View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS)

Department of Computer & Information Science

1-1-2012

Secure Time-Aware Provenance for Distributed Systems

Wenchao Zhou
University of Pennsylvania, wenchaoz@seas.upenn.edu

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Part of the Computer Engineering Commons

Recommended Citation

Wenchao Zhou, "Secure Time-Aware Provenance for Distributed Systems", . January 2012.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-12-14.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/973
For more information, please contact repository@pobox.upenn.edu.

Secure Time-Aware Provenance for Distributed Systems

Abstract

Operators of distributed systems often find themselves needing to answer forensic questions, to perform a variety of managerial tasks including fault detection, system debugging, accountability enforcement, and attack analysis. In this dissertation, we present Secure Time-Aware Provenance (STAP), a novel approach that provides the fundamental functionality required to answer such forensic questions – the capability to “explain” the existence (or change) of a certain distributed system state at a given time in a potentially adversarial environment.

This dissertation makes the following contributions. First, we propose the STAP model, to explicitly represent time and state changes. The STAP model allows consistent and complete explanations of system state (and changes) in dynamic environments. Second, we show that it is both possible and practical to efficiently and scalably maintain and query provenance in a distributed fashion, where provenance maintenance and querying are modeled as recursive continuous queries over distributed relations. Third, we present security extensions that allow operators to reliably query provenance information in adversarial environments. Our extensions incorporate tamper-evident properties that guarantee eventual detection of compromised nodes that lie or falsely implicate correct nodes. Finally, the proposed research results in a proof-of-concept prototype, which includes a declarative query language for specifying a range of useful provenance queries, an interactive exploration tool, and a distributed provenance engine for operators to conduct analysis of their distributed systems. We discuss the applicability of this tool in several use cases, including Internet routing, overlay routing, and cloud data processing.

Disciplines

Computer Engineering

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-12-14.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/973

SECURE TIME-AWARE PROVENANCE FOR DISTRIBUTED SYSTEMS

Wenchao Zhou

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

2012

Boon Thau Loo
Assistant Professor, Computer and Information Science
Supervisor of Dissertation

Jianbo Shi
Associate Professor, Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Zachary G. Ives (Chair), Associate Professor, Computer and Information Science

Val Tanner, Professor, Computer and Information Science

Andreas Haeberlen, Assistant Professor, Computer and Information Science

Margo Seltzer, Professor, Computer Science (Harvard University)

Micah Sherr, Assistant Professor, Computer Science (Georgetown University)

SECURE TIME-AWARE PROVENANCE FOR DISTRIBUTED SYSTEMS

COPYRIGHT

Wenchao Zhou

2012

Dedicated to my late father ZHOU, Xihua (1956 - 2009)

Acknowledgments

This dissertation would not have been possible without the guidance, helps and support from my advisor, colleagues, friends and family. I apologize in advance for any omission I may have made below.

I am very fortunate to have the privilege to work with my excellent advisor Boon Thau Loo. When I applied for graduate schools in 2007, Boon just started at Penn and generously took me as his third advisee. Admitting three enthusiastic yet inexperienced applicants in the first tenure year is aggressive and inarguably risky. I am very grateful that Boon was willing to take the leap of faith and give me the opportunity to ever start my Ph.D. at Penn. In the following five years, I have been tremendously benefited from the guidance and encouragement that I received from Boon. He has been a role model for me and other fellow advisees, by perfectly illustrating focus, discipline and strive for perfection in his work. Apart from professional advices, Boon has been amazingly generous with time and attention on other topics, including presentation skills, strategies for job search and balancing between family and work. I have learned an innumerable amount from Boon, and I am deeply indebted to him for his constant and unconditional support throughout my years at Penn.

I am also grateful to my dissertation committee chair Zachary G. Ives and the other members Andreas Haeberlen, Margo Seltzer, Micah Sherr and Val Tannen. Zack, together with Boon, mentored me in my first project at Penn on provenance-based view maintenance, which is the inception of my dissertation research on provenance. Andreas and I met in his outstanding seminar course "Distributed Systems and Social Networks" in Spring 2010. My final project "Secure Network Provenance" has since then evolved into a major portion of this dissertation. Margo kindly accepted the invitation to my dissertation committee, and gave invaluable feedback on earlier drafts of the dissertation. I am especially grateful that Margo read my dissertation thoroughly and attended the defense on the first day she recovered from a surgery. Micah and I have collaborated since the ExSPAN project in 2009. His insightful ideas and warm encouragement were one significant contributing factor of the productive collaboration. I am fortunate to further continue this collaboration with him at Georgetown University. Val is a mathematician and theoretician who always has the most profound understanding of a problem. The theoretic foundation of my dissertation was greatly inspired by the discussion with him during the TaPP workshop in 2011. This dissertation would not have been the same without his comments.

My appreciation also goes to Jennifer Rexford and Jonathan Smith for their advices on my post-graduation career and their support for my job applications, and Insup Lee for chairing my WPE-II exam in 2009.

I have spent two great summer internships at Yahoo! Research in 2010 and Microsoft Research in 2011. I would like to thank my fantastic mentors Russell Sears and Adam Silberstein (at Yahoo! Research), and Badrish Chandramouli and Suman Nath (at Microsoft Research), for providing me the great opportunities at research labs and allowing me to enjoy the pleasant and fruitful summers.

During the course of my Ph.D. study, most of my research work was performed in collaboration with my fellow students, including Xiaozhou Li, Yun Mao and Tao Tao (for the ExSPAN project), Chen Chen, Limin Jia and Anduo Wang (for the FSR project), Qiong Fei, Arjun Narayan and Sandy Sun (for the SNP project), Alexander J.T. Gurney and Mingchen Zhao (for the PVR project), Ling Ding, Yang

Li and Suyog Mapara (for the TAP project), and William R. Marczak and Andrew Mao (for the A3 project). I would like to acknowledge them here, and thank them for their relentless efforts.

Also at Penn, I have had the fortune to meet many great friends who made my five years at Penn an enjoyable experience. Changbin Liu and I share the same advisor, and we came to Penn taking the same flight on the same day, and defended the dissertations on the same day (2 hours apart). Zhuowei Bao, Mengmeng Liu and I were classmates at high school, and what a pleasant reunion at Penn's databases group! I also thank my other friends Adam Aviv, Wei Di, Harjot Gill, Dong Lin, Shivkumar Muthukumar, Santosh Nagarakatte, Lu Ren, Shaohui Wang, Jianzhou Zhao, and Zhuoyao Zhang for brightening my Ph.D. life.

Outside the computer science ecosystem, Xiaolei Su from the Harvard Medical School and Weiyuan Zhou from the Wharton Business School have been longtime friends of mine ever since high school. We accompanied each other and shared memorable moments along the journey.

Finally, but most significantly, I owe a tremendous amount to my family. Yiqing Ren, my wife, supported my decision of pursuing a Ph.D. degree unconditionally. When I left for Penn in 2007, she remained in Shanghai, China. After one year of painful long-distance relationship, she moved to Philadelphia and got enrolled in the CIS masters program at Penn, so that we could stay together. The dissertation was made possible with the sacrifices she made and her continued love. My mother, Xiuyu Wu, and my father, Xihua Zhou, have been very supportive of my education all these years. It has been the single most joyful moment when I saw their pride and happiness for my achievements. Sadly, my father passed away shortly after attending my masters graduation ceremony in 2009. I deeply miss his love. I think he would have been proud to see me finishing my studies. I dedicate this dissertation to him.

ABSTRACT

SECURE TIME-AWARE PROVENANCE FOR DISTRIBUTED SYSTEMS

Wenchao Zhou

Boon Thau Loo

Operators of distributed systems often find themselves needing to answer *forensic* questions, to perform a variety of managerial tasks including fault detection, system debugging, accountability enforcement, and attack analysis. In this dissertation, we present *Secure Time-Aware Provenance* (STAP), a novel approach that provides the fundamental functionality required to answer such forensic questions – the capability to "explain" the existence (or change) of a certain distributed system state at a given time in a potentially adversarial environment.

This dissertation makes the following contributions. First, we propose the STAP model, to explicitly represent time and state changes. The STAP model allows consistent and complete explanations of system state (and changes) in dynamic environments. Second, we show that it is both possible and practical to efficiently and scalably maintain and query provenance in a distributed fashion, where provenance maintenance and querying are modeled as recursive continuous queries over distributed relations. Third, we present security extensions that allow operators to reliably query provenance information in adversarial environments. Our extensions incorporate tamper-evident properties that guarantee eventual detection of compromised nodes that lie or falsely implicate correct nodes. Finally, the proposed research results in a proof-of-concept prototype, which includes a declarative query language for specifying a range of useful provenance queries, an interactive exploration tool, and a distributed provenance engine for operators to conduct analysis of their distributed systems. We discuss the applicability of this tool in several use cases, including Internet routing, overlay routing, and cloud data processing.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed systems have rapidly evolved, from simple client/server applications in local area networks, to Internet-scale peer-to-peer networks and large-scale cloud platforms deployed on thousands of nodes across multiple administrative domains and geographical areas. Despite the ubiquity and criticality of these distributed systems, designing and deploying these systems remains challenging due to their ever-increasing scale and the complexity and unpredictability of system executions. In both the design and deployment phases, designers/administrators of distributed systems often find themselves needing to answer *forensic* questions, to understand why and how a system execution reaches a certain state. Such examples include, but are not limited to, the following scenarios:

- **Fault detection.** A system fault may manifest itself as violations of predefined invariants or properties. For instance, in interdomain routing, prefix hijacking is a violation of the invariant that all the route advertisements to the same prefix should originate from the same autonomous system (AS). Often times, such invariants and properties are defined not only on the current state but on the derivation of that state as well. Verification of these invariants and properties can be performed by compiling the answers to the corresponding forensic questions.

- **System debugging.** A detected system fault in a prototype implementation may indicate a subtle yet critical bug in the design or the implementation. The system designer would greatly benefit from knowing the execution trace that led to this unexpected state, and the ability to reconstruct it for debugging purposes.

- **Access control.** Upon receiving a request for accessing local resources, the operators may decide to grant or deny the access based on how the request was formulated – whether it was originated from an authorized user or whether an authority has endorsed the request.

- **Accountability.** In a system deployment that crosses multiple administrative domains, each of the participating parties may act to maximize its own benefit regardless the (potentially negative) impact on the global system (e.g., "hot-potato" routing in the interdomain routing systems). Forensic questions that are answered in a collective and secure fashion are useful to enforce accountability – the ability to hold the parties to be accountable to their operations and outputs to the global system.

- **Attack analysis - root cause analysis.** Distributed systems are known to be vulnerable to malicious attacks. Given the symptoms (e.g., suspicious entries in the routing table), the operators must decide their root causes (e.g., intrusion by a malicious user) before they can take appropriate actions.

- **Attack analysis - damage assessment.** On the other hand, if an attack has been discovered, the operators must then determine its effects (i.e., its damage to the whole system), such as corrupted state on other nodes, so that the system can be repaired and brought back to a correct state.

Composing answers to forensic questions is not an easy task; in fact, the answers are often coupled with a particular *combination* of behaviors, both within the network and at different hosts, which can be hard to find. The key challenge is to inspect the data flows, dependencies, and updates to distributed (networked) nodes' state — often in ways that are not predictable in advance. Existing domain-specific solutions [22, 51, 101] often work by recording some forensic data at each node, e.g., a list of past routing changes, which are then used to answer the administrator's questions on demand. However, tailoring the schema and the introspection mechanisms to each new application is cumbersome and inflexible. It would be preferable to have a *generic* solution that can be applied to arbitrary distributed systems.

1.1 The Provenance Approach

The approach that we propose in this dissertation is to construct a distributed data structure called the *provenance* that, at a high level, tracks how data flows through the system. Data provenance itself is not a new concept — it has been extensively explored by the databases and the systems community, and has proven to be a useful and practical concept. It has been successfully applied to a variety of areas, including probabilistic databases [9, 85, 105], collaborative databases [33], file systems [42, 72, 73], scientific workflow computation [10, 15, 23, 78, 100], and cloud computing [45]. It is primarily used to answer questions concerning how query or computation results are derived and which data sources they come from. The capability of learning such information is essential to answer the cause-and-effect questions, and, therefore, enables provenance to be a promising approach for forensics in distributed systems.

Backed by the provenance system, we can support a large variety of queries to answer forensic questions. For instance, system administrators may use state queries (*"Why does a certain state τ exist?"*), which explains the derivations of system state at query time, for fault detection, history queries (*"Why did τ exist at a previous time t?"*) for system debugging and accountability, dynamic queries (*"Why and how did τ (dis)appear?"*) for root cause analysis, and causal queries (*"What state on other nodes was derived from τ?"*), which explains which parts of the system have been affected, for attack analysis and system recovery.

1.2 Research Challenges

To support the full range of functionality required for enabling forensics in distributed systems, there are a number of challenges that traditional data provenance cannot answer very well. To illustrate this, we consider a realistic scenario from today's Internet routing systems (illustrated by Figure 1.1). A network operator Alice might want to investigate why her route r_1 to the destination `foo.com` changed to route r_2 five minutes ago.

Surprisingly, this query cannot be easily answered by existing provenance systems, because it requires a comprehensive solution for several challenges arisen in distributed systems: the query may ask for a state change and for state (such as route r_1) that no longer exists; a route change during query processing could result in inconsistency in the query results; furthermore, if the change was brought about by an attacker who has "hijacked" r_1, the attacker can try to cover his traces by falsifying the provenance of r_2. To support forensic queries in this scenario,

- We must be able to capture **historical** information about past states and interactions within the systems, not just about the current state. Only maintaining relationships among current state is not enough; historical provenance would require recording relationships among entries in event logs.

- We must guarantee **correct and complete** provenance results even in **tran-**

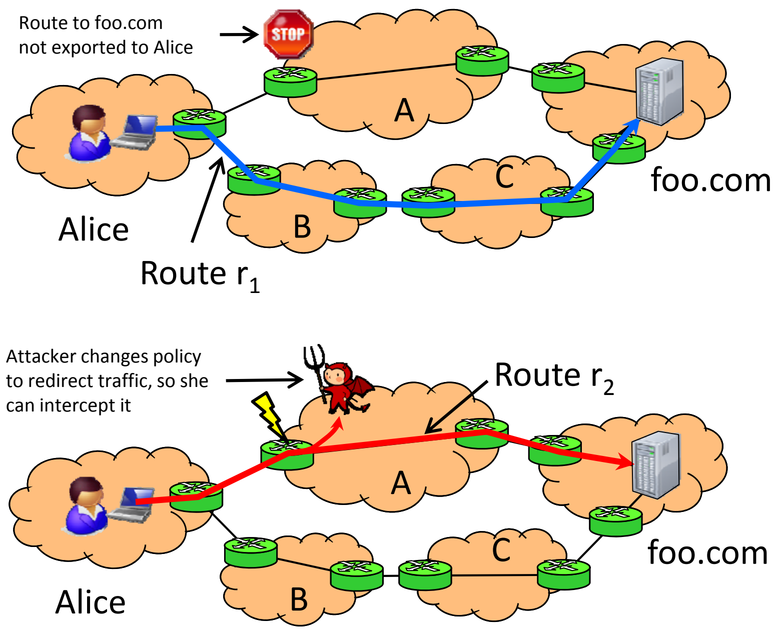


Figure 1.1: An example scenario. In the unmodified system (top), network A's policy does not allow cross traffic, so Alice can reach `foo.com` only through networks B and C. If evil Eve can compromise A's router (bottom), she can change this policy and thus gain the ability to listen to Alice's traffic.

sient state. In reality, the state of a distributed system can be highly dynamic; there can even exist instabilities or oscillations, for instance, a typical Internet router can incur hundreds of updates per minute.

- We must take **messages and timing** into account, rather than simply looking at global sequences of events. In a distributed system, the local clocks of the different nodes can be slightly out of sync, messages take time to propagate from one node to another and are sometimes lost by the network, etc.

- We must choose the right **cost tradeoff** to optimize performance. If some of the distributed nodes' operations are deterministic, provenance could be recorded at multiple levels of detail: one could record the entire record of state derivations, or instead just record the inputs and use deterministic replay to re-derive missing state.

- We must have the ability to **distribute the storage** of the provenance to keep communication costs down: for performance reasons, centrally archiving the system's entire provenance is impractical. This means that we also need the ability to **detect when nodes tamper with the provenance**; otherwise, a compromised node could cover its traces and avoid detection.

1.3 Contributions

Several prior work has proposed solutions that attack these research challenges *individually*. For instance, PASS [72] and several scientific workflow systems [10, 78, 100] present solutions for historical provenance; Orchestra [33], PA-S3fs [73] and RAMP [45] discuss distributed provenance maintenance and querying for specific applications; and Sprov [42] enforces the integrity of chain-structured provenance. (Chapter 8 summarizes and discusses the related work in greater details.) This dissertation provides the *first comprehensive* solution that addresses *all* the research challenges presented in the previous section. We demonstrate that *it is both feasible and practical to develop a general-purpose provenance system for distributed systems, that provides explanation for system behavior even in an untrusted and dynamic environment.*

This dissertation proposes and develops the foundations of *Secure Time-aware Provenance* (STAP). STAP captures time, distribution, and dependencies of updates; it enables the administrator of a distributed system to pose "ad hoc" queries over the system's prior states, communications patterns, event orderings, and more. In particular, the dissertation makes the following contributions:

1. **Time-aware provenance model.** We present a formal model and semantics of STAP [108, 112], and, using a Datalog abstraction for modeling distributed

protocols, we prove that STAP provides a sound and complete representation of the dependencies of events in a distributed system. Due to practical challenges in distributed systems (such as loosely synchronized clocks, node failures, and interactions via message passing), we rethink several of the key design decisions behind data provenance in the design of STAP.

2. **Efficient distributed provenance maintenance.** It is infeasible to maintain provenance in a centralized approach for large scale distributed systems, which may consist of thousands of nodes. To amortize the maintenance overhead, this dissertation explores a distributed storage model and the corresponding maintenance techniques [113], with the design goals that aim to minimize the impact to the existing concurrently running systems.

3. **Recursive provenance query evaluation.** As a distributed storage model is adopted, provenance queries have to be evaluated in a distributed recursive fashion. This dissertation presents a general querying framework that enables a large degree of customizations for various applications. We show that, based on a combination of recursive view maintenance and logging/deterministic replay, a recursive query can reconstruct the provenance of a system state/event that occurred at a specific time in the past [112, 113].

4. **Performance optimization and tradeoffs.** In addition, the dissertation explores a related research question – how optimizations/tradeoffs can be applied to improve the communication overhead and query response time. We present two alternative strategies for maintaining provenance: a *proactive* scheme in which provenance deltas are logged, and a *reactive* scheme in which only nondeterministic events (such as incoming messages) are logged for reconstructing provenance on demand.

   The proactive and the reactive schemes have different performance tradeoffs;

the optimal strategy for a given application depends on factors such as query frequency, system runtime, and the ratio of local vs. distributed derivations. To exploit these tradeoffs, we develop cost models [112] for both schemes, and we describe techniques for determining the optimal maintenance strategy at runtime.

5. **Security guarantees in malicious environments.** This dissertation further explores the question whether there exists an approach that can guarantee the correctness and completeness of provenance query results with the existence of compromised nodes; if not, whether we can provide a weaker yet useful guarantee. This dissertation presents a solution [109] that guarantees progress on provenance queries – given a query issued at a correct node, either a correct and complete query result is returned or at least one compromised node is exposed.

6. **Full-fledged implementation and evaluation.** Finally, we present the NETTRAILS system [111], a full-fledged TAP provenance engine. NETTRAILS uses a declarative networking [63] engine for maintaining and querying the provenance graph, which is coupled with a secure logging and replay system. NETTRAILS incrementally maintains and queries provenance in a workload-aware fashion, and it can provide meaningful guarantees despite network variability (such as instabilities or oscillations) and even in a malicious environment.

   Using three example applications (declarative network routing [63], Chord distributed hash table [99], and Hadoop MapReduce [37]) with a combination of realistic network simulations and an actual testbed deployment, we demonstrate that NETTRAILS is able to efficiently maintain and execute provenance queries at scale. Moreover, we validate that our cost model can accurately estimate the system's performance.

Chapter 2

System Model

Before we present the provenance model of STAP in Chapter 3, we first introduce a distributed system model (based on distributed Datalog) and some basic concepts that will be useful for our formal definitions. We also describe some key challenges that STAP needs to address in order to support a range of network forensic capabilities.

2.1 State Transition Systems

We consider a system that consists of a set of *nodes* $N = \{N_1, N_2, ..., N_n\}$ that are connected by a network and can communicate by sending messages. Each node has a local clock, but the clocks are not necessarily synchronized. A node N_i is either *correct*, where N_i conforms to a prescribed protocol $\mathcal{P}(N_i)$; or *faulty*, where N_i can behave arbitrarily. A faulty node externally exhibits as a Byzantine fault [55]. We write F to denote the set of faulty nodes.

We capture the behavior of a distributed system using a state transition system. We write $\mathcal{S}$ to denote the state of a distributed system. $\mathcal{S}$ consists of the set of the states of correct nodes $\mathcal{H}$ and the set of the in-fly messages $\mathcal{M}$. Events are treated as messages addressed to the local nodes. Note that, we do not consider the states of faulty nodes, as faulty nodes can behave arbitrarily regardless of their states. We now give the syntactic definition of node state and messages.

Definition 1. ***(State)*** *The state of a node N_i, $\mathcal{H}(N_i)$, is expressed as a set of* tuples *(typically with fixed schema), where each tuple $\tau \in \mathcal{H}(N_i)$ represents a relational data item that encodes information about N_i's current state.*

We model user input as tuples that are inserted or deleted directly by users, and computations performed by the system as *derivations* of new tuples from existing tuples. We say that a tuple is a *base tuple* if it was inserted directly by a user; otherwise, it is a *derived tuple*. As an illustrative example, consider a simple network routing system that computes pair-wise shortest paths among all nodes. The state of a node N_i contains the local links to its neighbor (e.g., `link(S,D,C)` for a direct link between `S` and `D` with cost `C`), and its routing entries to various destinations (e.g., `cost(S,D,C)` for a path from node `S` to `D` with cost `C`). Here, `link` tuples are the base tuples directly imported from external sources, and `cost` tuples are derived from these base `link` tuples.

Definition 2. ***(Update)*** *An update is either $+\tau$ or $-\tau$, where τ is a tuple that is being derived $(+)$ or underived $(-)$. We write $\triangle\tau$ to denote an update of either type.*

Definition 3. ***(Message)*** *Each message $m \in \mathcal{M}$ is a triplet $(src, dest, upd)$ where src denotes the sender, $dest$ denotes the receiver, and upd is the message payload. The message payload is either $+\tau$ or $-\tau$, indicating an intended update of state τ.*

We additionally write $txTime(m)$ to denote the time m is sent from src, and $rxTime(m)$ to denote the time m is received by $dest$. $txTime$ and $rxTime$ are respectively defined accordingly to the local clocks of src and $dest$.

Note that the sender src and the receiver $dest$ are not necessarily different; in the case $src = dest$, $upd = \Delta\tau$ represents a modification on the local state. Consider again the previous network routing example, `(Z, S, +cost(S,D,C))` represents a messages sent from `Z` to `S` that notifies the discovery of a new path from `S` to `D` with cost `C`, and `(S,S,-link(S,D,C))` represents a withdraw of the direct link between node `S` and `D`.

Based on the syntactic definition of system state, we next present the semantics of the state transition system. We define valid state transition as follows:

Definition 4. ***(Transition).*** *Suppose $\mathcal{S} = \{\mathcal{M}, \mathcal{H}\}$ and $\mathcal{S}' = \{\mathcal{M}', \mathcal{H}'\}$, a valid state transition $S \rightarrow S'$ corresponds to one of the following scenarios:*

1. $\mathcal{H}' = \mathcal{H}, \mathcal{M}' = \mathcal{M} + \{m\}$, *where* $m = (src, src, upd)$*: the user sends an external input (which is not yet delivered) to node* src*;*

2. $\mathcal{H}' = \mathcal{H}, \mathcal{M}' = \mathcal{M} + \{m\}$, *where* $m = (src, dest, upd), src \in F$*: a faulty node* src *sends out a message* m *to node* $dest$*;*

3. $\mathcal{H}' = \mathcal{H}, \mathcal{M}' = \mathcal{M} - \{m\}$*: the message* m *is lost during the transmission;*

4. $\mathcal{H}' = \mathcal{H}, \mathcal{M}' = \mathcal{M} - \{m\}$, *where* $m = (src, dest, upd), dest \in F$*: the message* m *is delivered to a faulty node* $dest$*;*

5. $\mathcal{H}'(dest) = \mathcal{H}(dest) + \Delta\tau$, $\mathcal{H}'(N_i) = \mathcal{H}(N_i)$ *(for* $N_i \neq dest$*),* $\mathcal{M}' = \mathcal{M} - \{m\} + \{m'_1, m'_2, ..., m'_o\}$, *where* $m = (src, dest, \Delta\tau), m'_i = (src'_i, dest'_i, \Delta\tau'_i), src'_i = dest$*: a correct node* $dest$ *receives a message* $m \in \mathcal{M}$*, applies the modification to its local state, and generates a set of new messages according to its execution logic described by protocol* $\mathcal{P}(N_i)$*.*

The first four scenarios in the definition are straightforward, where the nodes' states remain unchanged during the transition. The transition in the last scenario largely relies on the semantics of the execution logic $\mathcal{P}(N_i)$, which remains undefined. We introduce a model for execution logic and complete the definition of the state transition system in the next section.

2.2 Protocol as Derivation Rules

The execution logic of a given node N_i is entailed by the prescribed protocol $\mathcal{P}(N_i)$, which is encoded as a set of explicit *derivation rules*. This is the case, e.g., for systems that are written in a declarative language such as Network Datalog (NDlog) [63]. However, STAP is not specific to declarative systems written in NDlog, and can be applied to any legacy distributed systems whose dependencies between incoming and outgoing tuples can be modeled using derivation rules as described in Section 7.2.

As an illustrative example, we consider the simple MINCOST protocol for network routing, in which the nodes compute the lowest-cost path between each pair of nodes using the following *rules*:

```
mc1 cost(@S,D,C) :- link(@S,D,C).
mc2 cost(@S,D,C) :- link(@Z,S,C1), mincost(@Z,D,C2), C=C1+C2.
mc3 mincost(@S,D,MIN<C>) :- cost(@S,D,C).
```

As in traditional Datalog, each NDlog rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`p` should be derived whenever `q1`, `q2`, `...`, and `qn` all exist at the same time". NDlog supports a *location specifier* in each predicate, which is written as an `@` symbol followed by the node on which the tuple resides. For example, any `cost` tuples that are derived via rule `mc1` should reside on the same node as the corresponding `link` tuples, since both carry the same location specifier.

In this program, the base tuple `link(@S,D,C)` exists if node `S` has a direct link to node `D` with cost `C`. The tuple `cost(@S,D,C)` is derived when `S` has a (possibly indirect) path to `D` with total cost `C`, which can either be a direct link (`mc1`) or a path through another node `Z` (`mc2`). Rule `mc3` aggregates all paths with the same sources and destinations to compute the minimal path cost. The protocol runs continuously, and tuples can be derived or underived in response to changes to base tuples. For instance, `mincost` tuples may be updated if the cost of a link changes, since this can change the lowest-cost route.

2.3 Rule Execution Model

Next, we briefly describe the execution model for NDlog rules. The execution of a NDlog program consists of insertions and deletions of individual tuples; we refer to these as *events*. An event is essentially triggered by a node N_i receiving a message $m = (N_i', N_i, \Delta\tau)$, and applying the intended update carried by the message. We defer the formal definition of event till the end of the section, after the rule execution model is presented. Causal dependencies can exist between events; for instance, the insertion of a derived tuple causally depends on the insertion of the tuple(s) from which it was derived. It is these causal dependencies that will be captured by STAP.

NDlog programs are executed using *pipelined semi-naïve* evaluation (PSN) [63]. PSN first requires rewriting each NDlog rule into *delta rules* (also known as the event-condition-action rule in active database [20]) of the form `action :- event, conditions`. As an example, the generated delta rules for rule `mc2` in the MINCOST program are:

```
d1 +cost(@S,D,C) :- +link(@Z,S,C1), mincost(@Z,D,C2), C=C1+C2.
d2 -cost(@S,D,C) :- -link(@Z,S,C1), mincost(@Z,D,C2), C=C1+C2.
d3 +cost(@S,D,C) :- link(@Z,S,C1), +mincost(@Z,D,C2), C=C1+C2.
d4 -cost(@S,D,C) :- link(@Z,S,C1), -mincost(@Z,D,C2), C=C1+C2.
```

`d1-d2` and `d3-d4` are delta rules for the `link` and `mincost` predicates, respectively. Rules `d1` and `d3` describe insertions (+), and `d2` and `d4` describe deletions (−). For instance, in rule `d2`, `-link` is the event, `mincost` is the condition predicate, and `-cost` is the action that is taken when the event occurs and the condition holds.

For rules with aggregates (e.g. `mc3`), a similar set of insert/delete delta rules can be generated. The main difference here is that the action would result in an update of an aggregate in the rule head.

Since derivations can involve tuples on remote nodes (such as the rule `mc2` above), nodes must notify each other when they locally derive a tuple that could trigger a derivation on a remote node. This is done by sending a message that encodes the update. For instance, evaluating the delta rules of rule `mc2` results in sending message $m =$`(Z,S,+/-cost)` to node `Z`. Messages can be reordered by the network.

In PSN evaluation, when a node receives a message, it applies the update encapsulated in the message and then determines whether any additional derivations (or underivations) have been triggered by the rule evaluation. If so, the corresponding updates are formulated as messages and sent to the corresponding destinations.

We model an *event* as the application of one update and its corresponding delta rule evaluation. An event corresponds to one transition step in the system execution. We formally define an event as follows:

Definition 5. ***(Event)*** *An event $d@N_i = (e, r, t, c, e')$ represents the fact that delta rule r was triggered by update e and generated a set of updates e' at time t (relative to N_i's local clock), given the precondition c. The precondition c is a set of tuples that existed on N_i at time t that are used in the event.*

Given an initial state $\mathcal{S}$, we write $\mathcal{S} \xrightarrow{d@N_i} \mathcal{S}'$ to denote the transition from $\mathcal{S}$ to $\mathcal{S}'$, where $\mathcal{S}'$ is the resulting state after the application the event $d@N_i$.

Note that the definition of event is purposely made general to model not only the rule-based system execution (Scenario 5 in Definition 4), but the other scenarios as well. Specifically, for input externally from users and messages from faulty

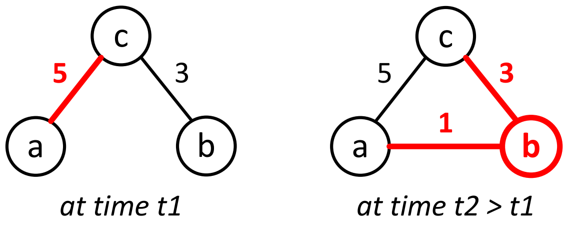


Figure 2.1: An example network, where the best path between node c and a changed at time t_2, due a change of the network topology.

nodes (Scenario 1 and 2), we write e, r and c as $\perp$; for message loss and messages delivered at faulty nodes (Scenario 3 and 4), we write r, c and e' as $\perp$.

