The DevOps 2.0 Toolkit. https://leanpub.com/the-devops-2-toolkit/read. Accessed: May 2017. [Cited on page 103.]

[25] The NANOG Archives. http://mailman.nanog.org/pipermail/nanog/. Accessed: May 2017. [Cited on pages 1 and 103.]

[26] What we know about Friday's massive East Coast Internet outage. https://www.wired.com/2016/10/internet-outage-ddos-dns-dyn/. Accessed: May 2017. [Cited on page 2.]

[27] M. Afanasyev, T. Kohno, J. Ma, N. Murphy, S. Savage, A. C. Snoeren, and G. M. Voelker. Privacy-preserving network forensics. *CACM*, 54(5), May 2011. [Cited on page 38.]

[28] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN traceroute: Tracing SDN forwarding without changing network behavior. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2014. [Cited on pages 10, 100, and 123.]

[29] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. [Cited on page 10.]

[30] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *ACM workshop on Assurable and Usable Security Configuration*, 2010. [Cited on pages 12 and 123.]

[31] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *IEEE International Conference on Network Protocols (ICNP)*, 2009. [Cited on pages 12 and 123.]

[32] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *ACM Symposium on Principles of Database Systems (PODS)*, 2011. [Cited on page 13.]

[33] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *ACM Symposium on Principles of Database Systems (PODS)*, 2011. [Cited on page 81.]

[34] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol (AIP). In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2008. [Cited on pages 11, 19, 20, 22, 37, 38, 55, 56, and 57.]

[35] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2014. [Cited on pages 12 and 123.]

[36] K. Argyraki, P. Maniatis, D. R. Cheriton, and S. Shenker. Providing packet obituaries. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2004. [Cited on pages 11 and 57.]

[37] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and delay accountability for the Internet. In *IEEE International Conference on Network Protocols (ICNP)*, 2007. [Cited on pages 3, 11, 18, 34, and 57.]

[38] K. Argyraki, P. Maniatis, and A. Singla. Verifiable network-performance measurements. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010. [Cited on pages 11, 22, 37, 38, 56, and 57.]

[39] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with NetPoirot. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, Aug. 2016. [Cited on pages 10 and 123.]

[40] M. Attariyan and J. Flinn. Using causality to diagnose configuration bugs. In *USENIX Annual Technical Conference (ATC)*, 2008. [Cited on page 95.]

[41] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris Traceroute. In *ACM Internet Measurement Conference (IMC)*, 2006. [Cited on pages 3, 18, 19, 22, 34, 38, 56, and 57.]

[42] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. Neill, and W. P. Marnane. FPGA implementations of the round two SHA-3 candidates. In *Second SHA-3 Candidate Conference*, 2010. [Cited on page 43.]

[43] A. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the Linux kernel. In *Proc. USENIX Security Symposium*, 2015. [Cited on pages 14, 105, and 123.]

[44] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *International Conference on Very Large Databases (VLDB)*, 2004. [Cited on page 13.]

[45] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, June 2005. [Cited on pages 61 and 66.]

[46] M. Blott, J. Ellithorpe, N. McKeown, K. Visssers, and H. Zeng. FPGA research design platform fuels network advances. *Xilinx Xcell Journal*, Fourth Quarter:24–29, 2010. [Cited on page 39.]

[47] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *The International Conference on Database Theory (ICDT)*, Jan. 2001. [Cited on pages 13, 20, 22, and 64.]

[48] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *ACM Symposium on Principles of Database Systems (PODS)*, 2002. [Cited on page 13.]

[49] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014. [Cited on page 2.]

[50] 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 *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, Aug. 2016. [Cited on pages 97, 98, 100, 102, and 123.]

[51] C. Chen, H. T. Lehri, L. K. Loh, L. Jia, B. T. Loo, W. Zhou, and A. Alur. Distributed provenance compression. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2017. [Cited on page 17.]

[52] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1(4):379–474, Apr. 2009. [Cited on page 13.]

[53] D. Clark. The design philosophy of the DARPA Internet protocols. *ACM SIGCOMM CCR*, 18(4):106–114, 1988. [Cited on pages 18 and 21.]