2.4 Execution Traces

The execution of an NDlog program can be characterized by the sequence of events that take place; we refer to this sequence as an *execution trace*. An execution trace can be used to explain a derivation that occurred during the execution – we can simply replay it and check which event triggered the derivation and which conditions held at that time. A full trace can recursively explain all derivations; if we are only interested in some specific derivations (e.g., the ones queried by the network operator), a subtrace is generally sufficient.

Definition 6. ***(Trace)****: A trace $\mathcal{E}$ of a system execution is an ordered sequence of events $d_1@N_{i_1}, d_2@N_{i_2}, ..., d_m@N_{i_m}$.*

Definition 7. ***(Subtrace)****: A* subtrace *$\mathcal{E}' \subseteq \mathcal{E}$ of a trace $\mathcal{E}$ is a subsequence of $\mathcal{E}$, i.e., $\mathcal{E}'$ consists of a subset of the events in $\mathcal{E}$ in the same order. In particular, we write $\mathcal{E}|N_i$ to denote the subtrace that consists of all the events on N_i in $\mathcal{E}$.*

Figure 2.1 shows an example scenario during the execution of the MINCOST program. At some past time t_2, the network protocol changed its min-cost path

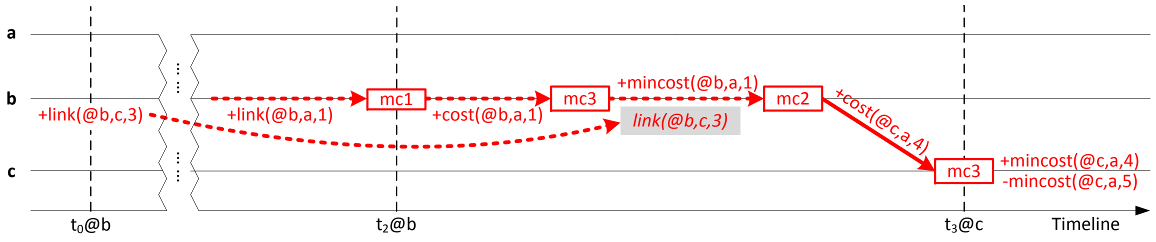


Figure 2.2: An execution subtrace of the MINCOST program that corresponds to scenario in Figure 2.1 and provides an explanation of +mincost(@c,a,4). Rectangles indicate that a rule is fired, dashed arrows indicate local event triggering, solid arrows indicate cross-node messages, and shaded boxes indicate the conditions for events.

between node `c` and `a` in response to updated link information that claimed there existed a shorter path between the two nodes. Figure 2.2 shows a part of the corresponding execution during which `+mincost(c,a,4)` is derived. The explanation for this event consists of the following trace (event tuples are denoted in **bold**):

- At time $t_2@b$, node `b` discovered a new link to node `a` and thus inserted the base tuple **`+link(@b,a,1)`**.
- Rule `mc1` was triggered by `+link(@b,a,1)`, resulting in **`+cost(@b,a,1)`**.
- Rule `mc3` was used to derive **`+mincost(@b,a,1)`** from `+cost(@b,a,1)`.
- Rule `mc2` (specifically its delta rule `d3`) was triggered by `+mincost(@b,c,1)`. The condition was satisfied by the existing tuple `link(@b,c,3)` that had been derived at time t_0; the resulting update **`+cost(@c,a,4)`** was then shipped to node `c`.
- At time $t_3@c$, node `c` received `+cost(@c,a,4)` from node `b` and derived **`+mincost(@c,a,4)`** using rule `mc3`, which then replaced the higher-cost **`mincost(@c,a,5)`**.

Note that the ordering of edges (arrows) in Figure 2.2 reflects dependencies, in the form of a *happens-before* relationship. For example, `+link(@b,c,1)` happens before `+cost(@b,a,1)` as a result of executing rule `mc1`.

2.5 Challenges and Requirements

The aim of STAP is to provide an explanation for the derivation of any network state. For example, in Figure 2.2, a network operator may issue a query asking for the explanation of `-mincost(@c,a,5)` at a particular time t_3 at node `c`. STAP's explanations should provide the entire chain of events, leading from `+link(@b,c,3)` at time t_3. To illustrate why this is a challenging problem, we consider the following realities in any distributed systems:

- **Continuous processing.** Distributed systems run continuously: nodes constantly process new information and update their state in response to local events and incoming messages from other nodes. Thus, a given tuple might have existed at time t_x but not at time t_y, or it might have existed at both times, but for different reasons. STAP should store enough information to return the correct explanation for a given time.

- **Updates.** Sometimes it is important to understand not only why a certain tuple exists, but also why it has appeared or changed. For example, to understand the route update presented in Figure 2.1, one would not only need to understand the derivation of the latest route, but also explain why the previous route was replaced by the current one. Prior distributed provenance engines [113] are unable to deal with recording explanations that evolve over time, let alone provide an explanation that causally links `-mincost(@c,a,5)` and `+mincost(c,a,4)` at time t_3.

- **Lack of synchrony.** There is no “global” time that could be used to order events. For instance, when `c` received the message in Figure 2.2, its local clock might show an earlier time than `b`’s clock when it sent the message! Also, since information takes time to propagate from node to node, there may not be a single, globally consistent explanation: if a tuple is obtained through a long chain of derivations from tuples on other nodes, some of the underlying tuples may have already changed or disappeared. Hence, STAP must capture time and dependencies at a logical level, based on rule execution and tuple instances.

- **Network effects.** Messages can be delayed and reordered. For instance, if `link(a,b,1)` is added and withdrawn within a short period of time, `mincost(@c,a,4)` would subsequently also be derived and deleted in quick succession, increasing the likelihood that the insert and subsequent delete messages are reordered in the network. The explanations of `+mincost(@c,a,4)` or `-mincost(@c,a,4)` should still be accurate in the presence of such message orderings. Message delays further complicate this, since reordering can separately happen to the actual network derivations and to the corresponding provenance metadata.

2.6 Summary

In order to present and prove correct the STAP provenance model, this chapter formally defines the system model that the provenance system will be targeting and deployed on. We model general distributed systems as state transition systems, in which the execution logic (or behavior) of each individual node is captured by a set of derivation rules. Based on the semantics of the defined state transition systems, we further present the definition of execution traces, which capture the dynamism of distributed system executions. In the subsequent section, we define the formal provenance model and show its close connection to the trace representation of system executions.

Chapter 3

Provenance Model

We use the system model described in the previous chapter as a basis for formalizing STAP. In this chapter, we present a (slightly) simplified version of STAP, called *Time-aware Provenance* (TAP), that assumes a trusted environment. We defer the discussion of STAP's security enhancement for untrusted environments to Chapter 6.

Given a distributed system, TAP is used to provide an explanation as to why a given tuple τ or update event is located on node N_i at time t. Tuple τ can be viewed as a *materialization point* that applies a sequence of the update events on τ. Intuitively, the answer for a provenance query on the existence of τ on node N_i at time t can be formulated as a sequence of query results for the update events (up to time t) on τ. Hence, we focus our discussion on the provenance of update events.

3.1 TAP Provenance Model

TAP encodes the provenance for a trace $\mathcal{E}$ in a graph $G(\mathcal{E}) = (V, E)$ in which each vertex $v \in V$ represents an event in $\mathcal{E}$, and each edge $(v_1, v_2) \in E$ represents a direct dependency between two such events. TAP's provenance graph can contain the following six types of vertices:

- INSERT$(t, \mathtt{n}, \tau)$ and DELETE$(t, \mathtt{n}, \tau)$: Tuple τ was inserted (deleted) on node $\mathtt{n}$ at time t.
- DERIVE$(t, \mathtt{n}, R, \tau)$ and UNDERIVE$(t, \mathtt{n}, R, \tau)$: Tuple τ was derived (underived) via rule R on node $\mathtt{n}$ at time t.
- SEND$(t, \mathtt{n}, \triangle\tau, \mathtt{n}')$ and RECEIVE$(t, \mathtt{n}', \triangle\tau, \mathtt{n})$: An update $\triangle\tau$ was sent (received) on node $\mathtt{n}$ at time t to (from) node $\mathtt{n}'$.

The last two vertices are needed because a derivation on one node can involve tuples on another; the corresponding messages are represented explicitly in G. The vertices are generated and connected according to the following rules:

- When a base tuple is inserted, an INSERT vertex is added.
- If a node N_i derives a tuple τ via rule r, a DERIVE vertex is added, which has incoming edges from all of r's preconditions, as well as from the triggering event, i.e., the INSERT that caused r to fire. The DERIVE vertex is then connected to a new INSERT vertex (if τ is local to N_i) or a new SEND vertex (if τ is sent to another node).
- When a message is received from another node, a RECEIVE vertex is added, with an incoming edge from the corresponding SEND vertex. This vertex is then connected to a new INSERT vertex.
- Whenever an INSERT vertex is added for a tuple τ that already has at least one derivation, an incoming edge is added to τ's most recent INSERT vertex (recall that tuples can have more than one derivation).
- When a tuple τ_1 replaces another tuple τ_2 due to a primary-key or aggregation constraint, an *update edge* is added from τ_1's INSERT vertex to τ_2's DELETE vertex.

The guidelines for deletions and underivations are analogous. Note that the graph is acyclic because edges are always added between an existing vertex and a new vertex, but never between two existing vertices. It is also monotonic because, as the execution continues, new vertices and edges are added but never removed.

Given the instantiated provenance graph $G(\mathcal{E})$, the provenance $G(\triangle\tau, \mathcal{E})$ of an update event $\triangle\tau$ on node N_i at time t is simply the subtree of $G(\mathcal{E})$ that is rooted at the corresponding INSERT(t, N_i, τ) (or DELETE(t, N_i, τ)) vertex.

When $G(\triangle\tau, \mathcal{E})$ includes a DERIVE vertex for some rule α :- $\alpha_1, \alpha_2, \ldots, \alpha_k$, it includes the provenance of *each* α_i, not just the provenance of the tuple (say, α_1) that triggered the rule. This is helpful when the provenance is used to explain the *existence* of τ, since α is a (direct or indirect) precondition for τ and each α_i is equally responsible for α's existence. However, when provenance is used to explain a state *change*, i.e., the appearance or disappearance of τ, only the provenance of the triggering tuple (here α_1) is relevant; the others merely clutter the graph. Because of this, TAP can optionally replace each subtree for a non-triggering α_i with a single EXIST vertex and a *snapshot* summarizing the current state at a particular node, as it was computed by applying all events up to the current time. State snapshots are discussed in more detail in Section 3.2.

Example: MINCOST Routing. Let us revisit our running example from the previous sections. Figure 3.1 shows a piece of the TAP graph that explains the deletion of the tuple `mincost(@c,a,5)` on node `c` at time t_3 that resulted from the new link `a-c` that was inserted at time t_0. Specifically, the edge at the DELETE vertex of `mincost(@c,a,5)` (indicated by a dotted line) corresponds to an aggregation constraint — that is, the minimal cost changed because a lower-cost path to node `a` became available. The updated lowest cost (`cost(@c,a,4)`) was derived on node `b` at time t_2 (and subsequently sent to node `c`) because a) a link `b-c` with cost three was inserted at time t_0 (and remained to exist at time t_2), and b) the tuple

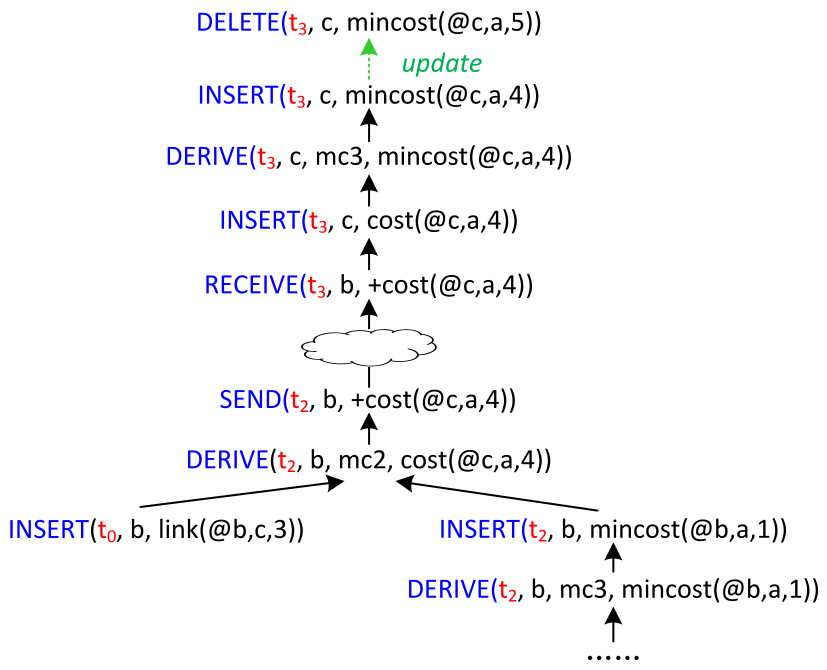

DELETE(t3, c, mincost(@c,a,5))
update
INSERT(t3, c, mincost(@c,a,4))
DERIVE(t3, c, mc3, mincost(@c,a,4))
INSERT(t3, c, cost(@c,a,4))
RECEIVE(t3, b, +cost(@c,a,4))
SEND(t2, b, +cost(@c,a,4))
DERIVE(t2, b, mc2, cost(@c,a,4))
INSERT(t0, b, link(@b,c,3))
INSERT(t2, b, mincost(@b,a,1))
DERIVE(t2, b, mc3, mincost(@b,a,1))
......

Figure 3.1: The TAP provenance graph for explaining the deletion of `mincost(@c,a,5)`.

`mincost(@b,a,1)` was newly derived at t_2 via rule `mc3`. The latter derivation was caused by the insertion of the base tuple `link(@b,a,1)`, which corresponds to the addition of the new link.

Note that the additional time dimension on the provenance graph enables another use of provenance: querying the *effects* of an update event. For example, if we want to determine how the insertion of the new link `a-b` has affected the system, we can simply locate the corresponding INSERT vertex in the graph and traverse the edges in the reverse direction.

3.2 Derivations and System Snapshots

Consider a delta rule of the form $\Delta\tau$:- $\Delta\tau_1, \tau_2, \ldots, \tau_k$. Since TAP is used to explain a state *change*, i.e., the appearance or disappearance of particular tuples, only the

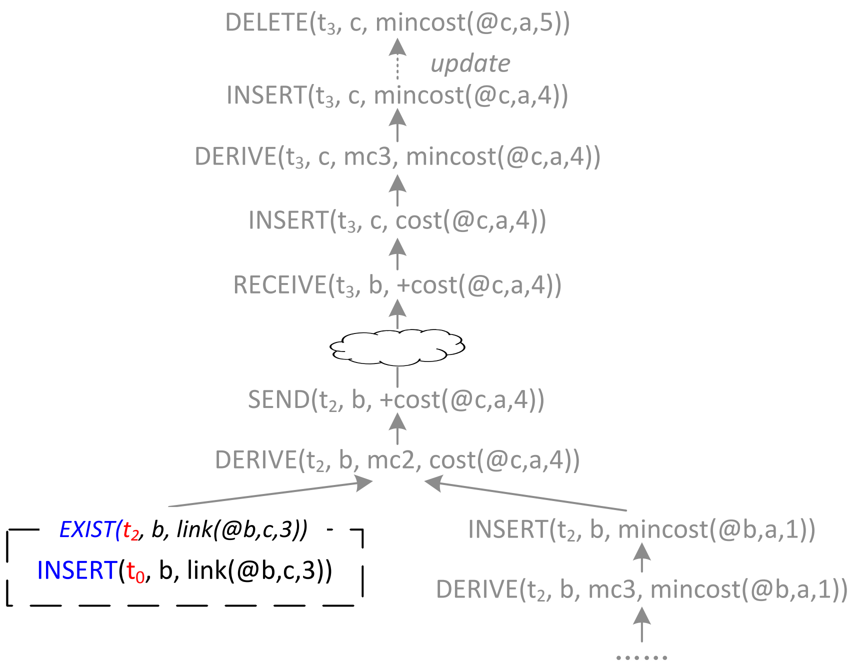

DELETE(t3, c, mincost(@c,a,5))
update
INSERT(t3, c, mincost(@c,a,4))
DERIVE(t3, c, mc3, mincost(@c,a,4))
INSERT(t3, c, cost(@c,a,4))
RECEIVE(t3, b, +cost(@c,a,4))
SEND(t2, b, +cost(@c,a,4))
DERIVE(t2, b, mc2, cost(@c,a,4))
EXIST(t2, b, link(@b,c,3))
INSERT(t0, b, link(@b,c,3))
INSERT(t2, b, mincost(@b,a,1))
DERIVE(t2, b, mc3, mincost(@b,a,1))
......

Figure 3.2: The TAP provenance graph for explaining the deletion of `mincost(@c,a,5)`. The subgraph that supports the precondition that `link(@b,c,3)` existed at time t_2 is replaced by the EXIST vertex.

provenance of the triggering tuple $\Delta\tau_1$ is relevant. Instead of storing full provenance information of all preconditions, we introduce a new vertex that provides a compact representation of per-node state at a given time:

- EXIST(t, n, τ): State of tuple τ at a particular node n at its local time t_n. This vertex includes all vertices $\{\text{INSERT}(t, n, \tau)|t \leq t_n\} \cup \{\text{DELETE}(t, n, \tau)|t \leq t_n\}$. To retrieve the snapshot value of τ at time t, one can simply replay the sequence of insertions and deletions, canceling out deleted insertions according to standard bag semantics.

In Figure 3.2, `mincost(@c,a,4)` is derived from rule `mc2`. The DERIVE vertex for `mincost(@c,a,4)` at time $t2$ includes the INSERT vertex for the triggering event `+mincost(@b,a,1)` and the EXIST vertex for `link(@b,c,3)`.

3.3 Correctness

Given the provenance $G(\triangle\tau, \mathcal{E})$ of an update event $\triangle\tau$, G should be "consistent" with the trace representation of the execution $\mathcal{E}$. We say $G(\triangle\tau, \mathcal{E})$ is correct if it is possible to extract a subtrace from G that has the properties of *validity*, *soundness*, *completeness*, and *minimality.* We first describe our subtrace extraction algorithm, followed by the correctness properties themselves.

Subtrace extraction. Given $G(\triangle\tau, \mathcal{E})$, the original subtrace can be obtained by running an algorithm $\mathcal{A}$ (presented in Algorithm 1) based on topological sort. Briefly, $\mathcal{A}$ converts each vertex in the provenance graph to an event and then uses a topological ordering to assemble the events into a trace – in other words, if two vertices $v_1, v_2 \in V$ correspond to events e_1, e_2 and are connected by edges in E, then e_1 will appear in the trace before e_2. This is possible as the provenance graph is acyclic (see Lemma 1).

In particular, Line 19 - 32 implements the construction of one individual event, where the information of a rule evaluation (such as the triggering event, conditions, and action) is extracted from the corresponding vertices in $G(\triangle\tau, \mathcal{E})$.

Correctness of subtrace. The extracted subtrace $\mathcal{A}(\Delta\tau, \mathcal{E})$ must satisfy the following four properties (Theorems and proofs for all four properties are included in Appendix A):

Property 1. ***(Validity)*** *$\mathcal{A}(\Delta\tau, \mathcal{E})$ is valid, that is, given the initial state $\mathcal{S}_0$, for any event $d_i@N_i = (e_i, r_i, t_i, c_i, e'_i) \in \mathcal{A}(\Delta\tau, \mathcal{E})$, (a) there exists $d_j@N_j = (e_j, r_j, t_j, c_j, e'_j)$ that precedes $d_i@N_i$ in $\mathcal{A}(\Delta\tau, \mathcal{E})$, $e_i \in e'_j$, and (b) for all $\tau_k \in c_i, \tau_k \in \mathcal{S}_{i-1}$, where $\mathcal{S}_0 \xrightarrow{d_1@N_1} \mathcal{S}_1 ... \mathcal{S}_{i-2} \xrightarrow{d_{i-1}@N_{i-1}} \mathcal{S}_{i-1}$. (Recall from Definition 5 that $\mathcal{S} \xrightarrow{d_k@N_k} \mathcal{S}'$ indicates a state transition from state $\mathcal{S}$ to state $\mathcal{S}'$ by applying event $d_k@N_k$.)*

Algorithm 1 Extracting subtraces from provenance

```
proc $ExtractTrace(G = (V, E))$
// calculate the out-degree of all vertices in $G$
for $\forall\ e = (v, v') \in E$ do $degree(v) \leftarrow degree(v) + 1$
// generate the subtrace based on topological sort
$NodeToProcess = V$
while $NodeToProcess \neq \phi$ do
    select $v \in NodeToProcess$ : $degree(v) = 0$, and $\nexists v' \in NodeToProcess$ that is located
    on the same node has a larger timestamp
    $NodeToProcess.remove(v)$
    if $typeof(v) =$ INSERT or DELETE then
        $find\ e = (v', v) \in E$, $degree(v') \leftarrow degree(v') - 1$
    end if
    if $typeof(v) =$ SEND or RECV then
        $find\ e = (v', v) \in E$, $degree(v') \leftarrow degree(v') - 1$
    end if
    if $typeof(v) =$ EXIST then
        $find\ e = (v', v) \in E$, $degree(v') \leftarrow degree(v') - 1$
    end if
    if $typeof(v) =$ DERIVE or UNDERIVE then
        $find\ e = (v, v') \in E$, $output \leftarrow v'$
        for $\forall\ e = (v', v) \in E$ do
            if $typeof(v') =$ INSERT or DELETE then
                $trigger = v'$ // $v'$ is the triggering event
            else
                $condition.add(v')$ // $v'$ is a EXIST vertex for one of the conditions
            end if
            $degree(v') \leftarrow degree(v') - 1$
        end for
        $ruleName \leftarrow v.ruleName$, $time \leftarrow v.time$
        $event \leftarrow (trigger, ruleName, time, condition, output)$
        $trace.push\_front(event)$
    end if
end while
return $trace$
```

Intuitively, validity means that $\mathcal{A}(\Delta\tau, \mathcal{E})$ must correspond to a correct execution of the NDlog program whose provenance is being captured. Any event that triggers a delta rule evaluation must be generated before the rule is evaluated, and that the conditions of the rule evaluation must hold at the time of the rule evaluation.

Property 2. ***(Soundness)*** *$\mathcal{A}(\Delta\tau, \mathcal{E})$ is sound, that is, $\mathcal{A}(\Delta\tau, \mathcal{E})$ is a subtrace of some $\mathcal{E}'$ that is equivalent to $\mathcal{E}$ (written as $\mathcal{E}' \sim \mathcal{E}$). We say two traces $\mathcal{E}$ and $\mathcal{E}'$ are* equivalent, *if for all nodes N_i, $\mathcal{E}|_{N_i} = \mathcal{E}'|_{N_i}$.*

Intuitively, soundness means that $\mathcal{A}(\Delta\tau, \mathcal{E})$ must preserve the all happens-before relationships among events in the original execution trace obtained from running the NDlog program. Ideally, we would like $\mathcal{A}(\Delta\tau, \mathcal{E})$ to be a subtrace of $\mathcal{E}$, but without synchronized clocks, we cannot always order concurrent events on different nodes. However, for practical purposes $\mathcal{E}$ and $\mathcal{E}'$ are indistinguishable: each node observes the same sequence of events in the same order.

Property 3. ***(Completeness)*** *$\mathcal{A}(\Delta\tau, \mathcal{E})$ is complete, that is, it ends with the event $\Delta\tau$.*

Intuitively, completeness means that $\mathcal{A}(\Delta\tau, \mathcal{E})$ must include all events necessary to reproduce τ. Note that the validity property already requires that any event that is needed for $\Delta\tau$ be included in $\mathcal{A}(\Delta\tau, \mathcal{E})$; hence, we can simply verify the completeness property of a valid trace by checking whether it ends with $\Delta\tau$.

Property 4. ***(Minimality)*** *$\mathcal{A}(\Delta\tau, \mathcal{E})$ is minimal, that is, no valid $\mathcal{E}' \subset \mathcal{A}(\Delta\tau, \mathcal{E})$ is sound and complete.*

Intuitively, minimality means that $\mathcal{A}(\Delta\tau, \mathcal{E})$ should not contain any events that are not necessary to reproduce $\Delta\tau$. If this property were omitted, $\mathcal{A}(\Delta\tau, \mathcal{E})$ could trivially output the complete trace $\mathcal{E}$.

3.4 Summary

In this chapter, we present the TAP provenance model based on the system model defined in Chapter 2. We show that it is *always* possible to extract, from the TAP provenance, a trace that is "consistent" with the actual system execution. We then further formalize and prove the correctness of the TAP provenance model as four properties (validity, soundness, completeness and minimality). (The complete proofs are presented in Appendix A.) In the subsequent chapter, we describe the provenance maintenance and querying techniques that realize the TAP provenance in distributed systems.

Chapter 4

Maintenance and Querying

In this chapter, we explore the generic data management challenges posed by the distribution, querying, and maintenance of provenance in large-scale distributed systems. Such scale has presented a unique challenge to provenance data management: Network applications in Internet domains usually involve thousands of nodes; moreover, provenance computations are required to co-exist with existing network protocols. Bandwidth efficiency and minimal impact on convergence time are of significant importance.

4.1 Storage Model

This section defines the storage model used by NETTRAILS to store and maintain provenance in distributed systems. NETTRAILS's graph-based data model is amenable to storage using a distributed relational database, and is sufficiently general to be used as a basis for generating other provenance representations.

4.1.1 Provenance as Relational Tables

NETTRAILS stores the graph representation of provenance in a relational table in a format similar to that used in existing work [29, 31]. NETTRAILS makes use of four provenance tables – called `prov`, `ruleExec`, `send`, and `recv` – that are incrementally updated as the derivation rules that model the protocols are executed. These tables store TAP's provenance graph in a distributed fashion.

Tuple instances: The `prov` table maintains information about each tuple (including both current tuples and tuples that existed in the past) as well as the specific rule that triggered its derivation. Entry `prov(@N,VID,Time,RLoc,RID)` indicates that the tuple on node `N` with unique identifier `VID` was derived at time `Time` by a rule execution on node `RLoc` that is uniquely identified by `RID`. If `N` and `RLoc` are different, the tuple was sent from `RLoc` to `N`, and this communication is recorded in additional `recv` and `send` entries (see below). `VID` is generated based on a cryptographic hash of the contents of the tuple and the time of its derivation; similarly, `RID` is a hash of the rule identifier, node location, and `VID` of the derived tuple. For base tuples, `RID` is set to `null`, since they are not derived by any rule.

In order to correctly generate the above entries, NDlog programs undergo an automatic rewrite process to include the `RID` and `RLoc` information with each tuple derivation. This process ensures that the appropriate `prov` entry will be generated on the node to which the derivation is sent.

Rule execution instances: The `ruleExec` table maintains information about each execution of a rule (not just about each rule). Entry `ruleExec(@RLoc,RID,Rule, ExecTime,Event,CList)` indicates the execution of a `Rule` on `RLoc` at `ExecTime`, triggered by an event `Event` (i.e., a tuple that changed, appeared, or disappeared) while the preconditions in `CList` were holding.

Message transmissions: The `send` and `recv` tables maintain information about message exchanges. `send(@Sender,VID,STime,RID)` and `recv(@Receiver,VID, RTime,Sender,STime)` refer to the rule execution identified by `RID` that affected the tuple identified by `VID`; the corresponding message was sent by `Sender` at time `STime` and received at time `RTime`. Whenever a rule execution causes a message

+/-	Loc	VID	Time	RLoc	RID	Derivation
$+$	b	$VID1$	$t0$	*null*	*null*	`+link(@b,c,3)`
$+$	b	$VID2$	$t2$	*null*	*null*	`+link(@b,a,1)`
$+$	b	$VID3$	$t2$	b	$RID1$	`+mincost(@b,a,1)`
$+$	c	$VID4$	$t3$	b	$RID2$	`+cost(@c,a,4)`
$+$	c	$VID5$	$t3$	c	$RID3$	`+mincost(@c,a,4)`
$-$	c	$VID6$	$t3$	c	$RID3$	`-mincost(@c,a,5)`

Table 4.1: An example `prov` relation based on Figure 3.1. The table is horizontally partitioned across all nodes, based on the location specifier `Loc`. The last column is not stored in the table; it is included here to show the derivation that corresponds to each entry. The first column indicates an insertion (+) or a deletion (−).

+/-	RLoc	RID	Rule	ExecTime	Event	CList	Derivation
$+$	b	$RID1$	$mc3$	$t2$	$VID2$	*null*	`+mincost(@b,a,1)`
$+$	b	$RID2$	$mc2$	$t2$	$VID2$	$(VID1)$	`+cost(@c,a,4)`
$+$	c	$RID3$	$mc3$	$t3$	$VID4$	*null*	`+mincost(@c,a,4)`

Table 4.2: An example `ruleExec` relation that corresponds to the DERIVE vertices shown in Figure 3.1. The last column shows the derivation rule that was executed in each instance.

to be sent, `send` and `recv` entries are generated at the sender and receiver, respectively, and are timestamped using nodes' local clocks. To handle clock skew, the receiver stores the sender's timestamp at message transmission; this timestamp is included in each message along with the (un)derived tuple. This information is used during query processing to correctly match up `send` and `recv` entries.

Given the distributed nature of provenance storage, these tables are naturally partitioned based on their first attributes, and distributed among the nodes. For instance, `prov` entries are co-located with the tuples to which the update events were applied, and `ruleExec` entries are located on the nodes on which the rule executions were performed.

Sender	VID	STime	RID	Derivation
b	$VID4$	$t2$	$RID2$	`+cost(@c,a,4)`

Receiver	VID	RTime	Sender	STime	Derivation
c	$VID4$	$t3$	b	$t2$	`+cost(@c,a,4)`

Table 4.3: Example `send` and `recv` relations that correspond to the SEND and RECEIVE vertices in Figure 3.1.

4.1.2 Example Tables

Tables 4.1, 4.2, and 4.3 show the entries for the tables above, based on the example provenance tree shown in Figure 3.1. The vertices defined by our provenance model (Section 3.1) are encoded in the above provenance tables as follows: INSERT and DELETE vertices are respectively represented as tuple insertions (`+prov`) and deletions (`-prov`). Likewise, DERIVE and UNDERIVE are stored as `+ruleExec` and `-ruleExec`. Edges between INSERT / DERIVE and DELETE / UNDERIVE pairs are represented by the `RID` and `VID` pairings in each `prov` entry. `recv` and `send` entries correspond to the RECV and SEND vertices. For each tuple uniquely identified by its primary key, each EXIST vertex consists of all updates (i.e., `+prov` and `-prov`) ordered by their timestamps.

4.2 Provenance Maintenance with Delta Rules

The TAP graph can be captured via the evaluation of delta rules (see Section 2.3) of the form `action :- event, conditions`. In a delta rule of the form $\triangle p$ `:-` $p_1, \ldots, \triangle p_i, \ldots, p_n$, the *event* (in this case, $\triangle p_i$) is represented as an INSERT or DELETE vertex, the *conditions* (the other p_k) are represented as a sequence of INSERT (or DELETE) vertices that support the existence of p_k (EXIST vertex), and the *action* ($\triangle p$) is represented as a DERIVE or UNDERIVE vertex.

When a delta rule $\triangle p$ `:-` $p_1, \ldots, \triangle p_i, \ldots, p_n$ is fired at time t, NETTRAILS performs the following steps:

- Generate a `+ruleExec` or `-ruleExec` tuple with timestamp t to represent the rule execution, and maintain pointers to the triggering event $\triangle p_i$ and preconditions $p_1, \ldots, p_n$ (excluding p_i).
- Generate a `+prov` or `-prov` tuple with timestamp t to represent the insertion or deletion event $\triangle p$, and to maintain a pointer to the generated `+/-ruleExec` tuple.
- If the generated event $\triangle p$ needs to be sent to another node, generate a pair of `send` and `recv` tuples at the sender and the receiver, respectively, with timestamps that correspond to nodes' local clocks.
- Finally, if the generated event $\triangle p$ results in a violation of a primary-key or aggregation constraint (e.g., the newly-generated tuple displaces another), generate an additional `+prov` or `-prov` tuple to represent the deletion caused by $\triangle p$. This corresponds to the update edge from Section 4.1.

4.2.1 Rule Generation for Provenance Maintenance

To perform the provenance maintenance described in the previous section, we leverage the distributed querying processing capability of the declarative networking engine. Given *any* NDlog program, additional NDlog provenance maintenance rules are automatically generated. Algorithm 2 shows the basic approach towards generating provenance maintenance rules. The algorithm takes as input an NDlog delta rule R of the form:

$$\texttt{+/-h(@H}_1\texttt{,...,H}_k\texttt{) :- +/-t}_0\texttt{(@X,P}^0_1\texttt{,...,P}^0_{k_0}\texttt{),t}_1\texttt{(@X,P}^1_1\texttt{,...,P}^1_{k_1}\texttt{),...,}$$
$$\texttt{t}_n\texttt{(@X,P}^n_1\texttt{,...,P}^n_{k_n}\texttt{),c}_1\texttt{,...,c}_p\texttt{.}$$

Algorithm 2 Generation of Provenance Maintenance Rules

```
1: proc $ProvenanceRewrite(R)$
2: $RS \leftarrow \{\}$
3: $R = \pm h(@H_1, ..., H_k)$ :- $\pm t_0(@X, P^0_1, ..., P^0_{k_0}), t_1(@X, P^1_1, ..., P^1_{k_1}), ...,$
                          $t_n(@X, P^n_1, ..., P^n_{o_n}), c_1, ..., c_p.$
4: $RS.add(eHTemp(@RLoc, H_1, ..., H_o, RID, Rule, ExecTime, Event, CList)$ :-
       $t_0(@X, P^0_1, ..., P^0_{k_0}), t_1(@X, P^1_1, ..., P^1_{k_1}), ..., t_n(@X, P^n_1, ..., P^n_{k_n}),$
       $c_1, ..., c_p, RLoc = X, Rule = r_{id},$
       $PID_0 = f\_sha1(t_0 + X + P^0_1 + ... + P^0_{k_1}),$
       ......
       $PID_n = f\_sha1(t_n + X + P^n_1 + ... + P^n_{k_n}),$
       $Event = PID_0$
       $CList = f\_append(PID_1, ..., PID_n),$
       $RID = f\_sha1(R + RLoc + Event + CList).)$
5: $RS.add(ruleExec(@RLoc, RID, Rule, ExecTime, Event, CList)$ :-
       $eHTemp(@RLoc, H_1, ...H_o, RID, Rule, ExecTime, Event, CList).)$
6: $RS.add(eH(@H_1, ..., H_o, RID, RLoc, ExecTime)$ :-
       $eHTemp(@RLoc, H_1, ...H_o, RID, Rule, ExecTime, Event, CList).)$
7: $RS.add(\pm h(@H_1, ..., H_o)$ :- $eH(@H_1, ...H_o, RLoc, RID, RecvTime).)$
8: $RS.add(prov(@H_1, VID, RecvTime, RLoc, RID)$ :-
       $eH(@H_1, ...H_o, RLoc, RID, RecvTime),$
       $VID = f\_sha1(h + H_1 + ... + H_o).)$
9: return $RS$
```

where the rule body consists of one trigger event t_0, n predicates +/-t_1, t_2,...,t_n and p constraints (or assignments) c_1, c_2,...,c_p. The derived rule head +/-h consists of k attributes which are generated from executing the rule body. Here, we assume that all body predicates are executed at location X (since the rule has been localized). We further consider the case in which the rule head location H_1 is at a different location from X (and hence the derivation is sent across the network).

The output of running the algorithm is the rule set RS, which is used for provenance maintenance and executing the original derivation. The first rule generates the local event eHTemp (line 4) which contains all information required for creating the local ruleExec entry (line 5) corresponding to the meta-data for rule execution, sending the event message eH to the target node H_1 (line 6) to create the corresponding result tuple +/-h (line 7) and the remote prov entry (line 8). Note that the only additional attribute shipped with each message `eH` is the (`RID`, `RLoc`, `ExecTime`) fields necessary to reconstruct the provenance information.

The generation of `send` and `recv` tuples is performed by the underlying transmission module, which, upon sending and receiving the packets, simply deserializes the packets and records the related fields. Note also that upon receiving the `eH` tuple, the transmission model replaces `ExecTime` field with `RecvTime`, which indicates the time when the tuple is received. This is necessary for reconstructing the provenance information by tracing from the `prov` entry back to the corresponding `recv` entry.

Recent work by Amsterdamer et al [2] has shown that provenance models should be extended to take into account the individual values within tuples to handle aggregation in general. It is worth mentioning that Algorithm 2 can be extended to support general aggregations, where Line 4 needs to be updated to treat aggregation rules as a special case, in which the individual tuples in the relations of the rule body need to be taken into account.

4.2.2 Proactive and Reactive Maintenance

To answer provenance queries about past tuples or updates, the TAP model contains a temporal dimension. This could in principle be implemented by keeping a full copy of the provenance whenever it changes. However, this would require an enormous amount of storage, particularly for long-running distributed systems with many updates. Moreover, keeping full copies of the provenance is unnecessary because TAP provenance is *monotonic*: the provenance of historic updates and tuples (which eventually make up a major portion of a provenance graph) is immutable.

For better efficiency, NETTRAILS maintains provenance incrementally, i.e., it considers only the "deltas" between adjacent versions, which are sufficient to reconstruct the full provenance graph. NETTRAILS can store these deltas in the following two different ways:

- **Explicit deltas (proactive).** In this approach, all of the `+prov`, `-prov`, `+ruleExec` and `-ruleExec` entries are stored explicitly in a temporally ordered log that is indexed by time. Compared to keeping each version of the provenance, the storage cost is considerably lower; however, the full provenance information must be reconstructed from the deltas before a query can be answered. To permit fast reconstruction of EXIST vertices during query execution (see Section 4.3), NETTRAILS maintains reverse-time ordered pointers between all `+prov` and `-prov` entries that correspond to the same tuple. For instance, in Table 4.1, the `prov` entry with `VID4` (`+mincost(@c,a,4)`) points to the entry with `VID5` (`-mincost(@c,a,5)`) which in turn points to another entry for `+mincost(@c,a,5)`.

  The idea of keeping deltas between adjacent versions and reconstructing a specific version by merging deltas is known as a classic approach to perform efficient versioning. It has been extensively studied and adopted in many application domains, include transaction logs in database systems [30, 70], revision control systems [43, 102], and log-structure file systems [89, 93]. Several variants, such as forward deltas and reverse deltas [102], have been proposed to tailor the system performance for different system settings or requirements. We expect that similar treatment is applicable to the proactive provenance maintenance as well.

- **Per-node input logs (reactive).** In this approach, NETTRAILS maintains only the non-deterministic inputs (`recv` entries for incoming messages, as well as tuple insertions and deletions) at each node. If the underlying application is deterministic, NETTRAILS can replay these inputs at query time to reproduce the original execution of that node, and reconstruct the provenance on the

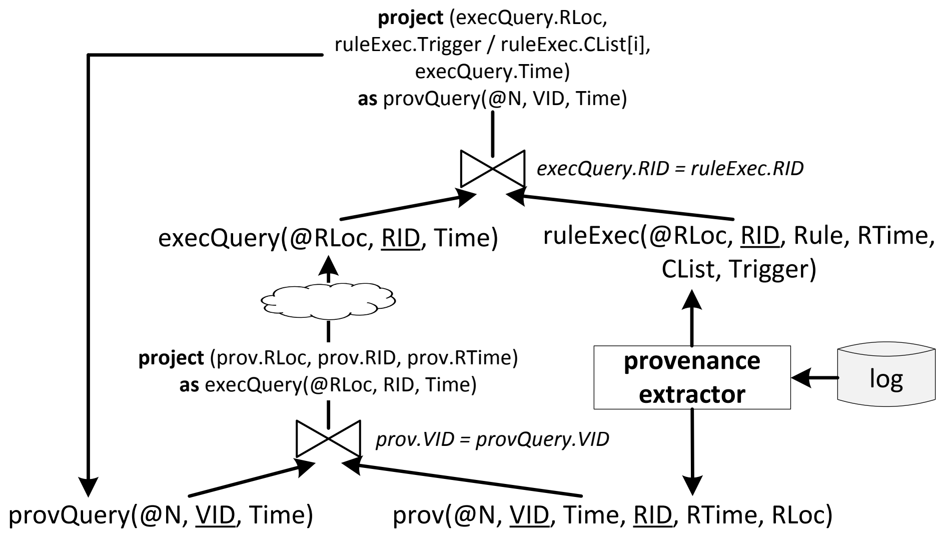

project (execQuery.RLoc, ruleExec.Trigger / ruleExec.CList[i], execQuery.Time) **as** provQuery(@N, VID, Time)
execQuery.RID = ruleExec.RID
execQuery(@RLoc, RID, Time)
ruleExec(@RLoc, RID, Rule, RTime, CList, Trigger)
project (prov.RLoc, prov.RID, prov.RTime) **as** execQuery(@RLoc, RID, Time)
provenance extractor
log
prov.VID = provQuery.VID
provQuery(@N, VID, Time)
prov(@N, VID, Time, RID, RTime, RLoc)

Figure 4.1: Logical query plan for recursive provenance queries. Underlined attributes are primary keys.

fly. As an optimization, each derived tuple sent across nodes needs only to include the sender's timestamp.

The first approach represents a *proactive* style of provenance maintenance in which provenance information is stored explicitly in the form of deltas; the second approach represents a *reactive* style in which provenance information is reconstructed at query time. There exists a tradeoff between the two: the proactive approach results in lower query latencies (since there is less overhead for reconstruction) but requires more storage space.

4.3 Provenance Query Processing

For ease of exposition, we first limit our discussion to the scenario in which the query result of interest is the entire provenance for the given update $\triangle\tau$ at time t. We then discuss in Section 4.3.4 how the query processing can be customized to handle a more diverse range of application scenarios.

4.3.1 Recursive Query Plan

To query the provenance of an update, NETTRAILS executes a distributed recursive query that reconstructs the relevant subtree of the provenance graph from the four tables we have described in Section 4.1. Figure 4.1 shows the logical query plan for evaluating this distributed recursive query; the query starts at the root of the subtree and iteratively adds vertices and edges until a fixpoint is reached (at the base tuples). The results are then returned in the form of tuples from the `prov`, `ruleExec`, `send`, and `recv` tables that encode the relevant subtree.

In Figure 4.1, the initial provenance query is represented as an input tuple `provQuery(@N,VID,Time)` to the logical plan. Based on this tuple, NETTRAILS carries out the following steps:

- **Step 1: Retrieve rule execution instances.** Since the `VID` uniquely identifies $\triangle\tau$, NETTRAILS uses it as a lookup into the `prov` table (via a database join) and then retrieves the corresponding `RID` used to derive the tuple, as well as the location `RLoc` at which the rule was fired. This corresponds to the generation of the DERIVE or UNDERIVE vertex. If `RLoc` is different from `Loc` (i.e., the tuple was derived from a remote rule execution), additional RECV and SEND vertices are generated by joining the `VIDs` of derived tuples with the `recv` and `send` tables[1]; for readability, these extra operations have been omitted from Figure 4.1. Next, NETTRAILS generates `execQuery` tuples to trigger queries on the `ruleExec` table.

- **Step 2: Expand dependent derivations.** NETTRAILS ships the resulting `execQuery(@RLoc,RID,Time)` tuple to `RLoc` and there joins it with the local `ruleExec` table to recursively expand the child derivations that have resulted

[1]After retrieving the `recv` entry based on `VID` and `RTime`, we use the `STime` (sender's timestamp) attribute in `recv` to fetch the appropriate `send` entry on the sender's side. This avoids explicit time synchronization.

in $\triangle_{\mathcal{T}}$. Here, multiple additional `provQuery` tuples are generated: one for the trigger event for the delta rule `RID`, and another for each condition predicate value that occurred during the execution of `RID`. Each expansion generates an INSERT or DELETE vertex, depending on whether the trigger event was an insertion or a deletion, and each expanded condition generates an EXIST vertex, which includes additional INSERT and DELETE vertices to explain why the condition held at the relevant `Time`.

- **Repeat until fixpoint.** Steps 1 and 2 are performed recursively until all child nodes are expanded. As the query progresses, the `provQuery` events are recursively propagated from the root of the provenance tree (where the queried update resides) towards the child nodes in order to construct the entire subtree. Each level of the tree can be expanded in parallel at different nodes. Upon reaching the leaf nodes (which correspond to base tuples), the query results are returned back to the root along the reverse path. At each level, the parent node returns only its portion of the query result (subtree) after all the child nodes have completed their respective subqueries.

For our prototype, we have implemented the query plan from Figure 4.1 in NDlog, and we execute it on a distributed recursive query engine [63]. To customize the query and to return other annotations of provenance [48], NETTRAILS supports user-defined functions for augmenting the query plan.

4.3.2 Reconstructing Provenance on Demand

Steps 1 and 2 assume that the entire `prov` and `ruleExec` tables are already constructed and available when the query is issued. To support the proactive and reactive maintenance techniques from Section 4.2.2, NETTRAILS needs an additional provenance extraction operator to reconstruct the `prov` and `ruleExec` entries from deltas or input logs whenever a `provQuery` or `execQuery` is received.

Reconstruction with provenance deltas. When the log consists of deltas, the `prov` entries are reconstructed as follows. Given a provenance query `provQuery(@N,VID,Time)`, the provenance extractor is invoked using `VID` and `Time` as the lookup keys. Using a fast binary-search data structure indexed by time, the extractor searches the log for an entry corresponding to `VID` at time `Time`, and returns the corresponding `+/-prov` tuple. In some cases, if an EXIST vertex is required, the provenance extractor first finds the latest `prov` entry at time `Time` and then follows the chain of updates backwards in time to retrieve all tuples with a `VID` smaller than `Time`. An analogous mechanism is used to reconstruct the `ruleExec` tuples by searching for the corresponding `RID` and `RTime`.

Reconstruction with input logs. In the reactive implementation, instead of searching for the appropriate `prov` and `ruleExec` entries, reconstruction involves replaying the entire log (messages and changes to base tuples) at the relevant nodes until the specified input time (`Time` or `RTime`) is reached. Each `recv` entry from the log is replayed on a reference implementation of the distributed system to regenerate `prov` and `ruleExec` entries.

In theory, one can always start replaying the input logs from the very beginning of the system execution. However, this can be costly when the application has been active for a long time, particularly if the derivation rules are computationally expensive. NETTRAILS reduces this overhead by periodically recording a *checkpoint*. Each node checkpoints only its local state — specifically, the currently extant tuples and any unprocessed updates in the local pool. This is sufficient because each node replays only its local execution, and it allows NETTRAILS to avoid the complex mechanisms needed for consistent global checkpoints. The input log can then be incrementally applied, starting from the latest checkpoint.

A possible optimization is to cache `prov` and `ruleExec` entries from previous replays in case they can be used in a subsequent query. This avoids unnecessary replays. Additional methods for improving querying and maintenance performance are discussed in the next chapter.

4.3.3 Distributed Recursive Query Formulation

We next present the mechanisms by which NETTRAILS formulates distributed queries to derive various representations from the distributed provenance.

To derive provenance information, NETTRAILS utilizes NDlog programs that express distributed recursive queries. These queries traverse provenance graphs (in the form the `prov`, `ruleExec`, `send` and `recv` tables) in a distributed fashion, returning results to the querying node. NETTRAILS is flexible to permit different granularities and representations of provenance (see Section 4.3.4). The programmer may select the type of network provenance by modifying the query specifications.

The following NDlog program demonstrates a generic distributed graph traversal operation on tables `prov` and `ruleExec`. The entire program is written in twelve NDlog rules: two base rules (`edb1` and `c0`), and two pairs of five rules for recursively querying the `prov` (`idb1-idb5`) and `ruleExec` (`rv-rv5`; not shown for brevity) tables. The rules are continuous, long-running queries that are initially installed at every NETTRAILS node for handling distributed provenance queries.

```
// Base case
edb1 provResults(@Ret,QID,VID,Prov) :- provQuery(@X,VID,Time,QID,Ret),
       prov(@X,VID,Time,RLoc,RID), RID==NULL, Prov=f_pEDB(VID).

// Count number of children for each VID
c0 numChild(@X,VID,COUNT<*>) :- prov(@X,VID,Time,RLoc,RID).

// Initializing Buffer
idb1 pResultTmp(@X,QID,Ret,VID,f_empty()) :- provQuery(@X,VID,Time,Ret),
       prov(@X,VID,Time,RLoc,RID), RID!=NULL.
```

```
// Recursive case - Local
idb2 execQuery(@RLoc,RID,ExecTime,RQID,X) :- provQuery(@X,VID,Time,Ret),
       prov(@X,VID,Time,RLoc,RID), RQID=f_sha1(QID+RID), RLoc==X,
       ExecTime=Time.
// Recursive case - Remote
idb3 execQuery(@RLoc,RID,ExecTime,RQID,X) :- provQuery(@X,VID,Time,Ret),
       prov(@X,VID,Time,RLoc,RID), RQID=f_sha1(QID+RID), RLoc!=X,
       recv(@X,VID,Time,RLoc,ExecTime).

// Buffer sub-results
idb4 pResultTmp(@X,QID,Ret,VID,Buf) :- execResults(@X,RQID,RID,Prov),
       pResultTmp(@X,QID,Ret,VID,Buf1),
       RQID=f_sha1(QID+RID), Buf=f_concat(Buf1,Prov).

// Calculate and return results
idb5 provResults(@Ret,QID,VID,Prov) :- pResultTmp(@X,QID,Ret,VID,Buf),
       numChild(@X,VID,C), C=f_size(Buf), Prov=f_pIDB(Buf,VID,X).
```

To customize provenance computations in the distributed graph traversal query, we introduce three user defined functions: `f_pEDB`, `f_pIDB`, and `f_pRULE`, which operate on the base tuples (`f_pEDB`), intermediate derivations (`f_pIDB`), and rule execution instance (`f_pRULE`). In Section 4.3.4, we describe these functions in greater detail, and show, via examples, how they can be customized to return different provenance representations and granularities.

The initial query is indicated by event `provQuery(@X,VID,Time,QID,Ret)`, where a query is issued to retrieve the provenance information of tuple `VID` stored on `X` at `Time`. An additional attribute `QID` is added to uniquely identify the query, and the query result should be returned to `Ret`. Note that upon receiving this query, node `X` executes rules `edb1`, `idb1`, `idb2` and `idb3`.

Rule `edb1` is the base case and applies when the tuple `VID` is a base tuple (EDB), as indicated by the fact that it has no associated rule execution instance (that is, `RID` is null). In such cases, the provenance information is `f_pEDB(VID)` — the result of applying the user-defined function for EDBs to `VID`. An example of `f_pEDB` is simply return the tuple itself, indicating the base tuple is involved in the derivation.

Rule `idb1` initializes the `pResultTmp` table, which is later used to buffer intermediate query results. Rule `idb2` and `idb3` represent the recursive case in which the `prov` table is retrieved. Each entry with matching `VID` in the `prov` table indicates a rule execution instance that leads to the derivation of `VID`. These rule execution instances are additionally retrieved and buffered in `pResultTmp` table by issuing a query `execQuery(@RLoc,RID,ExecTime,RQID,X)`. The rule execution time (i.e., `ExecTime`) is computed differently, based on whether it corresponds to a local (Rule `idb2`) or remote (Rule `idb3`) rule execution. The process continues recursively, where the nodes receiving the `execQuery` message retrieve the matching `ruleExec` tuples, and recursively traverse children derivations until the base case is reached.

NetTrails applies rule `idb4` when all children derivations have returned with the provenance information. The resulting provenance information is then combined in rule `idb4` using the `f_pIDB` function and the results are returned to the query node.

An additional four rules `rv1-rv5` (similar to `idb1-idb4`) perform a similar traversal of the `ruleExec` tables. The intuition behind these rules is that the user recursively traverses `prov` and `ruleExec` tables across nodes until the entire provenance tree has been obtained. Since each rule execution takes several predicates as input, an additional user defined function `f_pRULE` enables the user to customize how the various inputs to the rule can be combined in the provenance tree.

4.3.4 Query Customization

Given the general querying framework presented in Section 4.3.3, we now describe how users may customize the query processing for application requirements.

First example – Provenance Polynomials. Our first customization example stores provenance information in the form of an algebraic expression called a *provenance polynomials* [32]. Provenance can be encoded as an algebraic structure with two binary operations — addition and multiplication — indicated by "$+$" and "$\cdot$", where "$+$" indicates the combination of tuples with union and projection and "$\cdot$" denotes a natural join over tuples. The literals in the algebraic expression represent base tuples. By customizing the "$+$" and "$\cdot$" operators, various types of classic provenance annotation can be encoded. For example, $r_1(A+r_2(B\cdot C))$ indicates that rule r_2 applies JOIN on tuples B and C, and the result is then UNIONed with A in r_1.

To return provenance query results as polynomials, the three user-defined functions are implemented as follows:

- **`f_pEDB(VID)`** takes as input the VID that uniquely identifies the base tuple. The function simply returns the base tuple itself or its primary keys (which can be retrieved by reading a systems table that maps VIDs to tuples).

- **`f_pIDB(Derivations,Loc)`** takes as input `Derivations` that contain the polynomials (D_1,D_2,...,D_n) that represent all possible n ways to derive the tuple, and `Loc`, the location specifier of the tuple. The function iterates over all entries in `Derivations` and applies a "$+$" operation across them. The resulting provenance expression is then further annotated with the location as $(D_1 + D_2 + ... + D_n)@Loc$.

- **`f_pRule(ChildPred,R,RLoc)`** takes as input `ChildPred`, representing the polynomials of all n input tuples (P_1,P_2,...,P_n) that are used in the execution

	Node Set	# of Derivations	Derivability Test
f_pEDB	$\{NodeID\}$	1	$True$
f_pIDB	$\bigcup_{i=1}^{n} D_i$	$\sum_{i=1}^{n} D_i$	$\bigvee_{i=1}^{n} D_i$
f_pRULE	$\bigcup_{i=1}^{n} P_i$	$\prod_{i=1}^{n} P_i$	$\bigwedge_{i=1}^{n} P_i$

Table 4.4: User-defined functions for various queries

of rule `R` at location `Loc`. The function iterates over all entries in `ChildPred` and applies a "$\cdot$" operation across them. As above, the result is affixed with a rule label and location. The function returns the polynomial $\langle R@RLoc\rangle(P_1 \cdot P_2 \cdot ... \cdot P_n)$.

Additional examples. Table 4.4 presents additional representative examples of possible provenance customizations. (D_1,D_2,...,D_n) contains the provenance annotations of all possible ways to derive a given tuple VID, and ($P_1, P_2, ..., P_n$) denotes similar annotations of input tuples to a particular rule execution instance.

The first example, `NodeSet`, returns the set of nodes that participate in the derivation of a tuple. For each base tuple, `f_pEDB(VID)` returns the node ID where `VID` is stored. `f_pIDB(Derivations,Loc)` and `f_pRULE(ChildPred,R,RLoc)` both return the union of the set of nodes. Note that this query can be trivially extended to return the number of unique nodes participating in the query.

The second example returns the number of possible derivations of a given tuple. We define the three user-defined functions as follows: `f_pEDB(VID)` evaluates to 1, indicating each of the edb tuples has one derivation. For intermediate derived tuples, `f_pIDB(Derivations,VID,Loc)` calculates the sum of the sub-results, where each D_i is the number of derivations over its sub-results. For rule execution instances, `f_pRULE(ChildPred,R,Rloc)` is defined as the product of the sub-results.

Similarly, the user can modify the user-defined functions for derivability tests.

NetTrails's flexibility enables numerous other customizations. In all cases, the user need modify only the three user defined functions (`f_pEDB`, `f_pIDB` and `f_pRule`). The underlying NDlog program used for querying provenance is sufficiently general to support a diverse set of provenance applications.

As a final example, we briefly outline how *graph projection* can be achieved using NetTrails. When querying provenance, users may impose constraints that only allow a specific subset of the provenance information to be counted in the annotation computation. For example, in a multi-administrative domain, a node may trust only the provenance information within its own domain. This requires provenance information to be *projected* based on the constraints imposed by users. Projection of the provenance graph is straightforwardly achieved using NetTrails's querying framework by setting additional conditions during the traversal in the provenance graph. More concretely, when rule `idb2` is triggered, rather than spawning `eRuleQuery` for each of the alternative derivations, the rule can instead send the event to a particular targeted set of rule execution vertices.

NetTrails's querying framework is directly applicable to various domains. For example, in distributed trust management, access requests may be granted or denied based on the nodes involved in formulating the request. Alternatively, a trust value may be assigned to each derivation based on a specific definition of trust. In the domain of recursive view maintenance, one may use the provenance to perform efficient incremental deletion [59] by performing the derivability tests.

4.4 Correctness

In this section, we prove that the provenance maintenance presented in the previous sections can faithfully capture the provenance of a system execution based on the provenance model defined in Chapter 3.

We first show that, given an execution trace $\mathcal{E}$, the provenance information captured by proactive maintenance is consistent with $G(\mathcal{E})$, the provenance graph for $\mathcal{E}$. More specifically, we can find a mapping $\mathcal{M}$ between the tuple instances in tables `prov`, `ruleExec`, `send` and `recv`, and their corresponding INSERT (or DELETE), DERIVE (or UNDERIVE), SEND and RECV vertices in $G(\mathcal{E})$.

The execution logic of a prescribed protocol $\mathcal{P}(N_i)$ is captured as a set of derivation rules (i.e τ :- $\tau_1, \tau_2, ..., \tau_n$). Given the semi-naïve evaluation, each rule is rewritten into delta rules, which are in the form of $\triangle\tau$:- $\triangle\tau_i, \tau_1, ..., \tau_{i-1}, \tau_{i+1}, ..., \tau_n$, where $\triangle\tau_i$ corresponds to the trigger event locally derived or encapsulated in a received message, $\tau_1, ..., \tau_{i-1}, \tau_{i+1}, ..., \tau_n$ correspond to the preconditions, and $\triangle\tau$ corresponds to the generated update. According to the algorithm described in Section 4.2, for a single event (i.e., state transition), the generated tuple instances in tables `prov`, `ruleExec`, `send` and `recv` are consistent with the provenance denotation for this transition. Provided that provenance is monotonic, the final `prov`, `ruleExec`, `send` and `recv` tables are consistent with the denoted provenance $G(\mathcal{E})$, after state transitions $\mathcal{S}_0 \to \mathcal{S}_1 \to ... \to \mathcal{S}_n$.

We next show that the reactive provenance maintenance yields the same results as proactive maintenance. By assumption, the execution of a node N_i can be deterministically replayed, by taking the non-deterministic events recorded in the logs. Therefore, given the inputs of N_i (recorded as a log), N_i can regenerate the state transition in the same sequence as runtime, i.e., $\mathcal{S}_0 \to \mathcal{S}_1 \to ... \to \mathcal{S}_{n-1}$ (to generate the provenance for transition $\mathcal{S}_{n-1} \to \mathcal{S}_n$). At the end of the deterministic replay, node N_i is at the same state ($\mathcal{S}_{n-1}$) as the runtime, therefore the provenance maintenance is performed on the same state.

4.5 Evaluation

In this section, we experimentally evaluate the maintenance and querying performance of the NetTrails provenance system. The goal of our evaluation is two-fold: (1) to measure the performance overhead incurred by supporting TAP provenance; and (2) to comparatively study the performance tradeoffs between different maintenance and querying schemes.

We perform two sets of experiments. In the first set, we focus on the performance overhead for supporting the TAP provenance. We evaluate the overhead for both proactive and reactive maintenance and their corresponding query performances. The second set of experiments further studies the performance overhead for snapshot provenance (see Section 3.2) which captures provenance only for the derivations of *current* system state.

4.5.1 Performance of TAP Provenance

Our first set of experiments evaluates NETTRAILS using the following maintenance techniques from Section 4.2.2: (1) `NoProv` is a baseline in which no provenance information is maintained or queried; (2) `Proactive` maintains provenance proactively using logs to store provenance deltas; and (3) `Reactive-Chk` and `Reactive-NoChk` respectively maintain provenance reactively with and without checkpointing using per-node input logs. Additionally, as a comparative evaluation, we also measure the performance for maintaining and querying `Snapshot` provenance, which captures provenance only for the derivations of the *current* system state.

Experimental setup. We execute multiple instances of NETTRAILS using the ns-3 [74] network simulator. By running NETTRAILS over ns-3, we can study the scalability trends of NETTRAILS in a more controlled and repeatable environment.

Our experiments focus on the provenance maintenance and querying overheads of a declarative path-vector routing protocol called PATHVECTOR that consists of four rules. PATHVECTOR is an extension of the MINCOST protocol from Section 2.2; it stores an additional path attribute that encodes the actual path from a

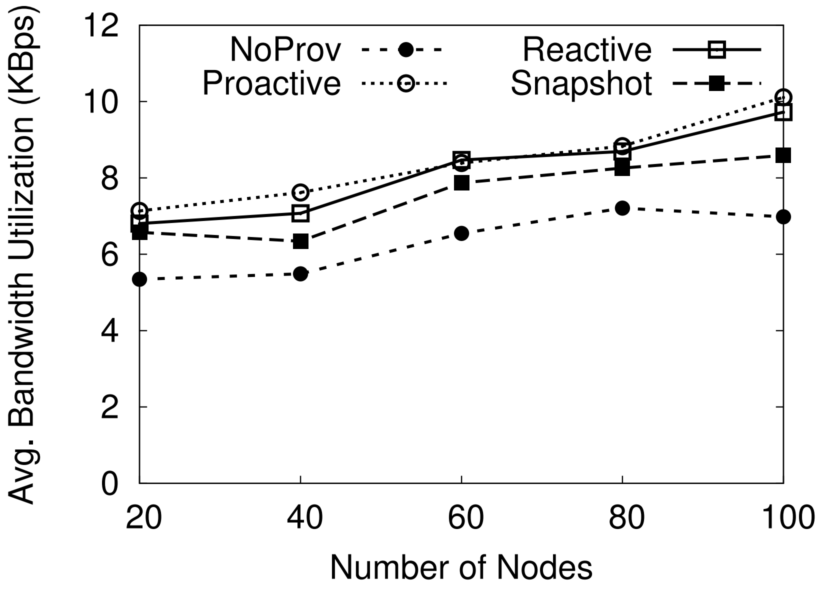


Figure 4.2: Average per-node bandwidth utilization (KBps).

given source to a destination. PATHVECTOR computes the shortest path (with minimal hop count) between any two nodes. We have chosen to evaluate PATHVECTOR because path-vector protocols are used widely; for example, they serve as the basis for the Internet's interdomain routing protocol.