[54] N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *ACM Symposium on Principles of Database Systems (PODS)*, June 2007. [Cited on page 106.]

[55] S. Davidson, Z. Bao, and S. Roy. Hiding data and structure in workflow provenance. In *International Workshop on Databases in Networked Information Systems*, 2011. [Cited on page 13.]

[56] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Apr. 2008. [Cited on page 117.]

[57] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security Symposium*, 2011. [Cited on pages 14, 105, and 123.]

[58] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling end users to detect traffic differentiation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010. [Cited on pages 11, 22, 38, 56, and 57.]

[59] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, 9(13):280–292, 2001. [Cited on pages 22, 38, 56, and 57.]

[60] R. Durairajan, J. Sommers, and P. Barford. Controller-agnostic SDN debugging. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2014. [Cited on pages 59 and 85.]

[61] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *ACM Internet Measurement Conference (IMC)*, 2014. [Cited on page 2.]

[62] Facebook. More details on today's outage. `https://www.facebook.com/notes/facebookengineering/more-details-on-todaysoutage/431441338919`. Accessed: May 2017. [Cited on pages 2 and 98.]

[63] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. BUZZ: Testing context-dependent policies in stateul networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016. [Cited on page 10.]

[64] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, Aug. 2004. [Cited on pages 3 and 18.]

[65] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. [Cited on pages 12 and 123.]

[66] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007. [Cited on pages 10, 100, and 123.]

[67] K. Gaj, E. Homsirikamol, and M. Rogawski. Comprehensive comparison of hardware performance of fourteen round 2 SHA-3 candidates with 512-bit outputs using field programmable gate arrays. In *Second SHA-3 Candidate Conference*, 2010. [Cited on page 43.]

[68] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif. Comprehensive evaluation of high-speed and medium-speed implementations of five SHA-3 finalists using Xilinx and Altera FPGAs. https://eprint.iacr.org/2012/368.pdf. [Cited on page 42.]

[69] A. Gehani and D. Tariq. SPADE: Support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2012. [Cited on pages 14, 64, 105, and 123.]

[70] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2011. [Cited on page 1.]

[71] Google. About today's App Engine outage. http://googleappengine.blogspot.com/2012/10/about-todays-app-engine-outage.html. Accessed: May 2017. [Cited on page 98.]

[72] Google. More on today's Gmail issue. https://gmail.googleblog.com/2009/09/more-on-todays-gmail-issue.html. Accessed: May 2017. [Cited on pages 2 and 98.]

[73] C. L. Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013. [Cited on page 98.]

[74] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *ACM Symposium on Principles of Database Systems (PODS)*, 2007. [Cited on page 13.]

[75] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, and Z. G. Ives. Orchestra: Facilitating collaborative data sharing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2007. [Cited on page 13.]

[76] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *ACM SIGCOMM CCR*, 39(1):68–73, Dec. 2008. [Cited on page 21.]

[77] 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. [Cited on page 60.]

[78] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2011. [Cited on page 103.]

[79] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *USENIX Workshop on Hot Topics in Operating Systems*, 2013. [Cited on page 98.]

[80] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009. [Cited on pages 3, 18, 20, 27, and 51.]

[81] A. Haeberlen, M. Dischinger, K. P. Gummadi, and S. Saroiu. Monarch: A tool to emulate transport protocol flows over the Internet at large. In *ACM Internet Measurement Conference (IMC)*, 2006. [Cited on page 18.]

[82] 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. [Cited on pages 11, 22, 34, 38, 49, 56, and 57.]

[83] A. Haeberlen and P. Kuznetsov. The fault detection problem. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2009. [Cited on page 36.]

[84] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007. [Cited on pages 32 and 33.]

[85] 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. [Cited on pages 99, 104, and 107.]

[86] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, Aug. 2012. [Cited on pages 9, 97, 100, and 123.]

[87] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2014. [Cited on pages 9, 22, 34, 38, 56, 57, 89, 95, 97, 100, 121, and 123.]