For our experiments, we use the declarative version of PATHVECTOR that was previously developed by Loo et al. [63]. The program is executed continuously over a dynamic network; as links are updated, new `mincost` tuples are derived, and provenance logs are generated for subsequent reconstruction. Each NETTRAILS node runs the PATHVECTOR protocol; in steady state, we add or delete two links every second. Our simulations last for 300 virtual seconds and are executed on machines with X3450 Xeon 2.66 GHz processors and 4 GB memory that run Fedora 12 (64-bit).

Provenance bandwidth overhead. In our first experiment, we vary the number of simulated NETTRAILS nodes from 20 to 100. Figure 4.2 shows the average

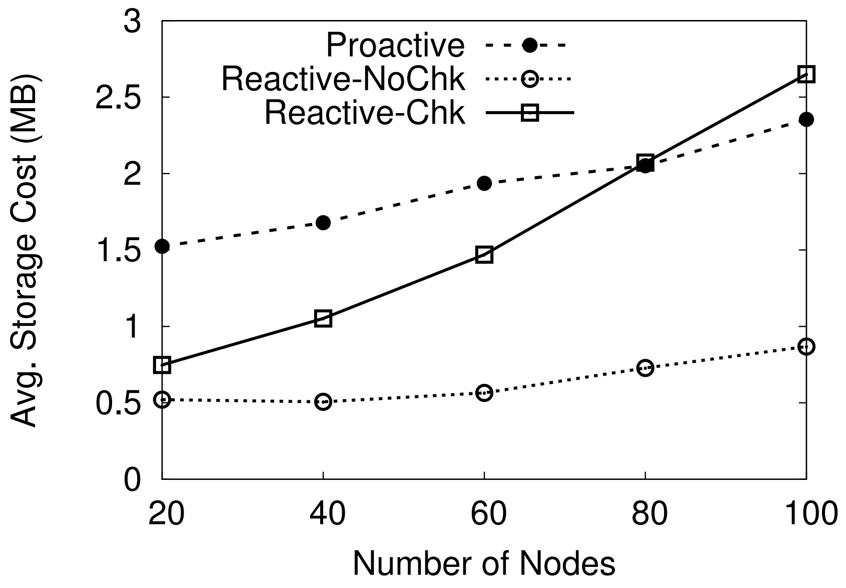


Figure 4.3: Average per-node storage overhead (MB).

amount of bandwidth used by each node; since checkpointing increases only local storage and does not result in any additional communication, we show only the results for `Reactive-NoChk` (i.e., without checkpoints). As expected, `NoProv`'s bandwidth usage was the lowest. We also observe that `Proactive` and `Reactive` incurred roughly the same bandwidth overhead. The reason is that both techniques add a few attributes (such as sender timestamp, address, and tuple identifier) to each message that is sent as part of the protocol execution. In both cases, the per-node bandwidth requirement in the 100-node network was less than 10.5 KBps. In comparison, `Snapshot` incurred a relative 88% bandwidth usage. The differences are mainly because of the different sets of attributes that were added to each message; `Snapshot` does not include timestamps in each of the message communicated.

Provenance storage overhead. NETTRAILS's per-node storage overheads are shown in Figure 4.3. We report only the storage used for logs (i.e., the overhead

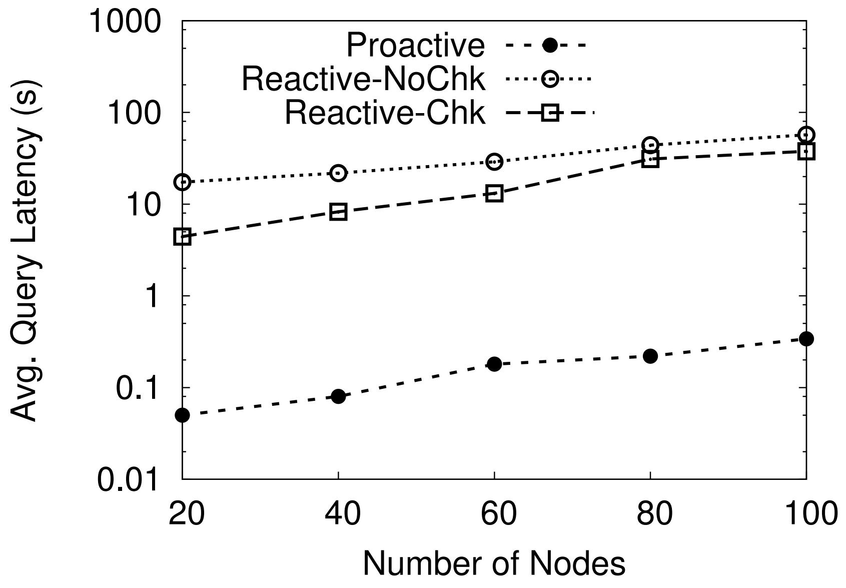


Figure 4.4: Query latencies (seconds) for various network sizes.

incurred by NETTRAILS), so we omit the results for `NoProv` and `Snapshot`, which maintain no logs. In `Reactive-Chk`, checkpoints were taken once every minute.

`Proactive` produced the largest logs because it explicitly stores each change to the `prov` and `ruleExec` tuples. `Reactive-NoChk` produced the smallest logs, at approximately one third of the size of the `Proactive` logs. The storage overhead for `Reactive-Chk` grows linearly with network size. Beyond a network size of 80, `Reactive-Chk` actually exceeds `Proactive`, since checkpoints grow with the size of the network. However, we note that the crossover point depends on additional factors such as the update rate of base tuples. Overall, the storage overhead is modest: in the largest network (100 nodes), it amounts to 2.65 MB in 300 seconds (approximately 9 KBps).

Query latency. Our second set of experiments evaluates the query latency of `Proactive`, `Reactive-NoChk`, `Reactive-Chk`. We use the same experimental setup as above. Additionally, we randomly issue 100 queries from different nodes, each of which retrieves the provenance of a particular tuple derived in the past.

Figure 4.4 shows the average query latency for various network sizes. In `Proactive`, the average query latency is within 0.34 seconds for the 100-node network, and each query's latency is dominated by network propagation delay. `Reactive-Chk` returns query results within 37.7 seconds on average. The overhead is dominated by log replays from a given checkpoint. `Reactive-NoChk` incurs the highest query latency as expected, due to the need to replay logs in their entirety. However, as noted in Figure 4.3, `Reactive-NoChk` has significantly lower storage overhead compared to other schemes, demonstrating a tradeoff in storage overhead and latency.

4.5.2 Performance of Snapshot Provenance

In our second set of experiments, we further evaluate the performance of two variants of the proactive maintenance scheme — the *value-based* provenance and the *reference-based* provenance [113]. In value-based provenance, each derived tuple includes its entire provenance when transmitted between nodes. On the other hand, in reference-based provenance, which is adopted in NETTRAILS, tuples contain only pointers that may be resolved recursively to reconstruct their derivations (see Section 4.1).

Experimental Setup. We perform a combination of simulation and deployment experiments. Our simulation experiments are carried out using the ns-3 [74] network simulator. We generate transit-stub topologies for our simulation experiments using the GT-ITM topology generator [35]. The transit-stub topology consists of eight nodes per stub, three stubs per transit node, and four nodes per transit domain. We increase the number of nodes in the network by increasing the number of domains.

Our deployment experiments are executed within a local cluster of eight dual-core Intel 2.8GHz Pentium D hosts and 16 quad-core machines with Intel Xeon 2.33GHz CPUs. All machines run Linux 2.6 and are interconnected by high-speed Gigabit Ethernet. NETTRAILS communicates messages between nodes via UDP packets. To increase the size of our network, we execute two instances of NETTRAILS on each quad-core machine, enabling us to scale our implementation experiments to 40 nodes.

As workloads for our simulation and deployment experiments, we use two NDlog applications: MINCOST, introduced in Section 2.2, computes the costs of the best (least cost) paths between pairs of nodes; and PATHVECTOR, introduced in Section 4.5.1, extends MINCOST and enables a node to discover the best path (transmitted as a vector of nodes) to a specified destination. For all experiments, each node is initialized with a `link` tuple for each of its neighbors. That is, nodes have *a priori* knowledge of their local links and use MINCOST and PATHVECTOR to discover longer network paths. Link costs are fixed at 1, and hence MINCOST measures hopcount to the destination.

Communication Overhead. In the case of value-based provenance, each tuple carries its (potentially lengthy) derivation history. Reference-based provenance attempts to decrease this overhead by communicating pointers to provenance information rather than directly conveying the information.

Figure 4.5 plots the communication cost (the number of transmitted bytes before reaching the fixpoint), averaged over all nodes, for various sized simulated networks when nodes execute the MINCOST program. (For readability, the order of the labels in all figures in this section mirror the ordering of the plotted curves.)

Value-based provenance results in significant communication overhead. For example, in the 300-node network, value-based provenance quadruples the cost of conducting the query compared with executing MINCOST without provenance

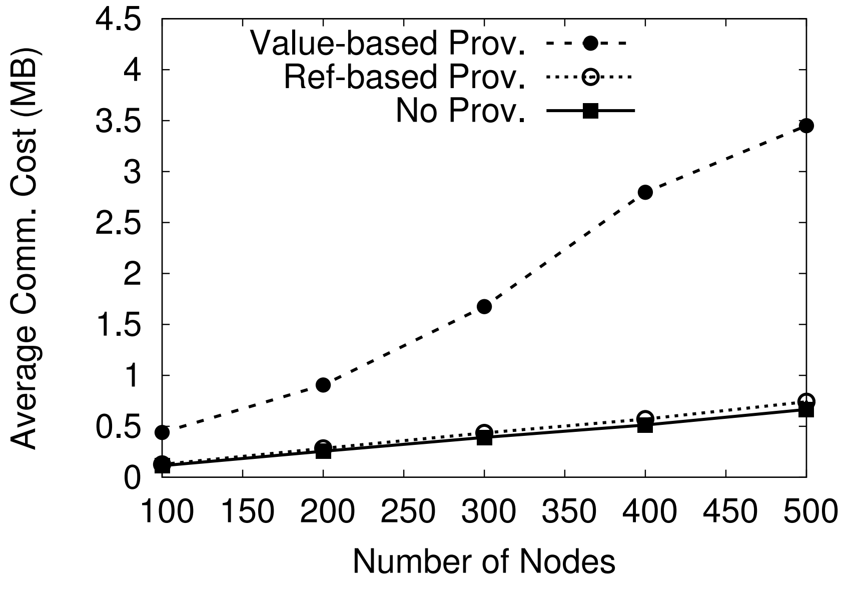


Figure 4.5: Average communication cost (MB) for MINCOST.

(line "No Prov."). In contrast, reference-based provenance (line "Ref-based Prov.") incurs little communication overhead, increasing the communication cost by just 0.04 MB (11.3%) in the same 300-node network. The vast difference in bandwidth costs is due to MINCOST's ability to produce multiple derivations for a given `mincost` tuple. All possible derivations must be communicated with each tuple when using value-based provenance. Our reference-based technique reduces the provenance information that must be transmitted since the same pointer may be shared between different derivations.

Due to the memory constraints of running large-scale ns-3 simulations on a single machine, we experimented with up to 500 nodes in our simulations. However, our results clearly demonstrate promising scalability trends for all protocols: the average communication costs of MINCOST protocol (without provenance) scale linearly with the number of nodes. This matches the expected scalability behavior for the protocol. Moreover, with the addition of reference-based provenance,

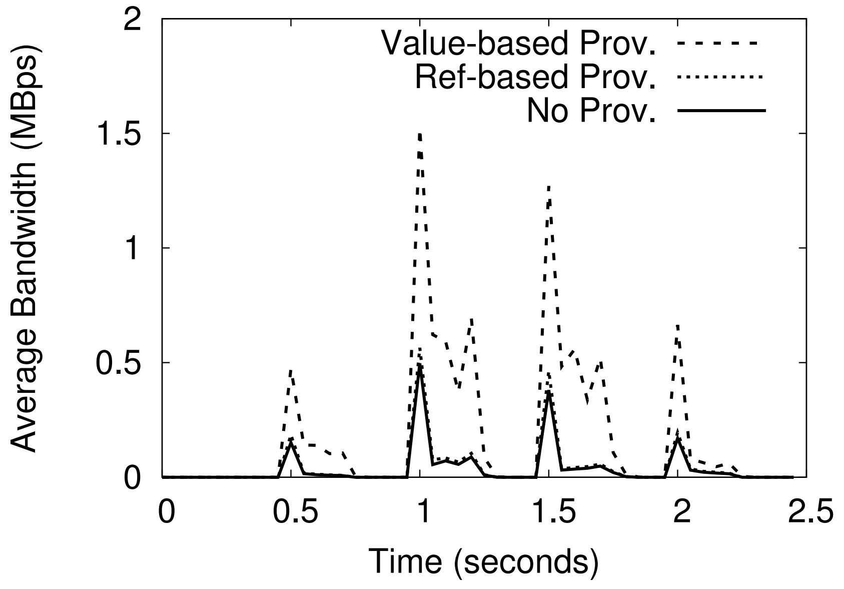


Figure 4.6: Average bandwidth cost (MBps) for MINCOST under churn.

we note that the communication costs continues to scale linearly, hence maintaining the original scalability trends, demonstrating that reference-based provenance incurs minimal impact on the scalability of an existing protocol.

Incremental Maintenance. The experiments described above model a static topology in which nodes neither leave nor join the network and links never fail. Here, we evaluate NETTRAILS's ability to mitigate a high level of node churn and link failure. We model churn by adding or deleting ten randomly selected stub-to-stub links in a 200-node simulated network (originally containing 315 stub-to-stub links) every 0.5 seconds, with link addition or deletion occurring with equal probability.

Figure 4.6 shows the respective average per-node bandwidth costs of running the MINCOST protocol. Reference-based provenance does not incur significant bandwidth overhead (the lines for `No Prov.` and `Ref-based Prov.` closely over-

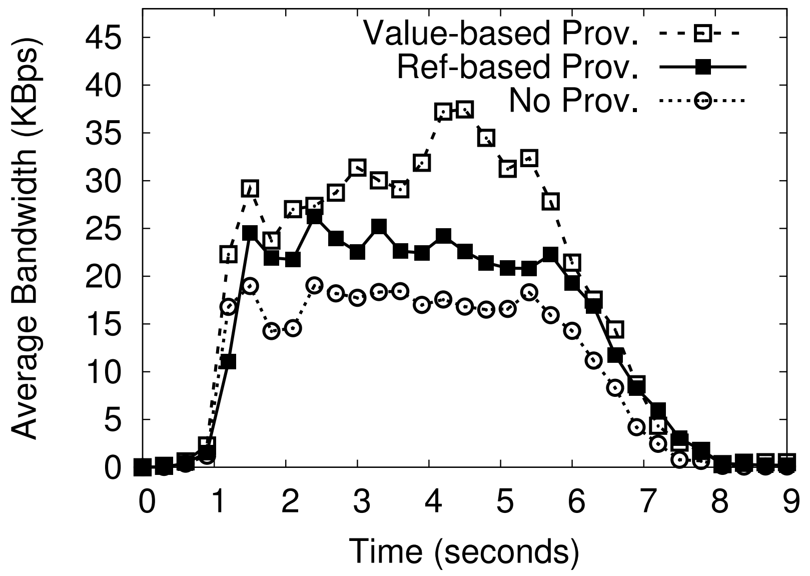


Figure 4.7: Average bandwidth cost for PATHVECTOR in testbed deployment.

lap). For example, the maximum increase in bandwidth due to reference-based provenance (relative to conducting the query without provenance) is 0.07 Mbps. Value-based provenance consumes significantly greater bandwidth as complete provenance information must be affixed to each transmitted tuple. The respective increases in bandwidth for MINCOST are 1.0 Mbps, 1329% greater than the equivalent overheads for reference-based provenance.

For both reference- and value-based provenance, a new fixpoint is reached within 0.5 seconds, indicating that NETTRAILS is resistant to even high levels of churn regardless of the type of provenance used.

Testbed Experiments To empirically evaluate NETTRAILS's computation and communication properties, we installed 40 instances of NETTRAILS in a local cluster. Since nodes on the cluster are fully connected via a shared switch, we impose a less trivial virtual topology as follows: to ensure reachability, nodes

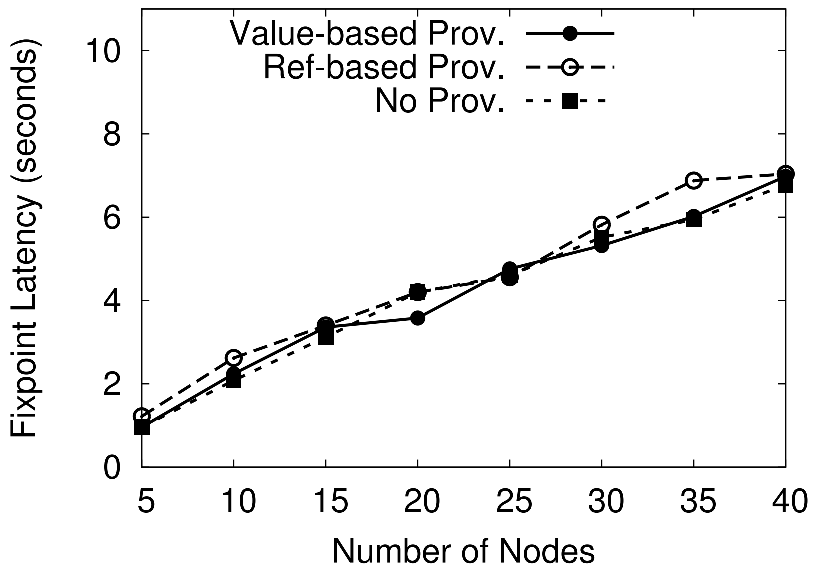


Figure 4.8: Fixpoint times for PATHVECTOR in testbed deployment.

are arranged in a ring structure. Each node in the network has links to its two neighbors (hence achieving the ring structure). Additionally, each node has a link to a random peer such that the maximum degree of all nodes is always three (a link to each ring neighbor and a third to a random peer). All nodes execute the PATHVECTOR protocol.

As with the simulation experiments, our reference-based provenance technique significantly reduces the overhead of provenance compared to the value-based approach. When sending no provenance information, the average per-node bandwidth cost of executing PATHVECTOR is 1.24 KB before a fixpoint is reached. Reference-based provenance increases this cost by 29%, far less than the 204% increase caused by value-based provenance. This trend can be observed from Figure 4.7 which plots the average per-node bandwidth over time for the experiments. The relative overheads of reference- and value-based provenance mirror our earlier simulation results.

In addition to examining bandwidth costs, our deployment provides a mechanism to study the computational overhead of using the different provenance techniques. Figure 4.8 shows the fixpoint time for different network sizes. (As an invariant of network size, the degree of each node in the network is fixed at three.) As can be discerned from the Figure, neither provenance technique imposes any significant increase in fixpoint time.

To summarize, the results of our deployment experiments indicate that reference-based provenance achieves a substantial decrease in communication cost as compared to value-based techniques, while imposing little or no increase in fixpoint latency.

4.6 Summary

In this chapter, we show how TAP provenance can be maintained and queried at scale in distributed settings. We present the relational storage model of provenance, and discuss two alternative approaches for provenance maintenance. We show that, given the derivation rules that specify the system execution logic, an automatic rewrite strategy can be employed to augment the original system with additional rules for provenance maintenance. Moreover, provenance queries are formulated as recursive queries over the distributed provenance data, and are specified in a declarative framework that allows easy customization for various applications. Through a combination of simulation and testbed deployment, we demonstrate that provenance can be maintained and queried with reasonable overhead. In the subsequent chapter, we discuss several optimization opportunities that further reduce the maintenance overhead and/or query performance.

Chapter 5

Optimizations

In Chapter 4, we have demonstrated the practicality and scalability of the proposed provenance maintenance and querying techniques. In practice, however, individual applications may impose different performance optimization goals that require tailored treatment in system design and implementation. For instance, fault detection systems may frequently issue provenance queries to monitor the progression of system execution, and, therefore, require optimized query performance; on the other hand, provenance information is not actively queried for attack analysis in a benign environment, so minimizing the maintenance overhead at runtime is prioritized in this scenario. Often times, the maintenance and querying techniques need to be customized to match these optimization goals.

In this chapter, we discuss several optimization techniques that balance the tradeoff along several axis. Section 5.1 presents a quantitative study of the tradeoffs between the proactive and reactive maintenance schemes; and Section 5.2 further discusses techniques that optimize query performance (such as query latency and communication overhead). We conclude the chapter by demonstrating the effectiveness of the proposed the optimizations in Section 5.3.

5.1 Cost-based Optimizations

The maintenance approaches introduced in Chapter 4 offer a spectrum of trade-offs between maintenance overhead and querying performance. The best tradeoff depends on a variety of factors, some of which we discuss below.

Querying frequency. We expect that the cost for query processing will be a function of (1) how frequently queries are issued, (2) how far apart the checkpoints are in the log, and (3) how much work is required to replay a log segment. If queries are expected to be rare, we can save space by maintaining input logs and taking checkpoints only occasionally. In this case, answering a query can be expensive because the relevant parts of the provenance graph must be reconstructed by replaying the execution of certain nodes from their latest checkpoint.

If queries are more frequent, we can trade some space for a lower query-processing cost by (1) taking checkpoints more frequently (for reactive maintenance), which reduces the expected length of the log segment that needs to be replayed, and/or (2) maintaining provenance deltas rather than input logs. The latter reduces the computational cost because replay needs only to incrementally apply the changes to the provenance data and does not require repeating the processing steps that produced them.

System runtime. Many distributed systems run for an indefinite amount of time. For example, the Internet's interdomain routing system has been running for decades. In such systems, checkpoints are indispensable because it is not practical for the querier to replay the execution of the system, or even just a single node, from the very beginning. On the other hand, there are distributed systems that run only for a limited time. For example, in this case, replaying the entire log may be practical, and if so, we can save even more space by not maintaining checkpoints.

Local derivations. Distributed systems differ in the relative frequency of remote derivations (i.e., derivations that involve message exchanges between nodes). When most derivations are remote, both provenance deltas and input logs should perform equally well since most state changes (which are recorded in provenance deltas) are due to incoming messages (which are recorded in the input logs). However, there are systems where most derivations are local; for example, a distributed machine-learning algorithm might just send a very few messages to transfer the raw data and the results. In this case, input logs should consume significantly less space than provenance deltas, but they would need much more computation when the provenance graph needs to be reconstructed to answer a query.

5.1.1 Cost Model

In order to decide which maintenance strategy(-ies) to adopt, we develop a cost model that captures the tradeoffs between maintenance and querying overhead. Our cost model takes as its inputs a set of runtime statistics collected from the system, including workload properties (e.g., message and query frequencies), the characteristics of the running protocol (e.g., the ratio of local derivations), and the measured overhead for reading and writing log entries.

The model parameters are summarized in Table 5.1 and are broadly classified into three categories: (1) properties of the NDlog program that the system executes, (2) properties of the nodes on which the system is deployed, and (3) properties of the workload. Unless otherwise specified, model parameters are system-wide; they are obtained by profiling the deployed system at runtime. Each node first averages its local statistics (e.g., the number of messages per unit time), and then the results are averaged across all nodes.

NDlog program. The first set of parameters relate to the distributed protocol itself. Since we have assumed (in Section 2.2) that the protocol is specified in NDlog, the

NDlog Program	
N_{pred}	# of predicates in rule body
N_{dep}	Node-level depth of the provenance graph
N_{exec}	# of derivation rules triggered by a message
N_{dup}	# of duplicate derivations per tuple
System Performance	
S_{log}/S_{chk}	Size of a log entry / checkpoint
T^{w}_{log}/T^{r}_{log}	Time taken to append / retrieve a log entry
T^{w}_{chk}/T^{r}_{chk}	Time taken to save / load a checkpoint
T_{rule}	Time taken to execute a delta rule
$T_{latency}$	Average propagation delay between two nodes
Input Workload	
F_{msg}	Message freq. (# of messages per unit time)
F_{qry}	Query freq. (# of queries per unit time for the entire system)
I_{chk}	Checkpoint interval (# of unit times between adjacent checkpoints)

Table 5.1: Summary of the statistics used for cost-based optimizations.

properties are expressed in terms of the structure of the program. N_{pred} denotes the average number of predicates in a rule body; for instance, in the MINCOST program from Section 2.2, rule `mc2` has a complexity of two because its rule body contains the predicates `link` and `mincost`. N_{dep} denotes the average node-level depth of a provenance tree for any tuple derived using the program. Note that this is not the same as vertex-level depth; for instance, in Figure 3.1, $N_{dep} = 2$, since the graph is partitioned at nodes c and b. N_{exec} is the number of derivation rules that are triggered (executed) by an incoming message at a given node. This includes all local rules executed until a local fixpoint is reached. For instance, in the MINCOST program from Section 2.2, an incoming `path` message will trigger rule `r3`, which may further trigger rule `r2` (if the received `path` is optimal). Finally, N_{dup} is the average number of duplicate derivations per tuple. N_{pred} can be analyzed statically from the protocol specification, whereas N_{dep} and N_{exec} are collected at runtime, e.g., by observing the previous provenance query results and rule executions.

System performance. The next set of properties relates to the runtime environment in which the distributed system is deployed, e.g. with respect to I/O, computation power, and message propagation delay in the network. S_{log} and S_{chk} are the average sizes of a log entry and a checkpoint, respectively; T_{log}^{w} and T_{log}^{r} are the times required to append and retrieve a log entry; T_{chk}^{w} and T_{chk}^{r} are the times required to save and load a checkpoint; T_{rule} is the average execution rule execution time; and $T_{latency}$ is the average propagation delay between two nodes.

Input workload. The final set of properties relate to external inputs that drive the execution of the protocol. For example, the rate at which links are updated in the MINCOST program has a direct impact on the number of times rules are fired on a node and the size of the log. Rather than capture the rate of change at a predicate level, we instead measure the average frequency of incoming messages at each node during protocol execution, denoted as F_{msg}. Unlike other parameters, F_{qry} represents a system-wide total, i.e., it represents the number of queries issued to the entire network.

NETTRAILS captures parameters at a coarse granularity (typically, system-wide averages of per-node averages). Our model can capture costs at finer granularity, e.g., at the level of individual relational operators, but the requisite fine-grained data leads to a massive number of parameters. As we show in Section 5.3.1, coarse-grained statistics are sufficient for the model to provide accurate estimates on actual system performance.

5.1.2 Applying the Cost Model

The proactive and reactive provenance maintenance techniques offer storage and latency tradeoffs: the proactive scheme has a higher log storage overhead but offers lower query latency than the reactive strategy. NETTRAILS applies the above cost model to estimate the storage and latency overheads, and then, as discussed in Section 5.1.3, selects the strategy that is likely to perform best.

- **Storage overhead.** In the proactive approach, the log stores `+/-prov`, `+/-ruleExec`, and incoming `recv` tuples as they are generated by the underlying protocol. Each log entry requires S_{log} space. With a message frequency of F_{msg}, the `recv` tuple yields a storage overhead of $F_{msg} \times S_{log}$, and the `prov` and `ruleExec` yield $2 \times F_{msg} \times N_{exec} \times S_{log}$.

  For the reactive approach, NETTRAILS needs maintain only one log entry for each incoming message. Here, the storage overhead is $(F_{msg} \times S_{log}) + (I_{chk} \times S_{chk})$, where the former term denotes the cost of storing the messages and the latter is the cost of storing the checkpoints.

- **Query latency**. We consider the time taken to construct a vertex (T_{vertex}) in the provenance graph in both the proactive and reactive cases.

  In the proactive approach, given a delta rule execution, three log retrievals are required to get the `recv`, `prov`, and `ruleExec` entries for the trigger event. We repeat this retrieval process for all duplicate derivations. Hence, $T_{vertex} = 3 \times T^r_{log} \times N_{dup}$.

  In the reactive approach, checkpoints need to be retrieved and replayed, requiring T^r_{chk} time. The expected number of log messages that must be retrieved and replayed by executing rules between two checkpoint intervals is $F_{msg} \times I_{chk}$. Hence, $T_{vertex} = T^r_{chk} + F_{msg} \times I_{chk} \times N_{exec} \times T_{rule}$.

  To estimate the average per-query latency, we would need to multiply by the average depth of the tree, and the time to replay each vertex (T_{vertex}) and propagation delays in sending the query results along the path of the tree, resulting in $N_{dep} \times (T_{vertex} + 2 \times T_{latency})$. By taking into account the query frequency (F_{qry}), one can further estimate the aggregate latency of all queries over a period of time.

5.1.3 Optimizations using the Cost Model

The above cost model can be used to select the maintenance mode that is likely to perform best according to a given metric. The choice of proactive or reactive provenance depends upon the underlying protocol and workload, as well as the metric (i.e., storage or query latency) that the administrator of the distributed system would like to optimize. If the administrator's goal is simply to minimize storage (resp. latency), then the straightforward approach of selecting the maintenance mode that incurs the least storage (resp. latency) cost is sufficient. More flexibility can be achieved with more complex objective functions; for instance, the administrator can adopt a strategy that uses reactive provenance when the estimated latency is lower than some threshold value.

Selecting the appropriate provenance maintenance mechanism can be done either prior to system deployment or at runtime. In the former approach, which we have adopted for our prototype implementation, NETTRAILS relies on *performance profiles* that capture the costs of a representative system. Cost-based analysis of the performance profile determines whether proactive or reactive provenance maintenance is likely to yield better performance.

Another possibility is *dynamic adaptation*, that is, adjusting the provenance maintenance mechanism at runtime. This requires the ability to reconstruct provenance from a log that contains a mix of provenance deltas and per-node input logs. Here, NETTRAILS takes advantage of the property that the two maintenance modes are interchangeable. For instance, given a query for a `prov` entry at time `t`, NETTRAILS can start replaying the log from the most recent checkpoint before `t`. If a `recv` message is encountered, the execution rules are fired to derive the corresponding `prov` and `ruleExec` tuples. On the other hand, if a `+/-prov` delta is read from the log, it can similarly be used to directly update the `prov` table.

5.2 Query Optimizations

In this section, we present a number of optimization techniques aimed at reducing the bandwidth and latency overheads of our distributed querying algorithm. These optimizations are orthogonal to which maintenance scheme the cost-based optimizer decides to use, and, therefore, can be deployed with the cost- based optimization simultaneously.

5.2.1 Query Results Caching

In the distributed storage and maintenance model adopted by NETTRAILS, each tuple maintains only reverse markers (via the `prov`, `ruleExec`, `send` and `recv` tables) that can be recursively traversed on demand in response to a query. An alternative approach, however, maintains and communicates the complete provenance information with each tuple derived. Because the provenance is readily available with the tuples, such an approach results in negligible query latency, in the cost of excessive communication and storage overhead (as we demonstrated in Section 4.5.2).

Our first optimization technique attempts to achieve a "sweet-spot" between these two approaches via the use of *query results caching*. In cases in which queries are rare, distributed provenance storage and maintenance aims to incur low communication overhead and minimally impact the convergence times of protocols. When queries are frequent, subsequent queries can leverage the results of the prior queries.

Caching scheme. Unlike traditional caching in a centralized database, the distributed provenance cache is distributed across several nodes in the network. Whenever a node `N` issues a distributed query to retrieve provenance information for a tuple `VID`, the resulting query results are not only cached at `N`, but also stored at intermediate nodes as the query results are returned along the reverse path.

Specifically, whenever rule `idb5` (or the equivalent rule for `ruleExec` traversal) is triggered (see Section 4.3.3), it indicates the completion of a query at an intermediate derivation. The query result will be maintained in events `provResults` or `execResults`. Before `provResults` or `execResults` is sent back to the query issuer, these results are cached in a `cache(@N,VID,Time,Results)` table that stores at node `N`, the provenance `Results` for `VID` at `Time`. Further attributes can be added to distinguish results based on provenance representation. Note that subsequent queries need not be for the exact tuple (i.e., `VID`) in order to benefit from the cache: since the intermediate results are cached along the reverse path, any graph traversal query that reaches node `N` and requires a subgraph rooted at `VID` can use the cache results. The cached results are then sent back on the reverse path back to the node conducting the query without further traversal.

5.2.2 Query Traversal Order

At each tuple vertex being traversed, the program shown in Section 4.3 simultaneously issues queries to all possible derivations. In essence, the distributed queries traverse the provenance graph using *Breadth First Order* (BFS). Intuitively, BFS must flood the queries throughout the whole provenance graph before any sub-results are obtained.

We explore another query traversal order, *Depth First Order* (DFS), in which alternative derivations are explored at each tuple vertex. DFS may incur longer querying latencies than BFS since the former can stall before a sub-result is received. However, DFS provides the opportunity for bandwidth savings for *threshold-based* queries in which a user asks, for example, whether a tuple has more than T derivations or whether fewer than T' unique nodes participate in the derivation. DFS allows such threshold-based queries to terminate as soon as the threshold is reached, without incurring additional communication overhead. That is, DFS trades off query latency in favor of reduced communication overhead for threshold-based queries.

The following modifications are required to adapt the BFS distributed graph traversal program (see Section 4.3.3) to a DFS query:

```
idb6 pQList(@X,QID,AGGLIST<Time,RID,RLoc>) :-
        provQuery(@X,VID,Time,QID,Ret),
        prov(@X,VID,Time,RLoc,RID), RID!=NULL.

idb7 eIterate(@X,QID,N) :- pResultTmp(@X,QID,Ret,VID,Buf),
        numChild(@X,VID,C), N=f_size(Buf)+1, N<=C,
        f_pIDB(Buf,X)<=Threshold.

idb2’ execQuery(@RLoc,RID,ExecTime,RQID,X) :- eIterate(@X,QID,N),
        pQList(@X,QID,LTime,LRID,LRLoc),
        RID=f_item(LRID,N), RLoc=f_item(LRLoc,N), RLoc==X,
        ExecTime=f_item(LTime,N).

idb3’ execQuery(@RLoc,RID,ExecTime,RQID,X) :- eIterate(@X,QID,N),
        pQList(@X,QID,LRID,LRLoc),
        RID=f_item(LRID,N), RLoc=f_item(LRLoc,N), RLoc!=X,
        recv(@X,VID,Time,RLoc,ExecTime), Time=f_item(LTime,N).

idb5’ provResults(@Ret,QID,VID,Prov) :-
        pResultTmp(@X,QID,Ret,VID,Buf), numChild(@X,VID,C),
        Prov=f_pIDB(Buf,X), C=f_size(Buf)||Prov>Threshold.
```

Instead of starting queries for each derivation simultaneously, the revised program iterates through the list of alternative derivations. The exploration of the next derivation is started only if the results of the previous explorations have been received.

To allow the result to be returned as soon as the threshold requirement is satisfied, the original rule `idb2` and `idb3` are replaced by four rules (`idb6`, `idb7`, `idb2'`, and `idb3'`) and an additional condition is added to `idb5` (as shown in rule `idb5'`). An additional set of similar rules is used for conducting a DFS traversal for each rule execution vertex, and is omitted for brevity.

The `pQList` table in rule `idb6` is a temporary table that maintains the list of rule execution vertices and their location. We introduce a special aggregation function `AGGLIST<A1,...,AN>` that generates lists for the designated attributes, indicated as `A1` to `AN`, for the tuples in each group.

Whenever there is an update in the `pResultTmp` table in rule `idb7`, the revised program counts the number of the results that have been ever received by checking the size of the buffer (that is, the `Buf` attribute in `pResultTmp`). If the current result is under the threshold, the program expands the next derivation using rule `idb2'` or `idb3'` (based on whether it corresponds to a local or remote rule execution). If not, rule `idb5'` is triggered and the result is returned.

In addition to BFS and DFS, *random moonwalk* [106] traversal can easily be implemented by randomly selecting N alternative derivations to explore, where N is a pre-defined constant. This technique is particularly useful when the number of a tuple's derivations is significantly large. The random moonwalk pinpoints with high probability the pivotal tuple that contributes to a derivation. In the context of networking applications, such random moonwalks are useful for ascertaining the dominating sources of incoming traffic [91].

5.2.3 Condensed Provenance

Our third optimization technique applies a previously proposed compression scheme known as *absorption provenance* [59] to the algebraic representation of provenance. Absorption provenance aims to reduce the number of variables in an algebraic representation. For example, consider the algebraic expression $a \cdot (a + b)$. By applying *boolean absorption rules* [59], the expression is reduced to $a \cdot (a + b) = a + (a \cdot b) = a$. Note that savings in size comes at the expense of information loss. In our example, absorption provenance loses the fact that b is also involved in the derivation. However, such absorbed encodings can still retain sufficient information for derivability tests or enforcing security policies based on the trust of source origins (base tuples).

To implement absorption, we utilize Binary Decision Diagrams (BDDs) [8] to encode provenance. BDDs provide a natural way to encode the algebraic representation of the provenance, and by default, apply absorption to save storage space. Since BDDs are frequently used in circuit synthesis and formal verification applications, highly optimized libraries provide abstract BDD types as well as Boolean operators that operate on them: pairs of BDDs can be ANDed or ORed; individual BDDs can be negated; and variables within BDDs can be set to true or false.

Note that the use of absorption provenance applies to both centralized provenance and value/reference-based distributed provenance. For example, in centralized and value-based provenance, the provenance information shipped for each tuple can be stored as BDDs. Similarly, for reference-based distributed provenance, query results can be returned in the form of BDD representations.

5.3 Evaluation

In this section, we present an experimental evaluation to demonstrate (1) the accuracy of the cost model developed in Section 5.1, and (2) the effectiveness of the proposed optimization techniques.

Upd. Interval (seconds)	**Proactive**		**Reactive-Chk**	
	Estimated	*Actual*	*Estimated*	*Actual*
4	1.0 MB	0.83 MB	4.4 MB	4.01 MB
2	1.4 MB	1.41 MB	4.5 MB	4.41 MB
1	2.2 MB	2.15 MB	4.8 MB	5.03 MB

Table 5.2: Comparisons between estimated storage (obtained from the cost model) and actual measured storage (in MB) for the path-vector protocol.

5.3.1 Cost-based Optimization

In the first set of experiments, we validate the cost model presented in Section 5.1.1. Our goal is to show that the model accurately predicts the storage and latency costs in a variety of different configurations. In particular, we show that, based on measurements of the parameters described in Table 5.1 (e.g., number of predicates, message frequencies, etc.) and the storage and query formulas specified in Section 5.1.2, NETTRAILS accurately estimates the storage overhead and query latency.

We repeat the experimental setup in Section 4.5.1 on actual physical machines in a local cluster testbed. Here, we utilize machines with a similar hardware and software configuration as the machine used in simulation. The machines are connected using high-speed Gigabit Ethernet. All NETTRAILS nodes run the same code as before, but communicate using actual network sockets instead of ns-3's simulated network stack.

Path-vector. We deploy the PATHVECTOR protocol on 60 NETTRAILS nodes (utilizing 60 cores on 15 cluster machines) for a duration of 900 seconds. Table 5.2 shows the differences between the estimated per-node storage (predicted using the cost model) and the corresponding actual storage overhead for each experimental run, repeated for different link update intervals. In both `Proactive` and `Reactive-Chk` (with a per-minute checkpoint interval), our results indicate that

Upd. Interval (seconds)	**Proactive**		**Reactive-Chk**	
	Estimated	*Actual*	*Estimated*	*Actual*
4	0.011 s	0.010 s	2.7 s	2.5 s
2	0.012 s	0.012 s	3.3 s	3.6 s
1	0.013 s	0.014 s	4.9 s	4.7 s

Table 5.3: Comparisons between estimated average query latency (obtained from cost model) and actual measured end-to-end latency (in seconds) for the path-vector protocol.

our cost model is accurate: the differences between the estimated and the measured storage costs range from 0.7% to 20.5%. Unlike the simulation results, the storage overhead for `Reactive-Chk` is significantly higher than `Proactive`. This is largely due to the relatively large checkpoints (compared to the I/O logs) that dominate the storage cost for `Reactive-Chk`. Our cost model is able to provide accurate estimation, such that users can fine-tune the checkpoint intervals accordingly.

Table 5.3 shows the differences between estimated and measured query latencies in a similar setup. The execution time is lower than in the earlier simulations, since our testbed experiments are carried out with lower update frequency, resulting in shorter log replay times between checkpoints. As before, our cost model accurately predicts query latencies for different frequencies of link updates; the differences between the estimated and measured latencies range from 0.0% to 10.0%.

Hadoop. We next validate the accuracy of our cost model on Hadoop MapReduce (version 1.0.0) [37]. We modified Hadoop so that the dependencies between incoming and outgoing tuples are reported to NetTrails and modeled as NDlog rules. Briefly, MapReduce consists of a *map* followed by a *reduce* phase. In the Map phase, each Map worker applies the user-defined Map function on each input tuple, and then locally combines intermediate results based on the partitioning key.

Number of Map/Reduce	**Proactive**		**Reactive-NoChk**	
	Estimated	*Actual*	*Estimated*	*Actual*
100/40	115.6 GB	115.6 GB	9.7 GB	9.7 GB
40/16	113.9 GB	113.9 GB	9.5 GB	9.5 GB

Table 5.4: Comparisons between estimated storage (obtained from the cost model) and actual measured storage (in GB) for Hadoop MapReduce.

Number of Map/Reduce	**Proactive**		**Reactive-NoChk**	
	Estimated	*Actual*	*Estimated*	*Actual*
100/40	6 s	8 s	41 s	47 s
40/16	15 s	19 s	62 s	66 s

Table 5.5: Comparisons between estimated average query latency (obtained from cost model) and actual measured end-to-end latency (in seconds) for Hadoop MapReduce

In the *reduce* phase, a reduce worker combines the outputs from Map workers and performs the reduce function. The dependency logic between incoming and output tuples of the map and reduce phases can be modeled as two NDlog rules each. Hence, the number of rules triggered per input (Table 5.1) is $N_{exec} = 2$.

We run Hadoop's WordCount program on up to 40 cores within our cluster. The program (WordCount) counts the number of occurrences of each word, given an input document size of 9.1 GB derived from the WebBase dataset [104] (dated 12/2010). In the first setup, 100 map tasks and 40 reduce tasks are executed, while in the second setup, 40 map tasks and 16 reduce tasks are executed.

Unlike the earlier path-vector experiment, the MapReduce execution is a one-time execution of static input data (as opposed to a continuously executed routing protocol). Since it does not make sense to perform periodic checkpoints for the duration of the execution, we compare only `Proactive` and `Reactive-NoChk`. Table 5.4 shows that the `Proactive` approach incurs 12 times larger storage overhead than `Reactive-NoChk` (which maintains only the original input files and the intermediate data communicated from map workers to reduce workers). This is largely due to the fact that the SHA1-based IDs assigned to `prov` and `ruleExec` entries are significantly larger than the input keywords: on average, a word is around 14 B (due to a large amount to HTML-specific tags), whereas each `prov` (or `ruleExec`) tuple is 35 B[1], i.e., the average size of a log entry (Table 5.1) is $S_{log} = 35$ B. Previous work [45, 81] has shown that MapReduce-specific optimization is possible to further reduce the size of log entries, however, we decide to retain the general-purpose provenance encoding in our evaluation.

Our estimated storage overhead is calculated based on the formula introduced in Section 5.1.2: the storage overhead is $N_{msg} \times (S_{log} + 2 \times N_{exec} \times S_{log})$, where N_{msg} denotes the number of input records to the map and reduce workers. Given that each input event triggers *exactly* two derivation rules, our cost model accurately estimates the storage overhead based on the number of input records to the map and reduce workers (reported by Hadoop).

Table 5.5 summarizes the average query latencies. We observe that, as expected, `Proactive` results in a lower query latency since no replay is necessary. Nevertheless, even with replay, `Reactive-NoChk` returns each query within 66 seconds. Replay latency is higher when the number of map/reduce workers is reduced since each worker is responsible for a larger set of input tuples (hence increasing the replay overhead of reexecuting the specific worker to generate dependency logic between incoming / outgoing tuples). In all cases, we note that our cost-model provides a good estimation of query latency.

5.3.2 Query Optimization

In our second set of experiments, we study the performance of *distributed querying* of provenance using the framework presented in Section 4.3. In addition, we

[1]In the Hadoop experiment, we use the first 80 bits of the SHA1 hashes for the IDs of `prov` and `ruleExec` tuples.

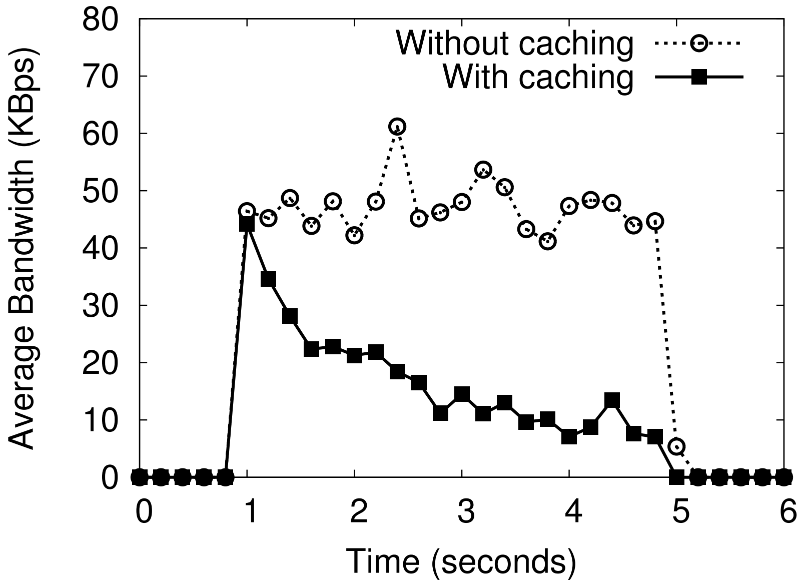


Figure 5.1: Average bandwidth cost (KBps) with and without caching.

validate the effectiveness of optimizations in reducing communication overhead during query. We adopt the same experimental setup as Section 4.5.2. The experiments are performed using a 100-node simulated network that runs the MINCOST protocol. Our query measurements begin after the network has reached a fixpoint. We utilize two queries in our evaluation: POLYNOMIAL and #DERIVATION. POLYNOMIAL acquires the provenance of an arbitrary tuple in the form of provenance polynomials (see Section 4.3.4). #DERIVATION computes the number of alternative derivations for a given tuple.

Caching Figure 5.1 plots the average per-node bandwidth over time when each node issues five POLYNOMIAL queries per second with each query targeted to a randomly selected `mincost` tuple. Without caching, the average bandwidth utilization for each node is approximately 50 KBps. Each query therefore incurs an average bandwidth cost of 0.1 KBps, an acceptable overhead for most current networks. (Of course, the precise cost of conducting POLYNOMIAL queries in other

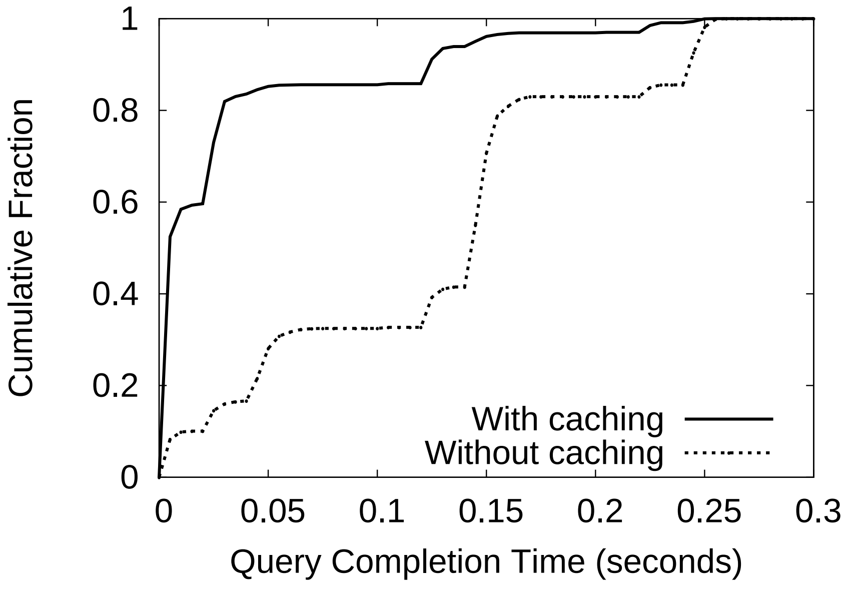


Figure 5.2: Cumulative distribution of query completion latencies with and without caching.

settings depends upon the provenance of the queried tuples.) The overhead imposed by POLYNOMIAL is due in part to its requirement that results must contain complete information regarding all possible tuple derivations.

As shown in Figure 5.1, POLYNOMIAL's overhead can be significantly reduced by enabling the caching optimization described in Section 5.2.1. Using caching, the average bandwidth utilization decreases to 20 KBps after two seconds. The performance improvement is attributed to the fact that queries are more likely to benefit from the cached results of previous queries as time progresses.

Figure 5.2 presents the cumulative fraction of query completion times. Regardless of whether caching is used, results are returned in less than 0.3 seconds, highlighting that NETTRAILS's provenance querying mechanisms are latency-wise efficient. The figure also shows the advantage of enabling caching: 80% of queries

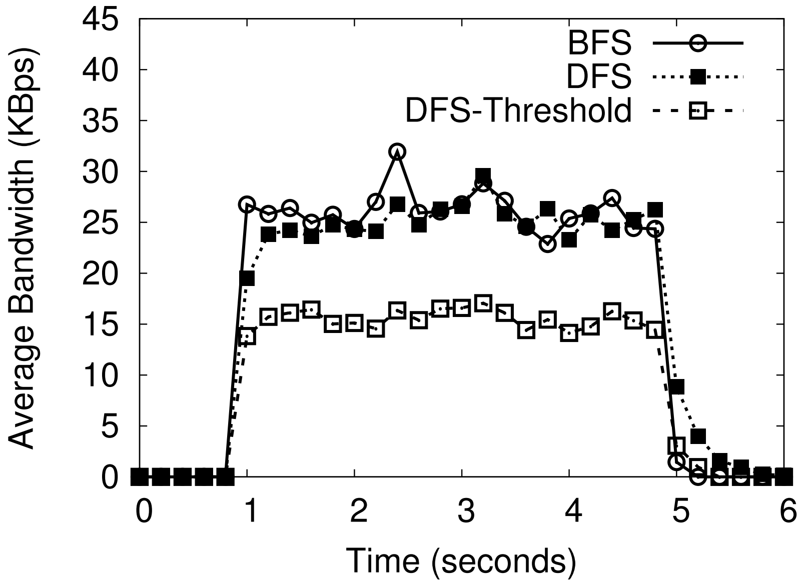


Figure 5.3: Average bandwidth cost (KBps) using different query traversal orders.

are returned within less than 50 ms if caching is enabled, a 67% improvement over query latency when caching is disabled.

Query Traversal Order To study the trade-offs between different query traversal orders, we conducted experiments in which nodes utilize the #DERIVATION query to determine whether a `minCost` tuple has more than three alternative derivations (the average number of alternative derivations for `mincost` is approximately three). The experiment is performed on three variants of the #DERIVATION query: (a) BFS, (b) DFS, and (c) DFS-THRESHOLD (DFS with threshold-based cutting). We use the same experimental setup as the caching experiment — that is, each node in the 100-node network issues five queries per second for randomly selected `mincost` tuples.

Figure 5.3 shows the average bandwidth consumption for different query traversal orders. As can be discerned from the figure, the bandwidth costs incurred

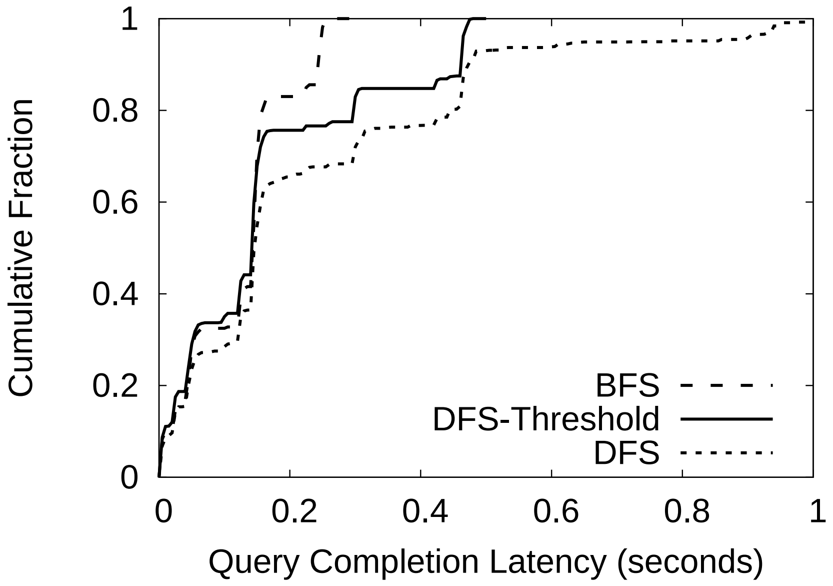


Figure 5.4: Cumulative distribution of query completion latencies using different query traversal orders.

by BFS and DFS are roughly equivalent (since both must traverse the entire provenance graph before a result is concluded). In contrast, DFS-THRESHOLD results in a 40% decrease in bandwidth consumption, due largely to its avoidance of a full traversal of provenance graphs for tuples with multiple derivations.

Figure 5.4 plots the cumulative distribution of query completion times for the query traversal strategies. Although the median latency is roughly equivalent for BFS and DFS, the latter experiences a long-tail distribution. For example, less than 80% of BFS queries complete within 0.16 seconds. In contrast, 80% of DFS queries require 0.45 seconds.

BFS's query completion is largely determined by the traversal depth in the provenance graph. Unlike BFS, DFS traverses alternative derivations in order, resulting in longer querying completion latencies. By terminating the query as soon as three derivations are explored, DFS-THRESHOLD avoids the long-tail distribu-

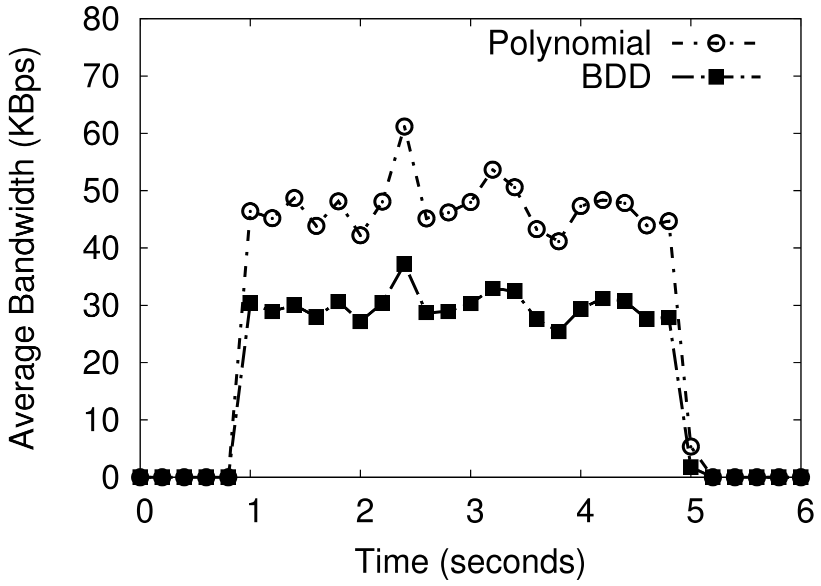


Figure 5.5: Average bandwidth (KBps) for querying POLYNOMIAL and BDD.

tion experienced by DFS. Using DFS-THRESHOLD, the query completion time for 80% of the queries decreases from 0.45 to 0.3 seconds.

Absorption Provenance We next compare the performance of the POLYNOMIAL and BDD queries. Figure 5.5 shows the average bandwidth incurred by the POLYNOMIAL and BDD queries. POLYNOMIAL incurs 18 KBps (57%) more bandwidth than BDD, due mainly to BDD's compact binary representation. As described in Section 5.2.3, BDD additionally decreases communication overhead by condensing provenance information using lossy compression.

POLYNOMIAL and BDD present near-identical performance when defined in terms of query completion latency. The latency of a query is largely decided by its traversal depth. Since both queries follow BFS query traversal order and operate on the same topology, the distributions of query completion latencies across nodes is consistent among the POLYNOMIAL and BDD queries.

5.4 Summary

In this chapter, we present two sets of optimization techniques. The first set studies the tradeoffs between the proactive and reactive provenance maintenance schemes. We develop a cost-based optimizer that takes as input the collected statistics that summarize the program characteristics, system performance and workload parameters. The decision on which maintenance scheme to adopt is made based on the cost estimation using these statistics. The second set of optimization techniques focuses solely on improving the query overhead in the cost of increased storage overhead (cache), delayed query latencies (DFS vs. BFS traversal order), and approximate query results (condensed provenance). We demonstrate the effectiveness of the proposed optimization techniques through a series of experimental evaluations performed on several example applications (such as network routing and cloud MapReduce computation).

Chapter 6

Secure Provenance

In the previous chapters, we have assumed a trusted environment, in which nodes are cooperative and correctly follow the provenance maintenance and querying protocols. However, distributed systems may be deployed across multiple administrative domains, where nodes may refuse to cooperate or even intentionally misbehave for various reasons, such as tensions between competing parties or malicious attacks.

In this chapter, we consider forensics in an *adversarial* setting, that is, we assume that a faulty node does not necessarily crash but can also change its behavior and continue operating. To be conservative, we assume that faults can be Byzantine [55], i.e., a faulty node can behave arbitrarily. This covers a wide range of faults and misbehavior, e.g., cases where a malicious adversary has compromised some of the nodes, but also more benign faults, such as hardware failures or misconfigurations. Getting correct answers to forensic queries in an adversarial setting is difficult because the misbehaving nodes can lie to the querier. For example, the adversary can attempt to conceal his actions by causing his nodes to fabricate plausible (but incorrect) responses to forensic queries, or he can attempt to frame correct nodes by returning responses that blame his own misbehavior on them.

Thus, the adversary can gain valuable time by misdirecting the operators and/or causing them to suspect a problem with the forensic system itself.

Existing solutions for forensics in an adversarial environment usually requires some trusted components, e.g., a trusted virtual-machine monitor [6, 51], a trusted host-level monitor [68], a trusted OS [72], or trusted hardware [14]. However, most components that are available today are not fully trustworthy; OSes and virtual machine monitors have bugs, which a powerful adversary could exploit, and even trusted hardware is sometimes compromised [49]. We argue that it is useful to have alternative techniques available that do not require this type of trust.

Towards this challenge, we introduce *Secure Time-aware Provenance* (STAP), a provenance system that can operate in a *completely untrusted* environment. We assume that the adversary may have compromised an arbitrary subset of the nodes, and that he may have complete control over these nodes. Despite the conservative threat model, a STAP system provides strong, provable guarantees: it ensures that an observable symptom of a fault or an attack can always be traced to a specific event — passive evasion or active misbehavior — on at least one faulty node, even when an adversary attempts to prevent this.

6.1 Threat Model and Assumptions

Since we would like to enable system administrators to investigate a wide range of problems, ranging from simple misconfigurations to hardware faults and even clandestine attacks, we conservatively assume Byzantine faults [55], i.e., that an adversary may have compromised an unknown subset of the nodes, and that he has complete control over them. Thus, the non-malicious problems are covered as a special case. We assume that the adversary can change both the primary system and the provenance system on these nodes, and he can read, forge, tamper with, or destroy any information they are holding. We also assume that no nodes or components of the system are inherently safe, i.e., system administrators do not a priori trust any node other than their own local machines.

6.1.1 Compromises

Ideally, we would like to correctly answer provenance queries even when the system is under attack. However, given our conservative threat model, this is not always possible. Hence, we make the following two compromises: first, we only demand that the system answer provenance queries about behavior that is *observable by at least one correct node* [40]; in other words, if some of the adversary's actions never affect the state of any correct node, the system is allowed to omit them. Second, we accept that the system may sometimes return an answer that is incorrect or incomplete, as long as the system administrator can a) tell which parts of the answer are affected, and she can b) learn the identity of at least one faulty node. In a forensic setting, this seems like a useful compromise: any unexpected behavior that can be noticed is observable by definition, and even a partial answer can help the system administrator determine whether a fault or misbehavior has occurred and which parts of the system have been affected.

6.1.2 Assumptions

In STAP, we make the following security assumptions:

1. An adversary cannot *indefinitely* prevent correct nodes from communicating;
2. Each node i has a keypair σ_i/π_i for signing messages, and verifying signatures by other nodes;
3. Faulty nodes cannot invert STAP's hash function or forge a signature by a correct node;

4. In the absence of an attack, messages are typically received within T_{prop};

5. The local clocks of the nodes are loosely synchronized to within Δ_{clock}; and

6. The execution of the application on each node is deterministic.

The first three assumptions are needed for the tamper-evident log. Assumption 2 could be satisfied by installing each node with a certificate signed by an offline CA; Assumption 3 is commonly assumed to hold for algorithms such as RSA and SHA-1. The next two assumptions are for simplicity; there are ways to build tamper-evident logs without them [41]. Both T_{prop} and Δ_{clock} can be large, e.g., on the order of seconds. The final assumption is needed to efficiently store and verify the provenance graph. This is also needed for some other security techniques, such as BFT [11], and it can be enforced for different types of applications [41], including legacy binaries [38].

6.2 Approach Overview

The definition of TAP in previous chapters assumes that, at least conceptually, the entire system execution $\mathcal{E}$ is known. However, in a distributed system without trusted components, no single node can have this information, especially when nodes are faulty and can tell lies. In this section, we present STAP, which constructs an approximation G_ν of the "true" provenance graph G that is based on information available to correct nodes.

6.2.1 Approximate Provenance using Evidence

Although each node can observe only its own local events, nodes can use messages from other nodes as *evidence* to reason about events on these nodes. We can require that messages be authenticated and acknowledged, such that each received message m is evidence of its own transmission. Once we discover inconsistencies from the input / output messages, the evidences can be used to tie faults to a particular node who is responsible for the inconsistency.

In addition, we can demand that nodes attach some additional information $\varphi(m)$, such as an explanation for the transmission of m. The validity of $\varphi(m)$ is checked against the expected execution logic and the evidence. For the purposes of this section, we will assume that $\varphi(m)$ describes the sender's entire execution prefix, i.e., all of its local events up to and including the transmission of m. Of course, this would be completely impractical; our implementation in Section 6.3.1 and 6.3.2 achieves a similar effect in a more efficient way.

When a provenance query is issued on a correct node, that node can collect some evidence $\bar{\mathcal{E}}$, such as messages it has locally received, and/or messages collected from other nodes. It can then use this evidence to construct an approximation $G_\nu(\bar{\mathcal{E}})$ of $G(\mathcal{E})$, from which the query can be answered.