[88] S. Hao, M. Thomas, V. Paxson, N. Feamster, C. Kreibich, C. Grier, and S. Hollenbeck. Understanding the domain registration behavior of spammers. In *ACM Internet Measurement Conference (IMC)*, 2013. [Cited on page 21.]

[89] R. Hasan, R. Sion, and M. Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *USENIX Conference on File and Storage Technologies (FAST)*, 2009. [Cited on pages 20 and 105.]

[90] X. Inc. Virtex-5 family overview. `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, Feb. 2009. [Cited on page 40.]

[91] Z. G. Ives, A. Haeberlen, T. Feng, and W. Gatterbauer. Querying provenance for ranking and recommending. In *International Workshop on Theory and Practice of Provenance (TaPP)*, 2012. [Cited on page 13.]

[92] V. Jacobson. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.gz`. [Cited on page 49.]

[93] Jeremy Schulman. DevOps for Networking? `https://puppet.com/blog/devops-for-networking`. Accessed: May 2017. [Cited on page 103.]

[94] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2017. [Cited on pages 103 and 104.]

[95] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *The Network and Distributed System Security Symposium (NDSS)*, Feb. 1999. [Cited on page 21.]

[96] Juniper Networks. Packets per second. `http://kb.juniper.net/InfoCenter/index?page=content&id=KB14737`. Accessed: May 2017. [Cited on page 41.]

[97] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, August 2009. [Cited on pages 10, 93, 95, and 123.]

[98] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. Anderson, and A. Krishnamurthy. Reverse traceroute. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010. [Cited on pages 3, 18, 19, 22, 38, 56, and 57.]

[99] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008. [Cited on pages 3 and 18.]

[100] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013. [Cited on pages 12, 95, and 123.]

[101] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012. [Cited on pages 1, 12, 59, 95, 102, 103, and 123.]

[102] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. [Cited on pages 12 and 123.]

[103] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007. [Cited on page 95.]

[104] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. [Cited on pages 12, 103, and 123.]

[105] T. H.-J. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig. Lightweight source authentication and path validation. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2014. [Cited on page 38.]

[106] Kirk Byers. Programming: An Essential Skill For Network Engineers. http://www.networkcomputing.com/data-centers/

programming-essential-skill-network-engineers/1722440870. Accessed: May 2017. [Cited on page 103.]

[107] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, 18(3):263–297, 2000. [Cited on page 39.]

[108] R. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. [Cited on page 111.]

[109] M. Kotadia. Trojan horse found responsible for child porn. ZDNet, 8/1/2003. [Cited on pages 20 and 21.]

[110] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao. Moving beyond end-to-end path information to optimize CDN performance. In *ACM Internet Measurement Conference (IMC)*, 2009. [Cited on pages 3, 18, 20, 21, 22, 38, 49, 56, and 57.]

[111] M. Liberatore, B. N. Levine, and C. Shields. Strengthening forensic investigations of child pornography on P2P networks. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010. [Cited on pages 11 and 18.]

[112] Lisa Sampson. A DevOps primer for network engineers. http://searchnetworking.techtarget.com/feature/A-DevOps-primer-for-network-engineers. Accessed: May 2017. [Cited on page 103.]

[113] X. Liu, A. Li, X. Yang, and D. Wetherall. Passport: Secure and adoptable source authentication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008. [Cited on pages 3, 11, 18, 20, 22, 37, 38, 56, and 57.]

[114] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2016. [Cited on page 57.]

[115] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: Language, execution and optimization. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006. [Cited on pages 14, 15, and 16.]

[116] 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. [Cited on pages 68, 82, and 105.]

[117] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. ElephantTrap: A low cost device for identifying large flows. In *IEEE Symposium on High-Performance Interconnects*, 2007. [Cited on page 47.]

[118] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003. [Cited on pages 3, 18, 22, 34, 38, 49, 56, and 57.]

[119] R. Mahajan, M. Zhang, L. Poole, and V. Pai. Uncovering performance differences among backbone ISPs with Netdiff. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008. [Cited on pages 3, 18, 20, 34, and 49.]

[120] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2012. [Cited on pages 12, 59, 95, 121, and 123.]

[121] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *USENIX Security Symposium*, 2002. [Cited on page 33.]