We use the similar mechanisms presented in Chapter 4 to construct provenance from the evidence. In the construction of $G_\nu(\bar{\mathcal{E}})$, the legitimacy of the vertices depends on the evidence collected from the other nodes. We introduce a *color* for each vertex v in $G_\nu(\bar{\mathcal{E}})$, which is used to indicate whether v is legitimate: correct vertices are black, and faulty vertices are red. For example, if a faulty node N_i has no tuple τ derived during the execution, but nevertheless sends a message $+\tau$ to another node. $+\tau$ has no legitimate provenance, so we use the red color to represent transmission of $+\tau$. In Section 6.2.2, we will introduce a third color, yellow, for vertices whose true color is not yet known.

6.2.2 Fundamental Limitations

When faulty nodes are present, we cannot always guarantee that $G_\nu(\bar{\mathcal{E}}) = G(\mathcal{E})$. There are four fundamental reasons for this. First, $\varphi(m)$ can be incorrect; for example, a faulty node can tell lies about its local inputs. As a human investigator, one may be able to recognize such lies (so there is still value in displaying all the available information), but it is not possible to detect them automatically, since nodes cannot observe each other's inputs. Thus, the corresponding vertices do not appear red in G_ν. Note, however, that a faulty node cannot lie arbitrarily; for example, it cannot forge messages from other nodes.

Second, $\varphi(m)$ can be incomplete. For example, if two faulty nodes secretly exchange messages but otherwise act normally, we cannot guarantee that these messages will appear in G_ν because the correct nodes cannot necessarily obtain any evidence about them. We *can*, however, be sure that detectable faults [40] are represented in the graph. Briefly, a detectable fault is one that directly or indirectly affects a correct node through a message, or a chain of messages. Recall that, in our motivating scenario, we have assumed that Alice has observed some symptom of the fault; any fault of this type is detectable by definition.

Third, faulty nodes can equivocate, i.e., there can be two messages m_1 and m_2 such that $\varphi(m_1)$ is inconsistent with $\varphi(m_2)$. If a correct node encounters both m_1 and m_2, it can detect the inconsistency, but it is not clear which of them (if any) is correct and should appear in G_ν. One approach is to liberally mark as incorrect each vertex that is involved in an inconsistency. However, this can lead to an excessive number of incorrect vertices on equivocating nodes, which limits the usefulness of G_ν. Another approach, which we adopt here, is to arbitrarily accept one of the explanations as true, e.g., the one that appears first.

Finally, if φ is evaluated on demand, $\varphi(m)$ can be unavailable. For example, a correct node that is trying to evaluate a provenance query on $\bar{\mathcal{E}}$ might ask the sender of some $m \in \bar{\mathcal{E}}$ for $\varphi(m)$ but might not receive a response. This situation is ambiguous and does not necessarily indicate a fault – for example, the queried node could be slow, or the response could be delayed in the network. However, the only way to avoid it reliably would be to proactively attach $\varphi(m)$ to every message, which would be prohibitively expensive. Instead, STAP uses a third color (yellow) for vertices whose color is not yet known. Yellow vertices turn black or red when the response arrives. If a vertex v remains yellow, this is a sign that $\text{HOST}(v)$ is refusing to respond and is therefore faulty.

6.2.3 Definition: STAP

Based on the intuition presented in Section 6.3, we give the definition of STAP, which is formulated based on the following properties.

Definition 8. ***(Monotonicity)*** *An approximation $G_\nu(\bar{\mathcal{E}})$ of $G(\mathcal{E})$ is* monotonic *if $G_\nu(\bar{\mathcal{E}})$ is a subgraph of $G_\nu(\bar{\mathcal{E}} + \bar{\mathcal{E}}')$ for additional evidence $\bar{\mathcal{E}}'$.*

Definition 9. ***(Accuracy)*** *$G_\nu(\bar{\mathcal{E}})$ is* accurate *if it faithfully reproduces all the vertices on correct nodes; in other words, if a vertex v on a correct node appears in $G_\nu(\bar{\mathcal{E}})$ then v must also exist in $G(\mathcal{E})$, be colored black, and have the same predecessors and successors.*

Definition 10. ***(Completeness)*** *$G_\nu(\bar{\mathcal{E}})$ is* complete *if, given sufficient evidence $\bar{\mathcal{E}}$ from the correct nodes, a) each vertex in $G(\mathcal{E})$ on a correct node also appears in $G_\nu(\bar{\mathcal{E}})$, and b) for each detectably faulty node, $G_\nu(\bar{\mathcal{E}})$ contains at least one red or yellow vertex.*

Monotonicity is an important property because it prevents G_ν from changing fundamentally once additional evidence becomes available, which could invalidate responses to earlier queries. Accuracy and completeness properties give guarantees that a correct node will never be falsely accused and that a detectably faulty node will be eventually detected.

Based on the above three properties, we define STAP as follows:

Definition 11. *Given an execution trace $\mathcal{E}$, we define STAP to be a monotonic approximation $G_\nu(\bar{\mathcal{E}})$ of the provenance graph $G(\mathcal{E})$ that is both complete and accurate in an untrusted setting.*

6.3 Secure Maintenance and Querying

In this section, we present the security enhancement to the provenance maintenance and querying for implementation a STAP system.

6.3.1 Secure Logging for Provenance Maintenance

Recall from Chapter 3 that provenance graph $G = (V, E)$ is designed so that each vertex $v \in V$ can be attributed to a specific node $\textsc{host}(v)$. Thus, we can partition the graph so that each $v \in V$ is stored on $\textsc{host}(v)$. To ensure accuracy, we must additionally keep evidence for each cross-node edge, i.e., $(v_1, v_2) \in E$ with $\textsc{host}(v_1) \neq \textsc{host}(v_2)$. Specifically, $\textsc{host}(v_1)$ must be able to prove that $\textsc{host}(v_2)$ has committed to v_2, and vice versa, so that each node can prove that its own vertex is legitimate, even if the other node is compromised. Finally, due to Assumption 6, each node's subgraph of G is completely determined by its inputs and outputs; hence, it is sufficient to store messages and changes to base tuples. When necessary, the microquery module can reconstruct G from this information.

Logs and Authenticator Sets. STAP's log is a simplified version of the log from PeerReview [41]. The log λ_i of a node i consists of entries of the form $e_k := (t_k, y_k, c_k)$, where t_k is a timestamp, y_k is an entry type, and c_k is some type-specific content. There are five types of entries: SND and RCV record messages, ACK records acknowledgments, and INS and DEL record base tuple insertions and deletions. Note that log entries are different from vertex types. Each entry is associated with a hash value $h_k = H(h_{k-1} \,||\, t_k \,||\, y_k \,||\, c_k)$ with $h_0 := 0$, where $H(\cdot)$ is a cryptographic hash function. Together, the h_k form a hash chain. A node i can issue an *authenticator* $a_k := (t_k, h_k, \sigma_i(t_k \,||\, h_k))$; $\sigma_i(\cdot)$ denotes a signature with i's key. An authenticator is a signed commitment that e_k (and, through the hash chain, $e_1, \ldots, e_{k-1}$) must exist in i's log. Each node i stores the authenticators it receives from other nodes in its *authenticator set* A_i.

Commitment. When a node i needs to send a message $+/-\ \tau$ to another node j, it first appends a new entry $e_x := (t_x, \textsc{snd}, (+/-\ \tau, j))$ to its local log. Then it sends $(+/-\ \tau, h_{x-1}, t_x, \sigma_i(t_x \,||\, h_x))$ to j. When a node j receives a message $(+/-\ \tau, a, b, c)$, j calculates $h'_x := H(a \,||\, b \,||\, \textsc{snd} \,||\, (+/-\ \tau, j))$ and then checks whether the authenticator is properly signed, i.e., $\pi_i(c) = (b \,||\, h'_x)$, and whether t_x is within $\Delta_{clock} + T_{prop}$ of its local time. If not, j discards the message. Otherwise, j adds (t_x, h'_x, c) to its authenticator set $A_{j,i}$, appends an entry $e_y := (k, \textsc{rcv}, (+/-\ \tau, i, a, b, c))$ to its own log, and sends $(\textsc{ack}, t_x, h_{y-1}, t_y, \sigma_j(t_y \,||\, h_y))$ back to i.

Once i receives $(\textsc{ack}, a, b, c, d)$ from j, it first checks its log to see whether there is an entry $e_x = (a, \textsc{snd}, (+/-\ \tau, j))$ in its log that has not been acknowledged yet. If not, it discards the message. i then calculates $h'_y := H(b \,||\, c \,||\, \textsc{rcv} \,||\, (+/-\ \tau, i, h_{x-1}, t_x, \sigma_i(t_x \,||\, h_x)))$, and checks whether $\pi_j(d) = (c \,||\, h'_y)$ and t_y is within $\Delta_{clock} + T_{prop}$ of its local time. If not, i discards the message. Otherwise, i adds (c, h'_y, d) to its authenticator set $A_{i,j}$ and appends an entry $e_z := (t, \textsc{ack}, a, b, c, d)$ to its log. If i does not receive a valid acknowledgment within $2 \cdot T_{prop}$, it immediately notifies the administrator of this.

Retrieval. The provenance maintenance module implements a primitive $\textsc{retrieve}(v, a^i_k)$ which, when invoked on $i := \textsc{host}(v)$ with a vertex v and an authenticator a^i_k of i, returns the prefix of the log in which v was generated. Typically, this is the prefix authenticated by a^i_k, but if v is an EXIST vertex that exists at e_k, the prefix is extended to either a) the point where v ceases to exist, or b) the current time. If the prefix extends beyond e_k, i must also return a new authenticator that covers the entire prefix. A correct node can always comply with such a request.

6.3.2 Secure Provenance Querying

STAP adopts a similar distributed recursive querying framework as TAP. To construct provenance in a secure manner, it uses a special primitive called MICROQUERY to navigate a STAP graph.[1] MICROQUERY has two arguments: a vertex v, and evidence $\bar{\mathcal{E}}$ such that $v \in G_\nu(\bar{\mathcal{E}})$. MICROQUERY returns one or two *color notifications* of the form BLACK(v), YELLOW(v), or RED(v). If two notifications are returned, the first one must be YELLOW(v). MICROQUERY can also return two sets P_v and S_v that contain the predecessors and successors of v in $G_\nu(\bar{\mathcal{E}})$, respectively. Each set consists of elements (v_i, e_i), where $\bar{\mathcal{E}}_i$ is additional evidence such that v_i and the edge between v_i and v appear in $G_\nu(\bar{\mathcal{E}} + e_i)$; this makes it possible to explore all of G_ν by invoking MICROQUERY recursively.

The microquery module implements MICROQUERY(v, e), and uses the information in this log to implement MICROQUERY; it uses authenticators as a specific form of evidence. At a high level, this works by 1) using e to retrieve a log prefix from HOST(v), 2) replaying the log to regenerate HOST(v)'s partition of the provenance graph G, and 3) checking whether v exists in it. If v exists and was derived correctly, its predecessors and successors are returned, and v is colored black; otherwise v is colored red.

More formally, the evidence for a vertex v is an authenticator from HOST(v) that covers a log prefix in which v existed. When MICROQUERY(v, e) is invoked on a node i, i first outputs YELLOW(v) to indicate that v's real color is not yet known, and then invokes RETRIEVE(v, e) on $j :=$ HOST(v). If j returns a log prefix that matches e, i replays the prefix to regenerate j's partial provenance subgraph $\bar{G}_j$. If this subgraph does not contain v or replay fails (i.e., the sent messages do not match the SEND entries in the log, a SEND does not have a matching ACK, or the authenticators in the RECV and ACK entries do not satisfy the conditions from Section 6.3.1, i outputs

[1]MICROQUERY returns a single vertex; provenance queries must invoke it repeatedly to explore G_ν. Hence the name.

RED(v); otherwise it outputs BLACK(v) and returns the predecessors and successors of v in $\bar{G}_j$. The evidence for a SEND predecessor and a RECEIVE successor are the authenticators from the RCV and ACK entries, respectively; the evidence for all other vertices is the authenticator returned by RETRIEVE, if any, or otherwise e.

As described so far, the algorithm colors a vertex v red for which HOST(v) does not have a correct 'explanation' (in the form of a log prefix), and it colors v yellow if HOST(v) does not return any explanation at all. The only remaining case is the one in which v's explanation is inconsistent with the explanation for one of its other vertices. To detect this, i performs the following check: it determines the interval I during which v existed during replay, and asks all nodes with which j could have communicated during I (or simply all other nodes) to return any authenticators that were a) signed by j, and b) have timestamps in I. If such authenticators are returned, i checks whether they are consistent with the log prefix it has retrieved earlier; if not, i outputs RED(v).

6.3.3 Optimizations

As described so far, each NETTRAILS node cryptographically signs every single message and keeps its entire log forever, and each microquery retrieves and replays an entire log prefix. Most of the corresponding overhead can be avoided with a few simple optimizations.

First, nodes can periodically record a checkpoint of their state in the log, which must include a) all currently extant or believed tuples and b) for each tuple, the time when it appeared. Thus, it is sufficient for MICROQUERY(v, ϵ) to retrieve the log segment that starts at the last checkpoint before v appeared, and start replay from there. Note that this does not affect correctness because, if a faulty node adds a nonexistent tuple τ to its checkpoint, this will be discovered when the corresponding EXIST or BELIEVE vertex is queried, since replay will then begin before the checkpoint and end after it. If the node omits an extant or believed tuple that affects a queried tuple, this will cause replay to fail.

Second, nodes can be required to keep only the log segment that covers the most recent T_{hist} hours in order to decrease storage costs. To speed up queries, the querier can cache previously retrieved log segments, authenticators, and even previously regenerated provenance graphs. As we show in Section 6.5, this reduces the overhead to a practical level.

Third, the overhead of the commitment protocol can be reduced by sending messages in batches. This can be done using a variant of Nagle's algorithm that was previously used in NetReview [39]: each outgoing message is delayed by a short time T_{batch}, and then processed together with any other messages that may have been sent to the same destination within this time window. Thus, the rate of signature generations/verifications is limited to $1/T_{batch}$ per destination, regardless of the number of messages. The cost is an increase in message latency by T_{batch}.

6.4 Correctness

Next, we argue that, given our assumptions from Section 6.1.2, NETTRAILS provides secure network provenance as defined in Section 6.2.3 — that is, monotonicity, accuracy, and completeness. Here we present the theorems and proof sketches; the complete proofs can be found in a technical report [110].

Theorem 1. NETTRAILS *is monotonic: if* ϵ *is a set of valid authenticators and* a_k^i *a valid authenticator,* $G_\nu(\epsilon)$ *is a subgraph of* $G_\nu(\epsilon + a_k^i)$.

Proof sketch. There are four cases we must consider. First, the new authenticator a_k^i can be the first authenticator from node i that the querying node has seen. In this case, the querying node will RETRIEVE the corresponding log segment, replay it, and add the resulting vertices to G_ν. Since the graph construction is compositional, this can only add to the graph, and the claim holds. Second, a can belong to a log segment NETTRAILS has previously retrieved; in this case, G_ν already contains the corresponding vertices and remains unchanged. Third, a can correspond to an extension of an existing log segment. In this case, the additional events are replayed and the corresponding vertices added, and the claim follows because the graph construction is compositional and incremental. Finally, a's log segment can be inconsistent with an existing segment; in this case, the consistency check will add a red SEND vertex to G_ν. □

Theorem 2. NETTRAILS *is accurate: any vertex v on a correct node that appears in $G_\nu(\epsilon)$ must a) also appear in G, with the same predecessors and successors, and b) be colored black.*

Proof sketch. Claim a) follows fairly directly from the fact that $i := \text{HOST}(v)$ is correct and will cooperate with the querier. In particular, i will return the relevant segment of its log, and since the graph construction is deterministic, the querier's replay of this log will faithfully reproduce a subgraph of G that contains v. Any predecessors or successors v' of v with $\text{HOST}(v') = i$ can be taken from this subgraph. This leaves the case where $\text{HOST}(v') \neq v$. If v' is a predecessor, then it must be a SEND vertex, and its existence can be proven with the authenticator from the corresponding SND entry in λ. Similarly, if v' is a successor, then it must be a RECV vertex, and the evidence is the authenticator in the corresponding ACK entry in λ.

Now consider claim b). Like all vertices, v is initially yellow, but it must turn red or black as soon as $i := \text{HOST}(v)$ responds to the querier's invocation of RETRIEVE, which will happen eventually because i is correct. However, v can only turn red for a limited number of reasons—e.g., because replay fails, or because i is found to have tampered with its log—but each of these is related to some form of misbehavior and cannot have occurred because i is correct. Thus, since v cannot turn red and cannot remain yellow, it must eventually turn (and remain) black. □

Theorem 3. NETTRAILS *is complete: given sufficient evidence ϵ from the correct nodes, a) each vertex in G on a correct node also appears in $G_\nu(\epsilon)$, and b) when some node is detectably faulty, recursive invocations of* MICROQUERY *will eventually yield a red or yellow vertex on a faulty node.*

Proof sketch. Claim a) follows if we simply choose ϵ to include the most recent authenticator from each correct node, which the querying node can easily obtain. Regarding claim b), the definition of a detectable fault implies the existence of a chain of causally related messages such that the fault is apparent from the first message and the last message m is received by a correct node j. We can choose v' to be the RECV vertex that represents m's arrival. Since causal relationships correspond to edges in G, G_ν must contain a path $v' \rightarrow^* v$. By recursively invoking MICROQUERY on v' and its predecessors, we retrieve a subgraph of G_ν that contains this path, so the vertices on the path are queried in turn. Now consider some vertex v'' along the path. When v'' is queried, we either obtain the next vertex on the path, along with valid evidence, or v'' must turn red or yellow. Thus, either this color appears before we reach v, or we eventually obtain evidence of v. □

6.5 Evaluation

In this section, we evaluate NETTRAILS using three applications in a total of five different scenarios. Since we have already proven the correctness of the STAP algorithm in [110], we focus mostly on overheads and performance. Specifically, our goal is to answer the following high-level questions: (i) can NETTRAILS answer useful forensic queries? (ii) how much overhead does NETTRAILS incur at runtime? and (iii) how expensive is it to ask a query?

6.5.1 Experimental Setup

We examine NETTRAILS's performance across five application configurations: Quagga [82] routing deployment, two Chord [99] installations, and two Hadoop [37] clusters. We describe how NETTRAILS interacts with each application in greater details in Section 7.4.

Our **Quagga** experiment is modeled after the setup used for NetReview [39]: We instantiated 35 unmodified Quagga daemons, each with the NETTRAILS proxy from Section 7.4.3, on an Intel machine running Linux 2.6. The daemons formed a topology of 10 ASes with a mix of tier-1 and small stub ASes, and both customer/provider and peering relationships; the internal topology was a full iBGP mesh. To ensure that both the BGP traffic and the routing table sizes are realistic, we injected approximately 15,000 updates from a RouteViews [90] trace. The length of the trace, and thus the duration of the experiment, was 15 minutes. In all experiments, each node was configured with a 1024-bit RSA key.

We evaluated the Chord prototype (Section 7.4.1) in two different configurations: **Chord-Small** contains 50 nodes and **Chord-Large** contains 250 nodes. The experiments were performed in simulation, with stabilization occurring every 50 seconds, optimized finger fixing every 50 seconds, and keep-alive messages every 10 seconds. Simulation ran for 15 minutes of simulated time.

In the **Hadoop-Small** experiment, we ran the prototype described in Section 7.4.2 on 20 c1.Medium instances on Amazon EC2 (in the `us-east-1c` region). The program we used (WordCount) counts the number of occurrences of each word in a 1.2 GB Wikipedia subgraph from WebBase [104]. We used 20 mappers and 10 reducers; the total runtime was about 79 seconds. Our final experiment, **Hadoop-Large**, used 20 c1.medium instances with 165 mappers, 10 reducers, and a 10.3 GB data set that consisted of the same Wikipedia data plus the

12/2010 Newspapers crawl from WebBase [104]; the runtime for this was about 255 seconds.

Quagga, Chord, and Hadoop have different characteristics that enable us to study NETTRAILS under varying conditions. For instance, Quagga and Chord have small messages compared to Hadoop, while Quagga has a large number of messages. In terms of rate of system change, Quagga has the highest, with approximately 1350 route updates per minute. In all experiments, the actual replay during query evaluation was carried out on an Intel 2.66GHz machine running Linux with 8GB of memory.

6.5.2 Example Queries

To evaluate NETTRAILS's ability to perform a variety of forensic tasks as well as to measure its query performance, we tested NETTRAILS using the following queries, each of which is motivated by a problem or an attack that has been previously reported in the literature:

Quagga-Disappear is a dynamic query that asks why an entry from a routing table has disappeared. In our scenario, the cause is the appearance of an alternative route in another AS j, which replaces the original route in j but, unlike the original route, is filtered out by j's export policy. This is modeled after a query motivated in the Omni paper [101]. Note that, unlike Omni, NETTRAILS works even when nodes are compromised. **Quagga-BadGadget** query asks for the provenance of a 'fluttering' route; the cause is an instance of BadGadget [34], a type of BGP configuration problem.

Chord-Lookup asks which nodes and finger entries were involved in a given DHT lookup, and **Chord-Finger** returns the provenance of a given finger table entry. Together, these two queries can detect an Eclipse attack [96] in which most of a node's lookups are routed through a small set of nodes controlled by the adversary.

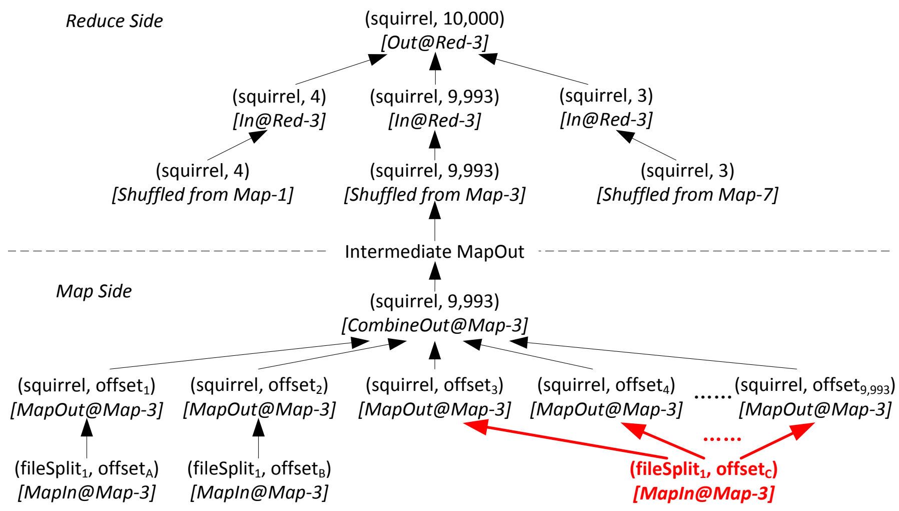


Figure 6.1: Example result (with simplified notations) of the Hadoop-Squirrel query.

Hadoop-Squirrel asks for the provenance of a given key-value pair in the output; for example, if the WordCount application produces the (unlikely) output `(squirrel,10000)` to indicate that the word "squirrel" appeared 10,000 times in the input, this could be due to a faulty or compromised mapper. Such queries are useful to investigate computation results on outsourced Cloud databases [80].

6.5.3 Usability

In addition to the formal guarantees in Section 6.2.3, we also need to demonstrate that NETTRAILS is a useful forensic tool *in practice*. For this purpose, we executed each of the above queries twice – once on a correct system and once on a system into which we had injected a corresponding fault. Specifically, we created an instance of BadGadget in BGP routing, we modified a Chord node to always return its own ID in response to lookups, and we tampered with a Hadoop map worker so it would produce inaccurate results. For all queries, NETTRAILS clearly identified the source of the fault.

To illustrate a specific example query in detail, Figure 6.1 shows the output of the Hadoop-Squirrel query, where one of the Mappers (i.e. Map-3) is configured to misbehave – in addition to emitting a `(word, offset)` tuple for each word in the text, it injects 9,991 additional `(squirrel, offset)` tuples (shown in red). A forensic analyst who is suspicious about the enormous prevalence of squirrels in this dataset can use NETTRAILS to query the provenance of the `(squirrel, 10,000)` output tuple. NETTRAILS responds by selectively reconstructing the provenance subgraph of the corresponding reduce task. Seeing that one mapper output $9,993$ squirrels while the others only reported 3 or 4, she can "zoom in" further by requesting the provenance of the `(squirrel, 9,993)` tuple, at which point NETTRAILS reconstructs the provenance subgraph of the corresponding map task. This reveals two legitimate occurrences and lots of additional bogus tuples, which are colored red.

Once the faulty tuples are identified, NETTRAILS can be used to determine their effects on the rest of the system, e.g., to identify other outputs that may have been affected by key-value pairs from the corrupted map worker.

In this example, the analyst repeatedly issues queries with a small scope and inspects the results before deciding which query to issue next. This matches the usage pattern of provenance visualization tools, such as the VisTrails visualizer [10], which allow the analyst to navigate the provenance graph by expanding and collapsing vertices. The analyst could also use a larger scope directly, but this would cause more subgraphs to be reconstructed, and most of the corresponding work would be wasted because the analyst subsequently decides to investigate a different subtree.

6.5.4 Network Traffic at Runtime

NETTRAILS increases the network traffic of the primary system because messages

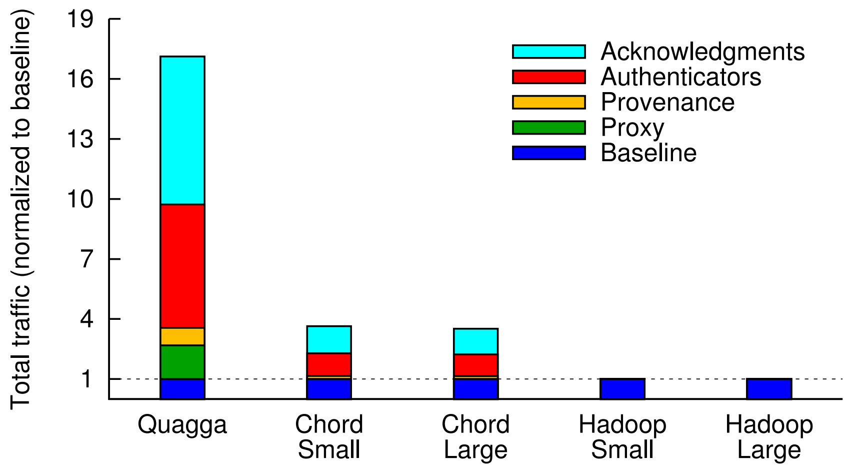


Figure 6.2: Normalized increase in traffic by NETTRAILS, compared to a baseline system (no provenance).

must contain an authenticator and be acknowledged by the recipient. To quantify this overhead, we ran all five experiments in two configurations. In the *baseline* configuration, we run the original Hadoop, Quagga, or declarative Chord in RapidNet with no support for provenance. In the NETTRAILS-enabled prototype, we measured the additional communication overhead that NETTRAILS adds to the baseline, broken down by cause, i.e. authenticators, provenance, proxy, etc.

Figure 6.2 shows the NETTRAILS results, normalized to the baseline results. The overhead ranges between a factor of 17.1 for Quagga and 1.002 for Hadoop. The differences are large because NETTRAILS adds a fixed number of bytes for each message – 22 bytes for a timestamp and a reference count, 156 bytes for an authenticator, and 187 bytes for an acknowledgment. Since the average message size is small for Quagga (68 bytes) and very large for Hadoop (1.08 MB), the relative overhead for Quagga is higher, although in absolute terms, the Quagga traffic is still low (78.2 Kbps with NETTRAILS). Chord messages are 145 bytes on average, and hence its overhead factor is in between Quagga and Hadoop.

The Quagga proxy causes some additional overhead because, unlike the original BGP implementation in Quagga, it does not combine BGP announcements and withdrawals into a single message; this is not an inherent overhead of NETTRAILS, and could mostly be optimized away with an efficient proxy that batches tuples.

In summary, NETTRAILS adds a constant number of bytes to each message, so its relative overhead depends on the primary system's average message size.

6.5.5 Storage

Each NETTRAILS node requires some local storage for the graph recorder's log. To quantify this cost, we ran our five experiments in the NETTRAILS configuration and measured the size of the resulting logs.

Figure 6.3 shows the average amount of log data that each node produced per minute, excluding checkpoints. In absolute terms, the numbers are relatively low; they range from 0.066 MB/min (Chord-Small) to 0.74 MB/min (Quagga). We expect that most forensic queries will be about fairly recent events, e.g., within one week. To store one week's worth of data, each node would need between 7.3 GB (Quagga) and 665 MB (Chord-Small). Note that, in contrast to proactive detection systems like PeerReview [41], this data is merely archived locally at each node and is only sent to the query node during replay.

The log contains copies of all the received messages (for Hadoop, references to files), authenticators for each sent and received message, and acknowledgments. Thus, log growth depends both on the number of messages and their size distribution. As a result, in Figure 6.3, we note that Quagga incurs the highest log size, given that its baseline system generates the largest number of messages. Interestingly, the overhead of Hadoop is extremely low (less than 0.1 MB/minute), due to the optimizations described in Section 7.4.2 where the proxy only logs small control messages. In addition to the log itself, NETTRAILS must keep the Hadoop input files (1.2 GB for Small, 10.3 GB for Large) to be able to replay tasks if neces-

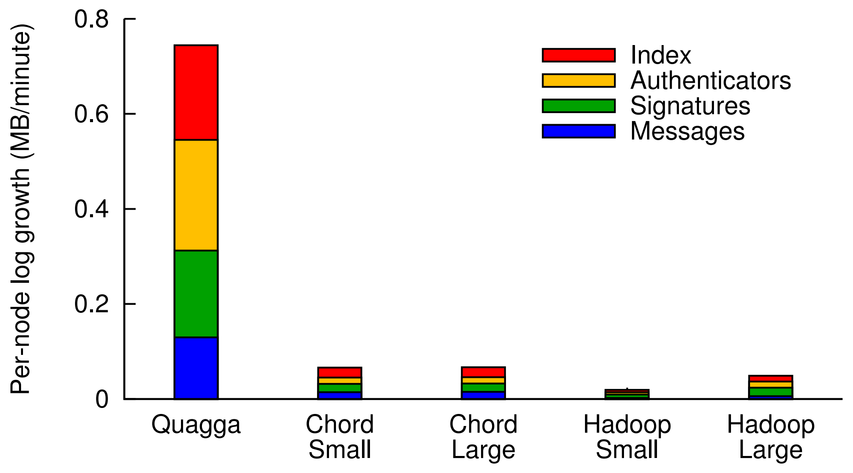


Figure 6.3: Per-node log growth in NETTRAILS, excluding checkpoints.

sary. We do not include this in Figure 6.3 because the input files are already kept by default in Hadoop, unless the user explicitly deletes them.

NETTRAILS can additionally keep checkpoints of the system state. The size of a typical checkpoint is 25 KB for Chord and 64 MB for Quagga; since replay starts at checkpoint, more checkpoints result in faster queries but consume more space. For Hadoop, the equivalent of a checkpoint is to keep the intermediate files that are produced by the Map tasks, which requires 207 MB for Small and 682 MB for Large.