[122] S. Matsuo, M. Knezevic, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyama, and K. Ota. How can we conduct fair and consistent hardware evaluation for sha-3 candidate? In *Second SHA-3 Candidate Conference*, 2010. [Cited on page 43.]

[123] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient synthesis of network updates. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015. [Cited on pages 12 and 124.]

[124] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. In *VLDB Endowment*, 2010. [Cited on page 13.]

[125] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. Why so? or why no? functional causality for explaining query answers. In *International Workshop on Management of Uncertain Data (MUD)*, 2010. [Cited on page 13.]

[126] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 337–348, 2012. [Cited on page 13.]

[127] R. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy*, 1980. [Cited on page 30.]

[128] H. E. Michail, L. Ioannou, and A. G. Voyiatzis. Pipelined SHA-3 implementations on FPGA: Architecture and performance analysis. In *Workshop on Cryptography and Security in Computing Systems*, Jan. 2015. [Cited on page 43.]

[129] Microsoft. Summary of Windows Azure Service Disruption on Feb 29th, 2012. https://azure.microsoft.com/en-us/blog/summary-of-windows-azure-service-disruption-on-feb-29th-2012/. Accessed: May 2017. [Cited on pages 2 and 98.]

[130] A. Mizrak, S. Savage, and K. Marzullo. Detecting compromised routers via packet forwarding behavior. *Network, IEEE*, 22(2):34–39, 2008. [Cited on pages 22, 56, and 57.]

[131] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. [Cited on pages 68, 83, and 116.]

[132] 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 *USENIX Annual Technical Conference (ATC)*, 2009. [Cited on pages 14, 82, 105, and 123.]

[133] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Provenance for the cloud. In *USENIX Conference on File and Storage Technologies (FAST)*, 2010. [Cited on page 14.]

[134] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the NetFPGA platform. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2008. [Cited on page 40.]

[135] J. Naous, M. Walfish, A. Nicolosi, D. Mazières, M. Miller, and A. Seehra. Verifying and enforcing network paths with ICING. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2011. [Cited on pages 3, 11, 18, 19, 20, 22, 38, 55, 56, and 57.]

[136] D. Naylor, M. K. Mukerjee, and P. Steenkiste. Balancing accountability and privacy in the network. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2014. [Cited on page 42.]

[137] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. [Cited on pages 12 and 123.]

[138] Nick Feamster. Tomorrow's Network Operators Will Be Programmers. `http://2015.splashcon.org/event/splash2015-keynotes-nick-feamster-keynote`. Accessed: May 2017. [Cited on page 103.]

[139] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009. [Cited on page 84.]

[140] A. Panda, K. Argyraki, M. Sagiv, M. Schapira, and S. Shenker. New directions for network verification. In *Summit on Advances in Programming Languages*, 2015. [Cited on page 102.]

[141] 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 *ACM Symposium on Operating Systems Principles (SOSP)*, 2009. [Cited on page 95.]

[142] M. Piatek, T. Kohno, and A. Krishnamurthy. Challenges and directions for monitoring P2P file sharing networks. In *USENIX Workshop on Hot Topics in Security (HotSec)*, July 2008. [Cited on page 38.]

[143] A. Ramachandran, K. Bhandankar, M. B. Tariq, and N. Feamster. Packets with provenance. In *Poster, ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2008. [Cited on page 38.]

[144] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2006. [Cited on page 21.]

[145] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008. [Cited on pages 11, 22, 38, 56, and 57.]

[146] J. Ruckert, J. Blendin, and D. Hausheer. Rasp: Using OpenFlow to push overlay streams into the underlay. In *IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2013. [Cited on page 85.]

[147] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing data-plane configurations for network policies. In *ACM Symposium on SDN Research (SOSR)*, 2015. [Cited on pages 97, 98, 102, and 124.]

[148] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2000. [Cited on pages 22, 38, 56, and 57.]

[149] B. Schlinker, R. N. Mysore, S. Smith, J. C. Mogul, A. Vahdat, M. Yu, E. Katz-Bassett, and M. Rubin. Condor: Better topologies through declarative design. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, Aug. 2015. [Cited on pages 12 and 124.]