6.5.6 Computation

We next measure the additional overhead imposed by NETTRAILS. The dominating additional computation overhead is due to NETTRAILS's graph recorder, which requires additional CPU processing at runtime for generating and verifying signatures and, in the case of Hadoop, for hashing the input and output files (see Section 7.4.2). The exact load is difficult to measure directly, but we estimated it from (1) the number of signatures generated and verified, as well as the amount of data hashed, and (2) benchmarks (`openssl speed`) on the target machines.

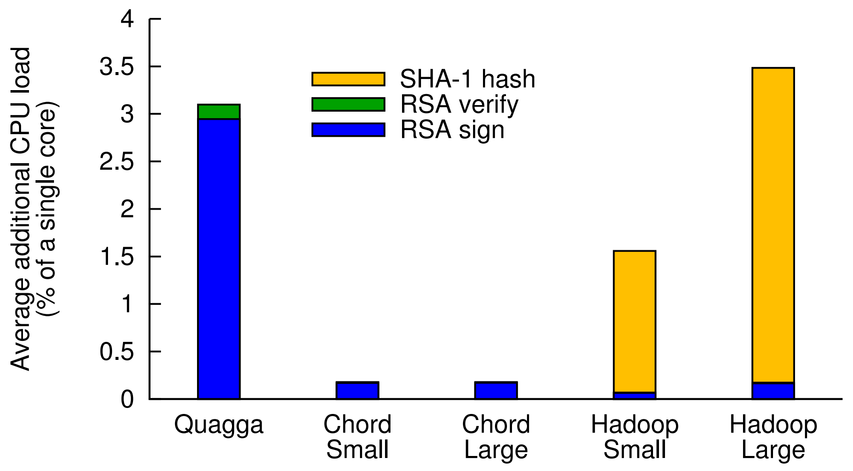


Figure 6.4: Additional CPU load for generating and verifying signatures, and for hashing.

Specifically, we benchmark the time taken for signature generation/verification and hashing on a single core, and multiply that by the total number of such operations. Based on this subtotal, we can then compute the additional CPU load, averaged over the total execution time.

Figure 6.4 shows our estimates for the average additional CPU load caused by these operations. The results show that the average additional CPU load is below 4% in all three applications. For Quagga and Chord, the increase is dominated by the signatures, of which two are required for each message – one for the authenticator and the other for the acknowledgment. Hadoop sends very few messages (one from each mapper to each reducer) but handles large amounts of data, which for NETTRAILS must be hashed for commitment. Note that we do not include I/O cost for the hashed data because the data would have been written by the unmodified Hadoop as well; NETTRAILS merely adds a SHA-1 hash operation, which can be performed on-the-fly as the data is written.

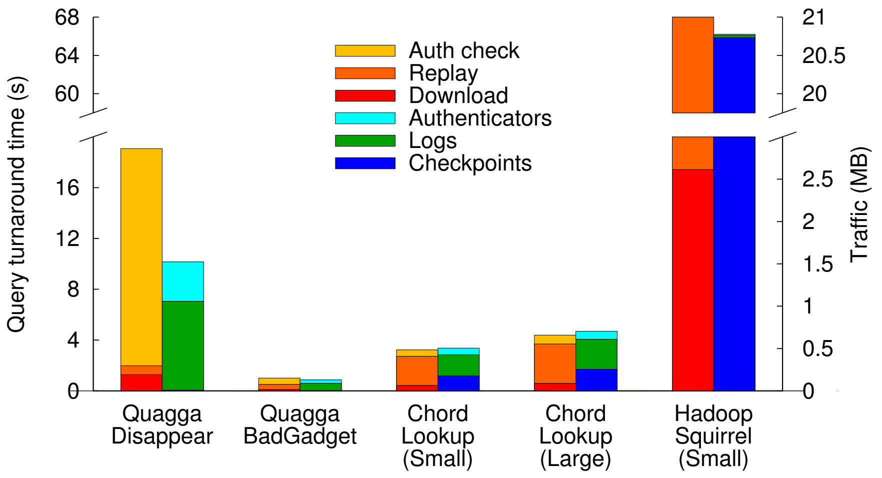


Figure 6.5: Query cost: time elapsed (left bar) and data downloaded (right bar).

6.5.7 Query Performance

Next, we evaluate how quickly NETTRAILS can answer queries, and how much data needs to be downloaded. Since the answer depends on the query, we performed several different queries in different systems. For each query, we measured 1) how much data (log segments, authenticators, and checkpoints) was downloaded, 2) how long it took to verify the log against the authenticators, and 3) how much time was needed to replay the log and to extract the relevant provenance subgraph. Figure 6.5 shows our results. Note that the query turnaround time includes an estimated download time, based on an assumed download speed of 10 Mbps.

The results show that both the query turnaround times and the amount of data downloaded can vary considerably with the query. The Chord and Quagga-BadGadget queries were completed within less than five seconds; the Quagga-Disappear query took 19 seconds, where 14 were used to verify partial checkpoints (using a Merkle Hash Tree); and the Hadoop-Squirrel query required 68 seconds, including 51 for replay. The download varied between 133 kB for Quagga-BadGadget and 20.8 MB for Hadoop-Squirrel. The numbers for Hadoop are larger

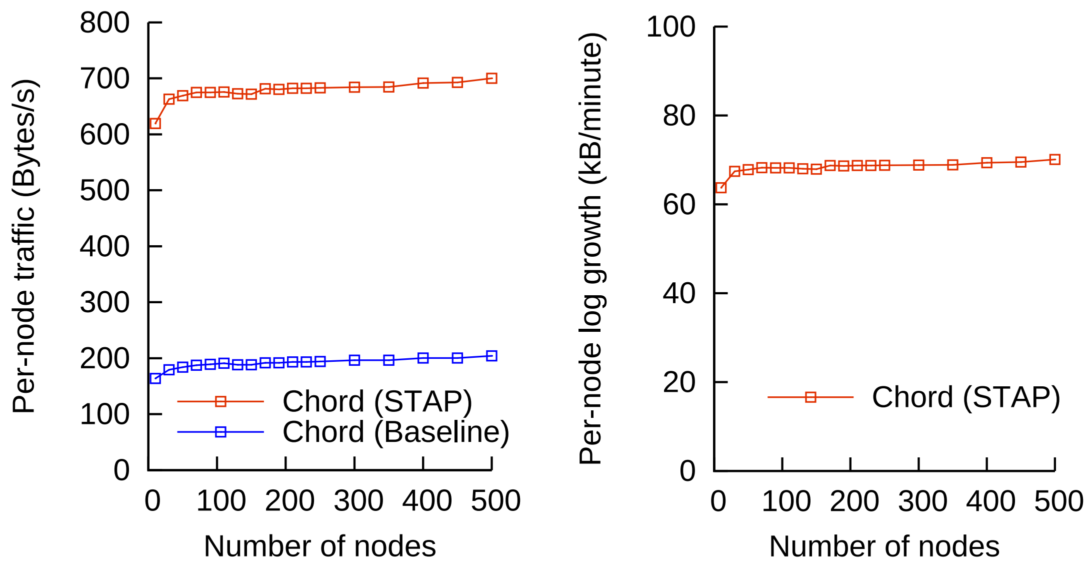


Figure 6.6: Scalability for Chord: total traffic (left) and log size (right).

because our prototype does not create checkpoints *within* map or reduce tasks, and so must replay a node's entire task to reconstruct a vertex on that node. Fine-grain checkpoints could be added but would require more changes to Hadoop. Generally, there is a tradeoff between storage and query performance: finer-grain checkpoints require more storage but reduce the size of the log segments that need to be downloaded and replayed.

In summary, the downloads and query turnaround times vary between queries, but generally seem low enough to be practical for interactive forensics.

6.5.8 Scalability

In our final experiment, we examine how NetTrails's overhead scales with the number of nodes N. We ran our Chord experiment with a range of different system sizes between $N = 10$ and $N = 500$ nodes, and we measured two of the main overheads, traffic and log size, for each N. Figure 6.6 shows our results, plus the baseline traffic for comparison.

The results show that both overheads increase only slowly with the system size. This is expected because, as discussed in Sections 6.5.4 and 6.5.5, the overhead is a function of the number and size of the messages sent. If the per-node traffic of the application did not depend on N, the runtime overhead would not depend on N either; however, recall that Chord's traffic increases with $O(\log N)$, as illustrated here by the baseline traffic results, so the NETTRAILS overheads in this experiment similarly grow with N.

Note the contrast to accountability systems such as PeerReview [41] where the overhead itself grows with the system size. This is because PeerReview uses witnesses to ensure that each pair of authenticators from a given node is seen by at least one correct node. NETTRAILS relies on the querier's node for this property (see Section 6.3.2) and, as a forensic system, it does not audit proactively.

In summary, NETTRAILS does not reduce the scalability of the primary system; its per-node overheads mainly depend upon the number of messages sent.

6.6 Summary

This chapter introduces STAP's security enhancement for securely constructing provenance graphs in untrusted environments with Byzantine faults. Since faulty nodes can tell lies or suppress information, a STAP system cannot always determine the exact provenance of a given system state (or state change), but it can approximate it and give strong, provable guarantees on the quality of the approximation. To demonstrate that STAP is practical, we evaluate our NETTRAILS system with three different example applications: the Quagga BGP daemon, a declarative implementation of Chord, and Hadoop MapReduce. Our results show that the maintenance costs vary with the application but are low enough to be practical.

We also describe several example queries that can be used to investigate attacks previously reported in the literature, and NETTRAILS can answer them within a few seconds.

Chapter 7

Implementation: NETTRAILS System

In the previous chapters, we have focused on the provenance maintenance and querying techniques, and the security treatment for provenance support in untrusted environments, which are the fundamental and core components in a provenance system. As demonstrated in Section 6.5, such a system allows system operators to handle a variety of forensic tasks. In this chapter, we present in greater detail NETTRAILS, the implementation of our proposed secure time-aware provenance (STAP) system. We discuss how each component presented in this dissertation is integrated into a coherent framework (Section 7.1), and present the techniques that further improve the usability of NETTRAILS. In particular, we discuss the following two aspects: 1) how the execution logic of a primary system can be extracted as derivations rules (Section 7.2); and 2) how operators can interactively retrieve the desired information (related to answering a forensic question) from a vast amount of provenance data (Section 7.3). We demonstrate the wide applicability of NETTRAILS by showing three example applications (Section 7.4), and we conclude this chapter by summarizing the limitations of the NETTRAILS system (Section 7.5).

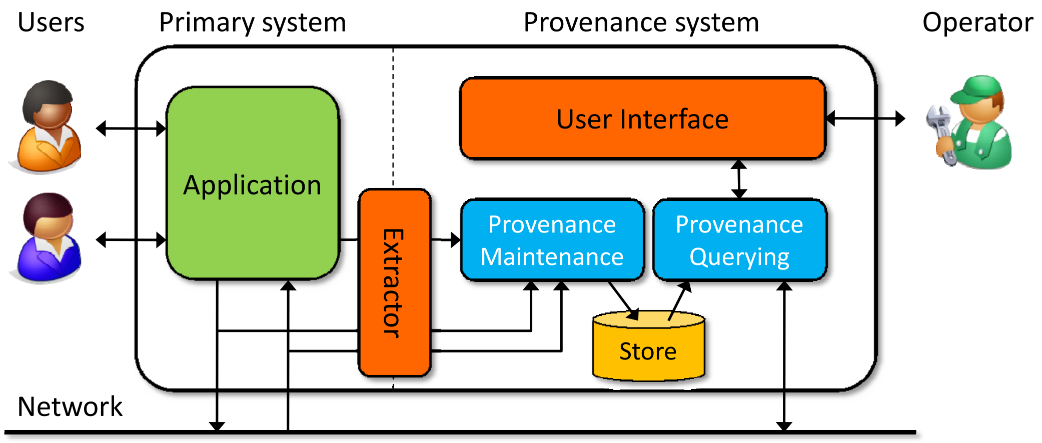


Figure 7.1: The architecture of a single node.

7.1 Architecture Overview

The provenance system consists of three major building blocks, a *provenance maintenance module*, a *provenance querying module*, and a *user interface* that permits easy access to the provenance information (Figure 7.1). An additional extractor is used to extract provenance information, using one of the methods discussed in Section 7.2, from the input/output and the exposed internal states of the *primary system*.

The provenance maintenance module stores and maintains provenance information in a distributed store, which is either a set of relational tables or time-ordered system logs, based on the maintenance techniques used. The provenance querying module evaluates the provenance of a single tuple: upon receiving a provenance query, the querying module accesses to the store, retrieves the immediate predecessors and successors of the given tuple, and recursively issues queries to the querying modules located on other nodes to assemble the complete answer to the provenance query.

Implementation. NETTRAILS is implemented as an add-on to the ns-3 [74] network simulator. It makes extensive utilization of RapidNet [84], a declarative networking platform that compiles NDlog programs into applications that are executed by the ns-3 runtime. NETTRAILS uses the identical codebase for both simulation and deployment modes. NETTRAILS relies on PeerReview [41] for secure message communication and acknowledgments, and the management of the tamper-evident logs. Since auditing in NETTRAILS is driven by the forensic investigator, PeerReview's witnesses are not required, so we disabled this feature.

7.2 Extracting Provenance

To generate provenance graphs, a provenance system must extract information about events from the application to which it is applied. Provenance extraction (or the more general problem of correlating changes to network state based on incoming/outgoing messages) is an ongoing area of active research (e.g., [72, 75]) that is orthogonal to the main focus of this dissertation. Nevertheless, we have found the following three techniques useful for extracting provenance for target applications:

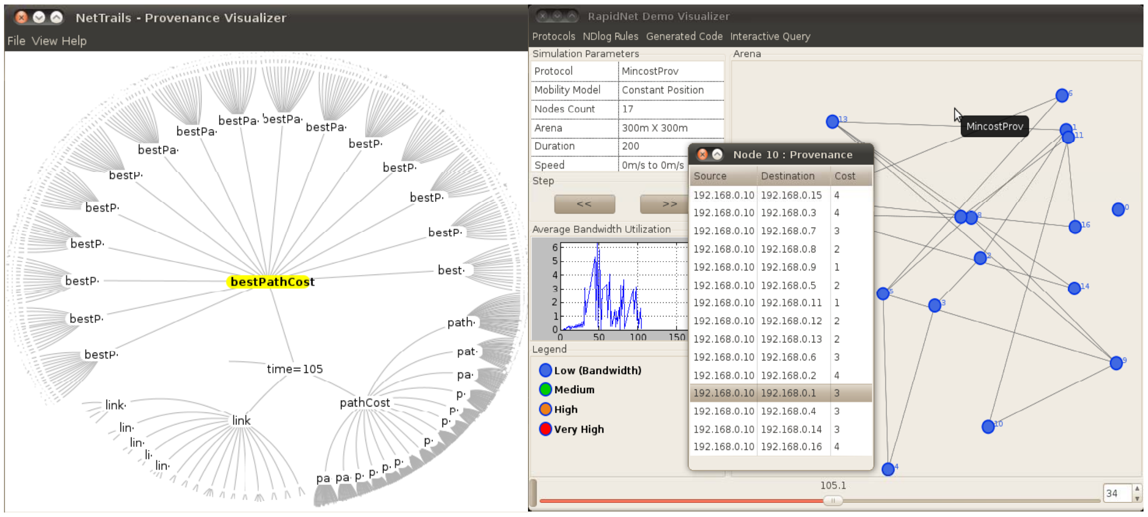
- **Method #1: Inferred provenance.** A provenance system can infer provenance data by transparently tracking data dependencies as inputs flow through the system. Inferred provenance can be applied when the dependencies are already explicitly captured in the programming language. We have applied this method to a version of the Chord DHT written in a declarative language (Section 7.4.1).

- **Method #2: Disclosed provenance.** Following the approach from [72], applications can explicitly call the interfaces provided by the provenance maintenance module to report data dependencies. This requires modifications to the source code; also, key parts of the application must be deterministic to enable the querier to verify that provenance was disclosed correctly. We have applied this method to the Hadoop MapReduce system (Section 7.4.2).

- **Method #3: Observed provenance.** When black-box applications cannot use either of the previous two approaches, one can observe how the application's outputs are derived from its inputs,and generate the provenance graph based on the observed inputs and outputs. We have applied this method to the Quagga BGP daemon (Section 7.4.3).

Extension – "maybe" rules. Since the third method cannot observe the internal state of the black box, it cannot necessarily predict all derivations completely. For example, a node n might choose a tuple τ from a set of other tuples, but the details of the decision process might not be known (e.g., because it is performed by a third-party binary). In this case, "maybe" rules can be used to infer provenance by observing the set of tuples: if all the τ_i exist, we cannot predict whether τ will appear, but if τ *does* appear, it must have been derived from the τ_i. To handle such situations, external specifications may contain *maybe rules*, written $\tau \xleftarrow{maybe} \tau_1@\mathtt{n}_1 \wedge \ldots \wedge \tau_k@\mathtt{n}_k$, which stipulates that the tuple τ on node n *may* be derived from tuples $\tau_1@n_1, \ldots, \tau_k@n_k$, but that the derivation is optional. In other words, as long as all of the underlying tuples are present, node n is free to decide whether or not to derive τ, and it is free to change its decision while the underlying tuples still exist. The rule merely describes τ's provenance if and when it exists.

There is another situation in which "maybe" rules are useful: a node may consider some rules or tuples as confidential. In this case, the node can be assigned *two* sets of rules: one full set for the actual computation (without "maybe" rules) and another to define provenance, in which the confidential computation is replaced by "maybe" rules. The second set can then be safely revealed to queriers. Another situation involves a node with a black-box computation, for which only the general dependencies are known.


NetTrails - Provenance Visualizer
File View Help
bestPathCost
time=105
pathCost
link
RapidNet Demo Visualizer
Protocols NDlog Rules Generated Code Interactive Query
Arena
Step
<< >>
Average Bandwidth Utilization
Legend
MincostProv
105.1
34

Simulation Parameters

Protocol	MincostProv
Mobility Model	Constant Position
Nodes Count	17
Arena	300m X 300m
Duration	200
Speed	0m/s to 0m/s

Legend

- 🔵 Low (Bandwidth)
- 🟢 Medium
- 🟠 High
- 🔴 Very High

Node 10 : Provenance

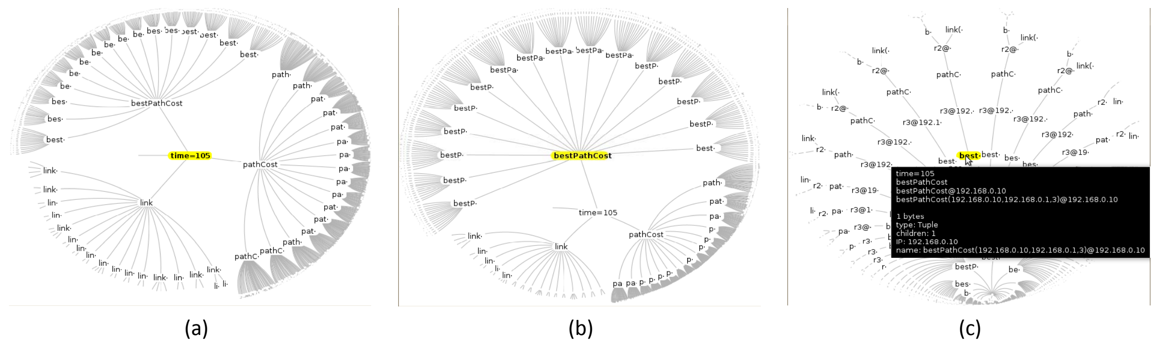
Source	Destination	Cost
192.168.0.10	192.168.0.15	4
192.168.0.10	192.168.0.3	4
192.168.0.10	192.168.0.7	3
192.168.0.10	192.168.0.8	2
192.168.0.10	192.168.0.9	1
192.168.0.10	192.168.0.5	2
192.168.0.10	192.168.0.11	1
192.168.0.10	192.168.0.12	2
192.168.0.10	192.168.0.13	2
192.168.0.10	192.168.0.6	3
192.168.0.10	192.168.0.2	4
192.168.0.10	192.168.0.1	3
192.168.0.10	192.168.0.4	3
192.168.0.10	192.168.0.14	3
192.168.0.10	192.168.0.16	4

Figure 7.2: A screenshot of the NETTRAILS system.

Extension – constraints. The second extension is intended for applications where the presence of *constraints* prevents us from modeling the state as completely independent tuples. For example, given tuples α and β, an application might derive *either* a tuple γ *or* a tuple δ, but not both. Modeling this with disjunctive rules would lose important information: if tuple δ replaces tuple γ, the appearance of δ and the disappearance of γ are not merely independent events, they are causally related. Thus, the explanation of δ's appearance should include the disappearance of γ. In G, we represent this by a direct edge between the corresponding INSERT and DELETE vertices.

7.3 Interactive Exploration Tool

While provenance maintenance is performed in a distributed fashion, some state may be centralized to increase the ease of visualizing provenance queries and results. In particular, per-node provenance information and other system state (e.g., the network topology and bandwidth utilization) may be periodically captured as



```
time=105
bestPathCost
bestPathCost@192.168.0.10
bestPathCost(192.168.0.10,192.168.0.1,3)@192.168.0.10

1 bytes
type: Tuple
children: 1
IP: 192.168.0.10
name: bestPathCost(192.168.0.10,192.168.0.1,3)@192.168.0.10
```

Figure 7.3: An example of interactive exploration in the provenance visualizer. Users start from the system-wide snapshot of the provenance at time T (screenshot a), select the table that they are interested in (screenshot b), and finally locate the provenance of a particular tuple instance (screenshot c). Focus changes are connected by smooth transitions, enabling progressive exploration.

system snapshots at each node and then propagated to a central *Log Store* which resides at the visualization node. These logs are subsequently used for interactive visualization, queries, and replays.

The generated logs are replayed using the *RapidNet visualizer* (to show the actual network topology and the position of nodes and links as the topology changes) and a *provenance visualizer*, which is based on hypertrees [44]. The provenance visualizer provides two useful features: the provenance graph is presented on a hyperbolic plane, enabling users to focus on small segments of the graph; additionally, users can navigate the provenance graph by changing focus with smooth transitions by clicking on or dragging the screen. To illustrate, Figure 7.2 shows an example execution of the NETTRAILS system where we show the provenance of the system state (captured as tuples) for a running MINCOST program. One may further issue customized queries against the provenance and visually show their progressive steps. We have created a video [21] that demonstrates the execution of the current NETTRAILS system. In the video, we maintain and query provenance for the MINCOST program running on a 16-node network.

Figure 7.3 shows a series of screenshots of the visualizer while it is used to interactively navigate the tree structure. This example is based on the MINCOST protocol, which computes pair-wise minimal path costs in a network. Figure 7.3(a) shows the root of the provenance tree at a particular point in time; Figure 7.3(b) shows all the pair-wise minimal path costs; and Figure 7.3(c) shows a close-up view of a particular tuple, as well as its attribute values and location (shown in the black rectangle).

Note that the topology and the provenance visualizer are interactively navigated in tandem: during the replay, users can interactively pause the network at a given time, and then view the provenance information of any node. Similarly, by navigating the hypertree provenance to explore dependencies among nodes, users can traverse and view the network state and the rules executing at another node. At any point in time, the users can customize the tree by issuing a provenance query that is then evaluated potentially across several nodes.

7.4 Applications

In this section, we demonstrate that NETTRAILS can be applied to three different types of applications with relatively little effort. Our examples cover all three provenance extraction methods described in Section 7.2. Moreover, these three applications differ in terms of the traffic they send, the amount of data they process, their scalability, etc., which allows us to evaluate NETTRAILS in a range of different scenarios.

7.4.1 Application #1: Chord

To test NETTRAILS's support for native NDlog programs, we applied it to a declarative implementation [63] of the Chord distributed hash table that uses Rapid-

Net [84]. The wide variety of known attacks against DHTs (e.g., eclipse attacks [96]) makes it a particularly attractive test case for secure network provenance. Since NETTRAILS can automatically transform any NDlog program into an equivalent one that automatically reports provenance, and since RapidNet is already deterministic, no modifications were required to the Chord source code.

7.4.2 Application #2: Hadoop MapReduce

To test NETTRAILS's support for disclosed provenance, we applied it to a version of Hadoop MapReduce [37] that we manually instrumented to report provenance to NETTRAILS at the level of individual key-value pairs. This application nicely complements prior work [73] on tracking data provenance in cloud stores.

Our prototype considers input files to be base tuples. The provenance of an intermediate key-value pair consists of the arguments of the corresponding `map` invocation, and the provenance of an output consists of the arguments of the corresponding `reduce` invocation. The set of intermediate key-value pairs sent from a map task to a reduce task constitutes a message that must be logged; thus, if there are m map tasks and r reduce tasks, our prototype sends up to $2mr$ messages (a request and a response for each pair). To avoid duplication of the large data files, we apply a trivial optimization: rather than copying the files in their entirety into the log, we log their hash values, which is sufficient to authenticate them later during replay. Since we are mainly interested in tracking the provenance of key-value pairs, we treat inputs from the JobTracker as base tuples. It would not be difficult to extend our prototype to the JobTracker as well.

Individual map and reduce tasks are already deterministic in Hadoop, so replay required no special modifications. We did, however, add code to replay map and reduce tasks separately, as well as a switch for enabling provenance reporting (recall that this is only needed during replay). Altogether, we added or modified less than 100 lines of code (LoC) in Hadoop itself, and we added another 550 LoC for the interface to NETTRAILS.

7.4.3 Application #3: Quagga

To test NETTRAILS's support for observed provenance, we applied it to the Quagga BGP daemon. BGP interdomain routing is plagued by a variety of attacks and malfunctions [77], so a secure provenance system seems useful for diagnostics and forensics. Rather than instrumenting Quagga for provenance and deterministic replay, we treated it as a "black box" and implemented a small proxy that a) transparently intercepts Quagga's BGP messages and converts them into NETTRAILS tuples, and b) converts incoming tuples back to BGP messages.

To enable NETTRAILS to capture the provenance of routes, we use a small NDlog specification of only four rules that captures the essential dependencies (but not the policy-level details) in BGP. The key rule is $\texttt{out}(AS, N_2, P, R_2) \xleftarrow{maybe} \texttt{in}(AS, N_1, P, R_1) \ \wedge \texttt{isExt}(R_2, R_1, AS)$, which stipulates that a route R_2 to an IP prefix P that is advertised to neighboring networks (through router N_2) is derived from another route R_1 to the same prefix, if 1) R_1 was previously announced to the local network AS (received at router N_1), and 2) the path vector in R_2 is the path vector in R_1 with at least one copy of the local AS number appended. The other rules specify how announcements propagate between networks, and that a network can export at most one route to each prefix at any given time. The latter constraint requires two separate rules.

In addition to the four rules, we wrote 626 LoC for the proxy; much of this code is generic and could be reused for other black-box applications. We did not modify any code in Quagga.

7.5 Limitations

By design, NETTRAILS is a forensic system; it cannot actively detect faults, but rather relies on a human operator to spot the initial symptom of an attack, which can then be investigated using NETTRAILS. Investigations are limited to the part of the system that is being monitored by NETTRAILS.

The security enhancement introduced by STAP assumes a full deployment, in which all the nodes in the distributed system are willing to deploy the NETTRAILS system and provide provenance information to the queries. While such setting is realistic in many application scenarios, such as in centrally managed systems or in systems that have some administrative authorities, we do not currently have a solution for partial deployments. It may be possible to use the approach adopted by NetReview [39] at the expense of slightly weaker guarantees.

NETTRAILS also does not have any built-in redundancy; if the adversary sacrifices one of his nodes and destroys all the provenance state on it, some parts of the provenance graph may no longer be reachable via queries. This could be mitigated by replicating each log on some other nodes, although, under our threat model, the problem cannot be avoided entirely because we have assumed that any set of nodes — and thus any replica set we may choose — could be compromised by the adversary.

Finally, NETTRAILS does not provide negative provenance, i.e., it can only explain the existence of a tuple (or its appearance or disappearance), but not its absence. Negative provenance is known to be a very difficult problem that is actively being researched in the database community [69]. We expect that NETTRAILS can be enhanced to support negative provenance by incorporating recent results from this community.

Chapter 8

Related Work

The secure time-aware provenance system presented in this dissertation expands upon previous results in the databases, networking and systems communities. In this section, we describe the related research in these areas.

8.1 Provenance

Since its importance was realized by the research community, provenance has been extensively studied, and successfully applied to a large range of application areas. Various provenance models have been proposed, and implemented in their corresponding systems. Our work presented in this dissertation was inspired by the rich previous work in this domain.

Provenance Model. A classic approach to model provenance, which is adopted in this dissertation, is to capture provenance as *graphs*. Provenance graphs reflect the relations between derived tuples and the base tuples that contribute to them. Each vertex represents a data object or an operation that transforms data objects (for instance, a database relational operator such as union, join, selection and projection), and each edge denotes a data flow among the vertcies. This approach is also adopted by many of the scientific computation systems [10, 15, 78, 100] and file systems [72], in which a directed-acyclic-graph (DAG) representation is used to describe dependencies.

Alternatively, data provenance may be more compactly represented using *algebraic representation* [9, 32]. Algebraic representations encode provenance using the binary operations $+$ and $*$ (representing, for example, union and join). For instance, let α, β, and γ represent base tuples, a tuple τ with provenance of $\alpha + \beta * \gamma$ means that τ is derivable if α exists or both β and γ are existent.

There have been several efforts to generalize provenance models and allow provenance interoperability. Green *et al.* [32] proposed provenance semiring, a provenance model that is useful for a variety of applications and generalizes previous models of provenance (such as lineage [16], why-provenance [9]) and query answering on annotated relations. The Open Provenance Model (OPM) [71] is a standardization effort that proposes an amalgamation of concepts from existing provenance systems, and aims to improve the provenance interoperability.

Maintenance and Querying. Provenance data are usually stored as additional tuple fields or separate tables in relational databases (such as Orchestra [33] and PermDB [29]), XML files (such as Kepler [100], ES3 [25]) and RDF files (such as Taverna [78]). In our NetTrails system, provenance information is maintained in an internal distributed relational database. While potential inconsistencies due to transient state could be resolved by maintaining provenance in bi-temporal databases [47, 54], NetTrails *inherently* captures and maintains temporal information along with the provenance data.

To facilitate access to the provenance data, provenance systems allow users to specify queries written in SQL [78, 100], XQuery [23], or query languages specifically designed for provenance (such as ProQL [48]). ProQL supports a wide variety of applications with derived data, and can be used to assess trust and derivability, detect side effects, as well as compute data annotations in particular provenance semirings. To improve query performance, recent work [3, 4] studies provenance labeling for efficiently evaluating reachability queries over large provenance graphs in a variety of workflow settings.

Visualization, an alternative approach to retrieve information from provenance, has been previously studied in VisTrails [10], in which workflow specifications can be compared side by side, and workflow specifications can be adjusted by example-based refinement [92].

Applications. Provenance has been implemented and integrated in many practical systems. Probabilistic databases, such as Trio [105], Mystiq [85, 86], and Panda [46], have applied provenance for efficient management of temporal and/or uncertainties. Trio, in particular, supports an uncertainty data model by associating each tuple with a confidential level, and updating the confidential levels of derived tuples based on their provenance. Collaborative data sharing systems (CDSS), such as Orchestra [33], uses provenance for trust management and reconciling conflicts among data from multiple sources. PASS [72] and Sprov [42] track file modification histories and causalities in file systems. PA-S3fs [73] and RAMP [45] focus on file systems and MapReduce workloads on the emerging cloud platform. Workflow systems, such as VisTrails [10], myGrid/Taverna [78], Kepler [100], Chimera [23], and ZOOM [15], use provenance support in scientific computations, to facilitate verification, reproducibility, and collaboration. VisTrails, for instance, captures the evolution of workflow specifications — the history of refining a workflow specification (e.g., the addition or deletion of a module, and the modification of a parameter). Several surveys [7, 17, 24] provides further details about workflow provenance systems.