[150] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016. [Cited on pages 10, 93, 95, and 123.]

[151] 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 *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2014. [Cited on pages 10, 59, 95, and 123.]

[152] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009. [Cited on page 95.]

[153] R. Sherwood, A. Bender, and N. Spring. DisCarte: A disjunctive Internet cartographer. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2008. [Cited on pages 18, 19, 28, and 39.]

[154] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *International Conference on Computer-Aided Verification (CAV)*, 2012. [Cited on page 104.]

[155] R. Singh and A. Solar-Lezama. Synthesizing data-structure manipulations from storyboards. In *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT*

Symposium on the Foundations of Software Engineering (ESEC/FSE), 2011. [Cited on page 103.]

[156] R. Sinha, C. Papadopoulos, and J. Heidemann. Internet packet size distributions: Some observations. Technical Report ISI-TR-2007-643, USC ISI, 2007. [Cited on page 42.]

[157] 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. [Cited on pages 3, 11, 18, 22, 34, 38, 49, 52, 56, and 57.]

[158] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190, 2017. [Cited on page 39.]

[159] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. H. Katz. Listen and Whisper: Security mechanisms for BGP. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004. [Cited on pages 22, 38, 56, and 57.]

[160] R. Sylvester. IP address typo leads to a false arrest in Kansas. The Wichita Eagle, `http://www.kansas.com/mld/eagle/news/local/crime_courts/12620843.htm`. Accessed: 2005. [Cited on pages 2 and 22.]

[161] P. Tammana, R. Agarwal, and M. Lee. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *ACM Symposium on SDN Research (SOSR)*, June 2015. [Cited on pages 10, 100, and 123.]

[162] R. Teixeira and J. Rexford. A measurement framework for pin-pointing routing changes. In *ACM SIGCOMM workshop on Network Troubleshooting*, 2004. [Cited on pages 2, 3, and 18.]

[163] Trevor Parsons. Infographic: The Modern IT and Dev Ops Toolkit. `https://blog.logentries.com/2014/12/infographic-the-modern-it-and-dev-ops-toolkit/`. Accessed: May 2017. [Cited on page 103.]

[164] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift. A placement vulnerability study in multi-tenant public clouds. In *Proc. USENIX Security Symposium*, 2015. [Cited on page 2.]

[165] Y. Velner, K. Alpernas, A. Panda, A. Rabinovich, M. Sagiv, S. Shenker, and S. Shoham. Some complexity results for stateful network verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2016. [Cited on page 102.]

[166] A. Voellmy, A. Agarwal, P. Hudak, N. Feamster, S. Burnett, and J. Launchbury. Don't configure the network, program it! domain-specific programming languages for network systems. Technical Report YALEU/DCS/RR-1432, Yale University, July 2010. [Cited on page 103.]

[167] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *USENIX Symposium on Operating Systems Design and Implementation*, 2004. [Cited on pages 1, 10, 59, 93, and 95.]

[168] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2017. [Cited on pages 1, 17, 97, 98, 100, 102, and 123.]

[169] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2014. [Cited on pages 4, 5, 6, 17, 59, 64, 83, 86, 88, 90, 94, 97, 99, 100, 106, 109, 116, 117, 121, and 123.]

[170] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2014. [Cited on page 20.]

[171] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference (ATC)*, 2011. [Cited on pages 10, 59, 95, and 123.]

[172] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. [Cited on page 57.]

[173] Y. Yuan, D. Lin, R. Alur, and B. T. Loo. Scenario-based programming for SDN policies. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2015. [Cited on pages 12, 102, 103, 104, and 124.]

[174] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012. [Cited on pages 10, 81, 84, 92, 95, 103, 121, and 123.]

[175] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. [Cited on pages 12 and 123.]

[176] 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 *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014. [Cited on page 95.]

[177] X. Zhang, A. Jain, and A. Perrig. Packet-dropping adversary identification for data plane security. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2008. [Cited on page 58.]

[178] Y. Zhang, Z. M. Mao, and M. Zhang. Detecting traffic differentiation in backbone ISPs with NetPolice. In *ACM Internet Measurement Conference (IMC)*, Nov. 2009. [Cited on pages 11, 18, 22, 34, 38, 56, and 57.]

[179] Z. Zhang, Y. Zhang, Y. C. Hu, Z. M. Mao, and R. Bush. iSPY: Detecting IP prefix hijacking on my own. *IEEE/ACM Trans. Netw.*, 18(6):1815–1828, Dec. 2010. [Cited on page 18.]

[180] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011. [Cited on pages 4, 5, 17, 20, 49, 59, 82, 100, 105, 117, and 123.]

[181] W. Zhou, Q. Fei, S. Sun, T. Tao, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. NetTrails: A declarative platform for provenance maintenance and querying in distributed systems. In *Demo, ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011. [Cited on page 27.]

[182] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *International Conference on Very Large Databases (VLDB)*, Aug. 2013. [Cited on pages 4, 17, 25, 70, and 81.]

[183] 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 *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010. [Cited on pages 3, 4, 14, 16, 20, 24, 59, 63, 97, 100, 104, and 123.]

[184] 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 *ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, Aug. 2015. [Cited on pages 81, 89, 95, and 121.]

[185] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, 2014. [Cited on pages 43 and 45.]

7

Appendix

In this appendix, we include more example code for building applications on top of SPP. In Figures 7.1–7.5, we have sketched the code snippets for implementing the functionalities in Table 3.3.

```
void reverse-tracert(Packet *p, Evidence *e) {
  IP *prevHop = gatewayIP;
  Packet *p0 = p;
  do {
    query(&p, &e, prevHop);
    print(prevHop+" "+(e.time-p0.time));
  } while (prevHop != START_OF_PATH);
}
```

Figure 7.1: Code for tracing a received packet's reverse path.

```
void identify-drop(Packet *p, Evidence *e) {
  IP *nextHop = gatewayIP;
  Packet *p0 = p;
  do {
    query(&p, &e, nextHop);
  } while (e != NULL);
  print(nextHop+" dropped the packet");
}
```

Figure 7.2: Code for identifying the node that dropped a packet.

```
void attest(Packet *p, Evidence *e) {
  IP *nextHop = gatewayIP;
  Packet *p0 = p;
  query(&p, &e, nextHop);
  if (e != NULL)
     print("Packet " + p + " had been transmitted.");
  else
     print("Packet " + p + " hadn't been transmitted.");
}
```

Figure 7.3: Code for attesting to the transmission of a packet.

```
void highestdelay(Packet *p, Evidence *e) {
  IP *nextHop = gatewayIP;
  Packet *p0 = p;
  IP *start[255];
  IP *end[255];
  int numHops = 0;
  double delay[255];
  do {
    start[numHops] = nextHop;
    query(&p, &e, nextHop);
    delay[numHops] = e.time-p.time;
    end[numHops] = nextHop;
  } while (nextHop != END_OF_PATH);
  double highest = delay[0];
  int linkID = 0;
  int i;
  for (int i = 0; i < numHops; i ++) {
     if (highest < delay[i]) {
       highest = delay[i];
       linkID = i;
     }
  }
  print("Highest-delay link: " + start[i] + "->" + end[i]);
}
```

Figure 7.4: Code for identifying the link on a path with the highest delay.

```
void throughput(Packet **p, Evidence **e, int n, Link l) {
  double delaysum = 0;
  int i;
  for (i = 0; i < n; i ++) {
    IP *nextHop = gatewayIP;
    bool linkIdentified;
    Packet *p0 = p[i];
    do {
      if (l.start == nexthop) {
         linkIdenfied = true;
      else
         linkIdenfied = false;
      }
      query(&p, &e, nextHop);
      if ((l.end == nextHop) & (linkIdentified) ) {
        delaysum += e.time-p.time
      }
    } while (nextHop != END_OF_PATH);
  double volume = 0;
  int j;
  for (j = 0; j < n; j ++) {
    volume += p[j].size;
  }
  double avg = volume/delaysum;
  print("Average throughput for the link: " + avg);
}
```

Figure 7.5: Code for the average throughput of a link.