Provenance Security. McDaniel *et al.* [68] outlines requirements for secure network provenance, emphasizing the need for provenance to be tamper-proof and non-repudiable. Hasan *et al.* proposes Sprov [42] that implements secure chain-structured provenance for individual documents; however, it lacks important features that are required in a distributed system, e.g., a consistency check to ensure that nodes are processing messages in a way that is consistent with their current state. Pedigree [83] captures provenance at the network layer in the form of per-packet *tags* that store a history of all nodes and processes that manipulated the packet. It assumes a trusted environment, and its set-based provenance is less expressive compared to NetTrails's graph-based dependency structure.

Provenance Privacy. More recently, researchers have studies, more specifically, the tradeoffs between privacy and utility for workflow provenance. For example, [18, 19] proposed the *module privacy* that ensures that the probability of guessing the correct outputs of a module, given the revealed inputs in the provenance, is below a given threshold. This is achieved by hiding a subset of the inputs (or outputs) of the modules in the provenance graph exposed to the users.

8.2 Forensics in Distributed Systems

Forensics in distributed systems has received a lot of traction in the system research community. There has been a substantial amount of work in this area. In this section, We summarize the work in the related topics.

Replay-based debugging. Replay-based debugging is enabled by recording all the non-deterministic events (such as network communications and interrupts from the operating systems) at runtime. Once a system fault is detected, users can then perform deterministic replay to reproduce the fault. Diagnosis is performed by inspecting how system states progress towards the fault, facilitated by watchpoints and breakpoints.

These systems, such as P2 debugger [98], liblog [28], Friday [27], WiDS [61], MaceMC [50], and QI [79] are designed to diagnose non-malicious faults, such as bugs or race conditions. When nodes have been compromised by an adversary, these systems can return incorrect results.

Log-based forensics. Log-based forensics systems capture execution logs at runtime by inserting additional statements in the source code, or by observing the inputs, outputs and system calls of each system component. For instance, Pip [88] logs path instances started from outside inputs; Backtracker [51, 52] records the objects and their causalities; logs of Magpie [5] are in the form of path instances consisting of the used system components; and D3S [60] modifies the underlying operating systems to allow automatic injection of state exposers and predicate checkers. Based on the logs and snapshots taken at runtime, users are enabled to reason about the causalities between system states, with the support from visualization tools and query engines.

The NETTRAILS system presented in this dissertation provides a general-purpose abstraction of dependencies, and enables richer functionalities. Among others, the main difference between NETTRAILS and these existing forensic systems is that NETTRAILS does not require trust in any components on the compromised nodes. For example, Backtracker [51, 52] and PASS [72] require a trusted kernel, cooperative ReVirt [6] a trusted VMM, and A2M [14] trusted hardware. ForNet [94] and NFA [106] assume a trusted infrastructure and collaboration across domains.

Accountability. Systems such as PeerReview [41] and NetReview [39] can automatically detect when a node deviates from the algorithm it is expected to run.

Tamper-evident logs are introduced to prevent modifications on history from unauthorized peers. In addition, equivocation, i.e. making conflicting statements to different nodes, are prevented by allowing peers to exchange logs to examine consistency. Attestation-based trusted hardware, such as A2M [14] and TrInc [56], can be used to further reduce the auditing overhead.

These systems cannot detect problems that arise from interactions between multiple nodes, such as BadGadget [34] in interdomain routing, or problems that are related to inputs or unspecified aspects of the algorithm. Also, accountability systems merely report that a node is faulty, whereas provenance systems also offer support for diagnosing faults and for assessing their effects on other nodes.

Proofs of misbehavior: Many systems that are designed to handle non-crash faults internally use proofs of misbehavior, such as the signed confessions in Ngan *et al.* [76], a set of conflicting tickets in SHARP [26], or the POM message in Zyzzyva [53]. In NETTRAILS, any evidence that creates a red vertex in G_ν essentially constitutes a proof of misbehavior, but NETTRAILS's evidence is more general because it proves misbehavior with respect to the (arbitrary) primary system, rather than with respect to NETTRAILS itself. Systems such as PeerReview [41] can generate protocol-independent evidence as well, but, unlike NETTRAILS's evidence, PeerReview's evidence is not diagnostic: it only shows that a node is faulty, but not what went wrong.

Fault tolerance: An alternative approach to the problem of Byzantine faults is to mask their effects, e.g., using techniques like PBFT [11]. Unlike NETTRAILS, these techniques require a high degree of redundancy and a hard bound on the number of faulty nodes, typically one third of the total. The two approaches are largely complementary and could be combined.

8.3 Declarative Networking

Declarative networking [62, 63, 64, 66] is a programming methodology that enables developers to concisely specify network protocols and services using a distributed recursive query language, and directly compile these specifications into a dataflow framework for execution. This approach provides ease and compactness of specification, and offers additional benefits such as optimizability and the potential for safety checks.

The development of declarative networking began in 2004 with an initial goal of enabling safe and extensible routers [65]. Declarative techniques have been then widely used in several domains including fault tolerance protocols [97], cloud computing [1], sensor networks [13], overlay network compositions [67], anonymity systems [95], mobile ad-hoc networks [58], wireless channel selection [57], network configuration management [12], and routing convergence analysis [87, 103].

In the NETTRAILS system, the incremental provenance maintenance and distributed query processing are enabled by a declarative networking engine. In principle, however, the techniques proposed in this dissertation can be generally realized using any sufficiently expressive distributed query processor. The advantage of using declarative networking is that robust implementations [84] exists that can be straightforwardly leveraged to develop provenance systems.

Chapter 9

Summary

This thesis research on Secure Time-aware Provenance (STAP) presents an approach that provides the fundamental functionality required for performing forensics queries — the capability to "explain" the existence (or change) of system state in a potentially adversarial environment. STAP reveals the dependencies between system states, and permits system operators to transitively tie observed faults to their potential causes, and to assess the damage that these faults may have caused to the rest of the system. We have identified several practical challenges in deploying STAP, and have presented the solutions that addressed each of the following main challenges:

Distribution. A key challenge of supporting provenance in distributed system is to develop an abstract system model in which provenance data can be maintained efficiently. we demonstrated that it is achievable by modeling the system state as a set of distributed databases, and by extracting logical dependencies from system specifications and runtime. Enabled by the distributed query processing capabilities, provenance information is then incrementally maintained as views of system state during the execution. We analytically and empirically showed that the overhead incurred by provenance maintenance is linear in the cost of the base system, and, therefore, does not affect its scalability.

Time-awareness. Another challenge is to track state changes over time in a relaxed system model, in which clocks are not synchronized and messages can be delayed, reordered or lost. To address this challenge, we examined the fundamental correlation between provenance and (observable) event ordering in distributed systems. We then presented an enhanced provenance model that provides a sound and complete representation that correctly captures the system dependencies.

Security. A final challenge is to provide security guarantees in completely untrusted environments, in which the adversary may have compromised an arbitrary subset of the nodes, and that he may have complete control over these nodes. We showed that, despite the conservative threat model, our security enhancement in STAP still provides strong, provable guarantees: it ensures that an observable symptom of a fault or an attack can always be traced to a specific event—passive evasion or active misbehavior—on at least one faulty node, even when the adversary attempts to prevent this.

To demonstrate STAP's practicality and generality, we have applied it to a variety of different systems, including the Internet's interdomain routing system, the Chord distributed hash table, and the Hadoop MapReduce system. The evaluation has demonstrated that STAP is able to detect a number of different problems that had been previously described in the literature, and that STAP is practical, both in terms of its run-time overhead and in terms of the effort required to deploy it.

9.1 Future Directions

We conclude with a list of several promising research directions suggested by the work in this dissertation.

9.1.1 Extensions to Thesis

STAP automates fault diagnosis and debugging by systematically maintaining and querying state dependencies as the system execution progresses. We intend to further improve its usability to encourage its adoption in academia and industrial development settings.

One important aspect for future research is to enhance the “readability” of the returned provenance results. Provenance information could be overwhelmingly large in systems with complex dependency logic. To address this challenge, we intend to explore the following two complementary approaches: the first approach focuses on developing an expressive yet easy-to-use interface (e.g., a SQL-like declarative query language tailored for the STAP model), for users to annotate and prune provenance data based on a customizable pattern; alternatively, the size and complexity of provenance information can be controlled by introducing layering into the provenance system, in which case provenance data can be captured at a variety of granularities, and be interactively expanded.

STAP mainly explores the authenticity and integrity aspects of security in provenance systems. We plan to extend the exploration to their counterparts, privacy and confidentiality. It is intriguing to study the tension between privacy and verifiability, two seemingly contradictory properties. As a first step, the private and verifiable routing (PVR) [36, 107] provides initial evidence that strong privacy guarantees can be achieved in interdomain routing, where the functionality of each node is well-restricted to route selection and advertisement based on a customized ranking function. We intend to further understand the performance implications or limits when extending the guarantees to more general systems.

9.1.2 Looking Beyond

Looking beyond the dissertation work, our long-time research direction is to facilitate the development of provably secure and reliable distributed systems. Specifically, we intend to accomplish this by enhancing the iterative development cycle of distributed systems, with research advances in the following aspects:

Systematic fault diagnosis and recovery. STAP can be used to systematically diagnose faults, where "explanations" of a suspicious symptom are compiled as a set of state dependencies that recursively trace back to the root causes. As the basis for inferring state dependencies, the high-level dependency logic (captured as derivation rules in STAP) is of critical importance. To generalize and further automate the extraction of such dependency logic from a target application, one potential avenue that we intend to explore is to employ programming language techniques that perform static (or dynamic) analysis on the information flow of target systems.

In addition to debugging, one intriguing direction is the use of STAP for provenance-based recovery. STAP maintains sufficient information to reproduce the system execution trace individually for each node. This brings opportunities to undo the damages caused by an exposed system fault, by applying the inverse operations in the reverse order. For example, a mistakenly deleted system state can be restored by the corresponding insertion. In addition, provenance keeps the dependency information and thus allows minimal recovery, i.e., the recovery only impacts the nodes that are actually affected by the fault.

Provenance-driven invariant generation. One important challenge in formal verification is for system designers to discover the safety properties (or invariants). The quality of these safety properties directly affects the quality of the verification results, however, there lacks a systematic approach to extract the safety properties, and the process largely relies on manual efforts today. To address this problem, we are interested in feeding the design bugs or security vulnerabilities exposed in fault diagnosis (as hints for the safety properties) to refine the invariants in the design and development phase.

Rigorous verification on design and vulnerabilities. Formal verification provides sound and complete guarantees by checking that a certain set of properties holds in all possible execution traces. Our current research focuses on static analysis, by leveraging prove-by-construction capability enabled by logic-based programming languages. We intend to extend the exploration to dynamic verification techniques, such as model checking, for verifying more complex properties and performing "what-if" analysis to discover potential vulnerabilities given certain assumptions on the attack model.

Appendix A

Correctness Proofs

We have proven that the TAP provenance model has the four properties (Validity, Soundness, Completeness and Minimality) presented in Section 3.3. The proofs for the corresponding four theorems are included below.

Lemma 1. *For any execution $\mathcal{E}$, and update $\Delta\tau$, the provenance graph $G(\Delta\tau, \mathcal{E})$ is acyclic.*

Proof. We first show that if there exists a cycle in $G(\Delta\tau, \mathcal{E})$, the cycle cannot include two vertices located on different nodes. Suppose there exists a cycle that contains two vertices v_1 and v_2 located on N_1 and N_2 respectively. Then, the cycle must contain a least one pair of SEND and RECV vertices in both the path from v_1 to v_2, and the path from v_2 to v_1. Each SEND and RECV corresponds to a message communication which takes a positive amount of time. Therefore, traversing from v_1 along the cycle back to v_1 results in an absolute increment in the timestamp. Contradiction.

If all the vertices in the cycle are located on the same node, then we can order the vertices according to their associated timestamps (now all the timestamps are with respect to the same local clock). Such order corresponds to the precedence of events in the execution. As time always progresses forward, such cycle cannot exist in $G(\Delta\tau, \mathcal{E})$. $\square$

Theorem 4. ***(Validity)*** *Given the initial state $\mathcal{S}_0$, for any event $d_i@N_i = (e_i, r_i, t_i, c_i, e'_i) \in \mathcal{A}(\Delta\tau, \mathcal{E})$, (a) there exists $d_j@N_j = (e_j, r_j, t_j, c_j, e'_j)$ that precedes $d_i@N_i$ in $\mathcal{A}(\Delta\tau, \mathcal{E})$, $e_i \in e'_j$, and (b) for all $\tau_k \in c_i, \tau_k \in \mathcal{S}_{i-1}$, where $\mathcal{S}_0 \xrightarrow{d_1@N_1} \mathcal{S}_1 \ldots \mathcal{S}_{i-2} \xrightarrow{d_{i-1}@N_{i-1}} \mathcal{S}_{i-1}$.*

Proof. Lemma 1 shows that any provenance graph $G(\Delta\tau, \mathcal{E})$ is acyclic, and thus $G(\Delta\tau, \mathcal{E})$ has a definitive depth. We prove the validity property using structural induction on the depth of the provenance graph $G(\Delta\tau, \mathcal{E})$.

Base case: The depth of $G(\Delta\tau, \mathcal{E})$ is one. In this case, $\Delta\tau$ is an insertion or deletion of a base tuple; $G(\Delta\tau, \mathcal{E})$ contains a single INSERT (or DELETE) vertex that corresponds to the update of the base tuple. Therefore, $\mathcal{A}(\Delta\tau, \mathcal{E})$ consists of a single event and is trivially valid.

Induction case: Suppose the validity of the extracted trace holds for any provenance graph with depth less than k ($k > 1$). Consider the provenance graph $G(\Delta\tau, \mathcal{E})$ with depth k. $\Delta\tau$ is an insertion or deletion of a derived tuple. Suppose the extracted event is $d_i@N_i = (e_i, r_i, t_i, c_i, e'_i)$, $\mathcal{A}(\Delta\tau, \mathcal{E})$ is valid if, a) the trigger event has been generated, namely, there exists $d_j@N_j = (e_j, r_j, t_j, c_j, e'_j)$ that precedes $d_i@N_i$ and $e_i \in e'_j$, and b) all the preconditions tuples exist, namely, for all $\tau_k \in c_i, \tau_k \in \mathcal{S}_{i-1}$, where $\mathcal{S}_0 \xrightarrow{d_1@N_1} \mathcal{S}_1 \ldots \mathcal{S}_{i-2} \xrightarrow{d_{i-1}@N_{i-1}} \mathcal{S}_{i-1}$.

We know that, by construction, the INSERT (or DELETE) vertex has an incoming edge from either a DERIVE (or UNDERIVE) vertex, or a RECV vertex.

- INSERT (or DELETE) has an incoming edge from a DERIVE (or UNDERIVE) vertex. Again, by construction, the DERIVE (or UNDERIVE) vertex has incoming edges from vertices representing the triggering event $\Delta\tau'$ (an INSERT or DELETE vertex) and all preconditions $\tau''_1, \ldots, \tau''_p$ (EXIST vertices).

  By the induction hypothesis, $\mathcal{A}$ outputs a valid trace $\{d'_1@N'_1, \ldots, d'_j@N'_j)$ for the subgraph for the trigger event $\Delta\tau'$ (i.e., $G(\Delta\tau', \mathcal{E})$), where $d'_j@N'_j$ corresponds to the generation of $\Delta\tau'$ (following the completeness property proved in Theorem 6). Because of the nature of algorithm $\mathcal{A}$ (which is based on topological sort), $d'_j@N'_j$ must be ordered before $d_i@N_i$, which satisfies condition a. Similarly, valid traces are generated for the updates that support the preconditions $\tau''_1, ..., \tau''_p$, which satisfies condition b. Therefore, $\mathcal{A}(\Delta\tau, \mathcal{E})$ is valid.

- INSERT (or DELETE) has an incoming edge from a RECV vertex. By construction, the RECV vertex is coupled with a SEND vertex which has an incoming edge from a DERIVE (or UNDERIVE) vertex. Following the same argument for the prior case, we can prove that $\mathcal{A}(\Delta\tau, \mathcal{E})$ is a valid trace.

□

Theorem 5. ***(Soundness)*** *$\mathcal{A}(\Delta\tau, \mathcal{E})$ is a subtrace of some $\mathcal{E}' \sim \mathcal{E}$.*

Proof. We need to show that a) all the events in $\mathcal{A}(\Delta\tau, \mathcal{E})$ also appear in $\mathcal{E}$ (and thus in any $\mathcal{E}' \sim \mathcal{E}$), and b) the local event ordering pertains on each node, that is, for any two events $d_1@N_i$ and $d_2@N_i$ in $\mathcal{A}(\Delta\tau, \mathcal{E})$ that are located on the same node N_i, $d_1@N_i$ precedes $d_2@N_i$ in $\mathcal{A}(\Delta\tau, \mathcal{E})$ iff $d_1@N_i$ precedes $d_2@N_i$ in $\mathcal{E}$.

Condition a. According to Algorithm 1, an event $d_i@N_i$ is generated and included in $\mathcal{A}(\Delta\tau, \mathcal{E})$ for each DERIVE (or UNDERIVE) vertex (and its direct parent and children) in the provenance graph $G(\Delta\tau, \mathcal{E})$. However, by construction, each DERIVE (or UNDERIVE) vertex v corresponds to an rule evaluation in $\mathcal{E}$. In the state transition model, the rule evaluation is modeled (see Definition 5) as an event $d_j@N_j = (\Delta\tau', r, t, \{\tau''_1, ..., \tau''_p\}, \Delta\tau)$, where $\Delta\tau$ is the trigger event, r and t are the rule used in and the time of the rule evaluation, each τ''_i represents a precondition, and $\Delta\tau$ is the generated update. We need to show that $d_i@N_i$ is identical to $d_j@N_j$.

This follows straightforwardly from the construction of $G(\Delta\tau, \mathcal{E})$: In addition to the DERIVE (or UNDERIVE) vertex v, its parent, either a SEND or INSERT (or DELETE) vertex v' for $\Delta\tau$, is also generated. Edges are added from v to its parent v', from the trigger event, a INSERT (or DELETE) vertex for $\Delta\tau'$, to v, and from the preconditions, EXIST vertices for $\tau_1'', ..., \tau_p''$, to v. Algorithm 1 reverses this process and generates event $d_i@N_i$ from these information, which gives $d_i@N_i = d_j@N_j$.

Condition b. According to Algorithm 1 (specifically, Line 7), $d_1@N_i$ precedes $d_2@N_i$ in $\mathcal{A}(\Delta\tau, \mathcal{E})$, iff $d_2@N_i$ has a larger timestamp than $d_1@N_i$. However, $d_2@N_i$ is assigned a larger timestamp iff $d_1@N_i$ precedes $d_2@N_i$ in the actually execution $\mathcal{E}$. Note that events on different nodes may be reordered in $\mathcal{A}(\Delta\tau, \mathcal{E})$, but this is captured by the equivalence ($\sim$) relation. $\square$

Theorem 6. ***(Completeness)*** *$\mathcal{A}(\Delta\tau, \mathcal{E})$ ends with the event that generates $\Delta\tau$.*

Proof. We need to show that a) $\mathcal{A}(\Delta\tau, \mathcal{E})$ contains an event $d_i@N_i$ that generates $\Delta\tau$, and b) $d_i@N_i$ is the last event in $\mathcal{A}(\Delta\tau, \mathcal{E})$.

Condition a. By construction, the INSERT (or DELETE) vertex for $\Delta\tau$ has an incoming edge from either a DERIVE (or UNDERIVE) vertex, or a RECV vertex.

- INSERT (or DELETE) has an incoming edge from a DERIVE (or UNDERIVE) vertex. By construction, the DERIVE (or UNDERIVE) vertex has incoming edges from vertices representing the triggering event $\Delta\tau'$ (an INSERT or DELETE vertex) and all preconditions $\tau_1'', ..., \tau_p''$ (EXIST vertices). Algorithm $\mathcal{A}$ (specifically, Line 18 - 31 in Algorithm 1) constructs an event $(\Delta\tau', r, t, \{\tau_1'', ..., \tau_p''\}, \Delta\tau)$, where r and t are the rule name and time encoded in the DERIVE (or UNDERIVE) vertex.
- INSERT (or DELETE) has an incoming edge from a RECV vertex. By construction, the RECV vertex is coupled with a SEND vertex which has an incoming edge from a DERIVE (or UNDERIVE) vertex. Following the same argument for the prior case, we can prove that $\mathcal{A}(\Delta\tau, \mathcal{E})$ contains an event that generates $\Delta\tau$.

Condition b. Now we have proved that some event $d_i@N_i$ that generates $\Delta\tau$ must exist in $\mathcal{A}(\Delta\tau, \mathcal{E})$, we next show that $d_i@N_i$ is the last event in $\mathcal{A}(\Delta\tau, \mathcal{E})$. The provenance graph $G(\Delta\tau, \mathcal{E})$ of $\Delta\tau$ is a subgraph of the provenance graph $G(\mathcal{E})$, and is rooted rooted by the INSERT (or DELETE) vertex that corresponds to $\Delta\tau$. Since all other vertices in $G(\Delta\tau, \mathcal{E})$ have a directed path to this INSERT (or DELETE) vertex, the corresponding events must all be ordered before $d_i@N_i$, so $d_i@N_i$ must necessarily be the last event in the subtrace. □

Theorem 7. ***(Minimality)*** *No valid trace $\mathcal{E}' \subset \mathcal{A}(\Delta\tau, \mathcal{E})$ is sound and complete.*

Proof. We prove the minimality property by induction on the syntactic structure of $\mathcal{A}(\Delta\tau, \mathcal{E})$: we show that an event $d_i@N_i \in \mathcal{A}(\Delta\tau, \mathcal{E})$ cannot be removed because it is necessary for some event $d_j@N_j$ appeared later in the trace. For presentation purposes, we suppose $\mathcal{A}(\Delta\tau, \mathcal{E}) = \{d_1@N_1, ..., d_m@N_m\}$.

Base case. According to the completeness property (Theorem 6), the last event $d_m@N_m$ in $\mathcal{A}(\Delta\tau, \mathcal{E})$ generates $\Delta\tau$. Therefore the base case trivially holds, as the remove of $d_m@N_m$ breaks the completeness property.

Induction case. Suppose the last k events $d_{m-k+1}@N_{m-k+1}, ..., d_m@N_m$ ($K \geq 1$) cannot be remove. We show that event $d_{m-k}@N_{m-k}$ cannot be removed as well: According to Algorithm 1, $d_{m-k}@N_{m-k}$ is constructed from a DERIVE (or UNDERIVE) vertex v and its direct parent and children. Consider the parent vertex v', v' must have an outgoing edge to some other vertex u in $G(\Delta\tau, \mathcal{E})$. Otherwise, v' would not be included in $G(\Delta\tau, \mathcal{E})$ which is a subgraph rooted by $\Delta\tau$.By construction, v' is either a SEND vertex, or a INSERT (or DELETE) vertex.

- v' is a INSERT (or DELETE) vertex, in which case, u is a DERIVE (or UNDERIVE) vertex. According to Algorithm 1, an event $d_j@N_j$ is constructed from u and its direct parent and children. Given the edges (v, v') and (v', u), we know that $d_j@N_j$ depends on $d_{m-k}@N_{m-k}$, and that $d_{m-k}@N_{m-k}$ precedies $d_j@N_j$. By applying the induction hypothesis ($d_j@N_j$ cannot be removed from $\mathcal{A}(\Delta\tau, \mathcal{E})$), we can conclude that $d_{m-k}@N_{m-k}$ also cannot be removed.

- v' is a SEND vertex, in which case, u is a RECV vertex that has a outgoing edge to a INSERT (or DELETE) vertex. By construction, the INSERT (or DELETE) vertex further has an outgoing edge to a DERIVE (or UNDERIVE) vertex. Following the similar argument for the prior case, we can prove that $d_{m-k}@N_{m-k}$ cannot be removed for validity.

$\square$

Bibliography

[1] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2010.

[2] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2011.

[3] Zhuowei Bao, Susan B. Davidson, Sanjeev Khanna, and Sudeepa Roy. An optimal labeling scheme for workflow provenance using skeleton labels. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[4] Zhuowei Bao, Susan B. Davidson, and Tova Milo. Labeling recursive workflow executions on-the-fly. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[6] Murtaza Basrai and Peter M. Chen. Cooperative ReVirt: adapting message logging for intrusion analysis. Technical Report University of Michigan CSE-TR-504-04, 2004.

[7] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Survey*, 37(1):1–28, 2005.

[8] Randall E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[9] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2001.

[10] Steven Callahan, Juliana Freire, Emanuele Santos, Carlos Scheidegger, Claudio Silva, and Huy Vo. VisTrails: Visualization meets data management. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006.

[11] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[12] Xu Chen, Yun Mao, Z. Morley Mao, and Jacobus van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Proceedings of ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010.

[13] David Chiyuan Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The Design and Implementation of a Declarative Sensor Network System. In *Proceedings of ACM Conference on Embedded networked Sensor Systems (SenSys)*, 2007.

[14] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[15] Sarah Cohen-Boulakia, Olivier Biton, Shirley Cohen, and Susan Davidson. Addressing the provenance challenge using zoom. *Concurrency and Computation : Practice and Experience*, 20:497–506, 2008.

[16] Yingwei Cui, Jennifer Widom, and Janet L.Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transaction on Database Systems (TODS)*, 25, 2000.

[17] Susan B. Davidson, Sarah Cohen Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M. McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 30(4):44–50, 2007.

[18] Susan B. Davidson, Sanjeev Khanna, Tova Milo, Debmalya Panigrahi, and Sudeepa Roy. Provenance views for module privacy. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2011.

[19] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, Julia Stoyanovich, Val Tannen, Yi Chen, and Tova Milo. Enabling privacy in provenance-aware workflow systems. In *Proceedings of Biennial Conference on Innovative Data System Research (CIDR)*, 2011.

[20] Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc., 1994.

[21] Demonstration Video for the NetTrails System. `http://netdb.cis.upenn.edu/rapidnet/sigmod11demo.html`.

[22] Anja Feldmann, Olaf Maennel, Z. Morley Mao, Arthur Berger, and Bruce Maggs. Locating internet routing instabilities. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMMM)*, 2004.

[23] Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of Scientific and Statistical Database Management Conference (SSDBM)*, 2002.

[24] Juliana Freire, David Koop, Emanuele Santos, and Claudio T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10, 2008.

[25] James Frew and Peter Slaughter. Provenance and annotation of data and processes. chapter ES3: A Demonstration of Transparent Provenance for Scientific Computation, pages 200–207. Springer-Verlag, Berlin, Heidelberg, 2008.

[26] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[27] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[28] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2006.

[29] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2009.

[30] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the system r database manager. *ACM Computing Survey*, 13(2):223–242, 1981.

[31] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2007.

[32] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2007.

[33] Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Olivier Biton, Zachary G. Ives, and Val Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007.

[34] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking*, 10(2):232–243, April 2002.

[35] GT-ITM. `http://www.cc.gatech.edu/projects/gtitm/`.

[36] Alexander J. T. Gurney, Andreas Haeberlen, Wenchao Zhou, Micah Sherr, and Boon Thau Loo. Having your cake and eating it too: Routing security with privacy protections. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets-X)*, 2011.

[37] Hadoop. `http://hadoop.apache.org/`.

[38] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[39] Andreas Haeberlen, Ioannis Avramopoulos, Jennifer Rexford, and Peter Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[40] Andreas Haeberlen and Petr Kuznetsov. The Fault Detection Problem. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, 2009.

[41] Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[42] Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing history forgery with secure provenance. *ACM Transactions on Storage (TOS)*, 5(4):1–43, 2009.

[43] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2):192–214, 1998.

[44] Hyperbolic Tree Java Library. `http://sourceforge.net/projects/hypertree/`.

[45] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. In *Proceedings of Biennial Conference on Innovative Data System Research (CIDR)*, 2011.

[46] Robert Ikeda and Jennifer Widom. Panda: A system for provenance and data. *IEEE Data Engineering Bulletin, Special Issue on Data Provenance*, 33:42–49, 2010.

[47] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and Richard Thomas Snodgrass. A glossary of temporal database concepts. *SIGMOD Record*, 21:35–43, 1992.

[48] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.

[49] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2007.

[50] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[51] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, 2005.

[52] Samuel T. King, Z. Morley Mao, Dominic Lucchetti, and Peter Chen. Enriching intrusion alerts through multi-host causality. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2005.

[53] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[54] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. Designing access methods for bitemporal databases. *IEEE Transaction on Knowledge and Data Engineering (TKDE)*, 10:1–20, 1998.

[55] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[56] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[57] Changbin Liu, Ricardo Correa, Harjot Gill, Tanveer Gill, Xiaozhou Li, Shivkumar Muthukumar, Taher Saeed, Boon Thau Loo, and Prithwish Basu. PUMA: Policy-based Unified Multi-radio Architecture for Agile Mesh Networking. In *Proceedings of International Conference on Communication Systems and Networks (COMSNETS)*, 2012.

[58] Changbin Liu, Richardo Correa, Xiaozhou Li, Prithwish Basu, Boon Thau Loo, and Yun Mao. Declarative policy-based adaptive mobile ad hoc networking. *IEEE/ACM Transactions on Networking (TON)*, 2011.

[59] Mengmeng Liu, Wenchao Zhou, Nicholas Taylor, Zachary Ives, and Boon Thau Loo. Recursive Computation of Regions and Connectivity in Networks. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2009.

[60] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: debugging deployed

distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[61] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[62] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006.

[63] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. *Communication of ACM*, 2009.

[64] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[65] Boon Thau Loo, Joseph M. Hellerstein, and Ion Stoica. Customizable Routing with Declarative Queries. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets-II)*, 2004.

[66] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMMM)*, 2005.

[67] Yun Mao, Boon Thau Loo, Zachary Ives, and Jonathan M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *Pro-*

ceedings of ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2008.

[68] Patrick McDaniel, Kevin Butler, Stephen McLaughlin, Radu Sion, Erez Zadok, and Marianne Winslett. Towards a Secure and Efficient System for End-to-End Provenance. In *Proceedings of the Workshop on the Theory and Practice of Provenance (TaPP)*, 2010.

[69] Alexandra Meliou, Wolfgang Gatterbauer, Katherine M. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2011.

[70] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[71] Luc Moreau, Beth Plale, Simon Miles, Carole Goble, Paolo Missier, Roger Barga, Yogesh Simmhan, Joe Futrelle, Robert E. McGrath, Jim Myers, Patrick Paulson, Shawn Bowers, Bertram Ludaescher, Natalia Kwasnikowska, Jan Van den Bussche, Tommy Ellkvist, Juliana Freire, and Paul Groth. The open provenance model (v1.01). `http://eprints.ecs.soton.ac.uk/16148/1/opm-v1.01.pdf`.

[72] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2006.

[73] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[74] Network Simulator 3. `http://www.nsnam.org/`.

[75] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2005.

[76] Tsuen-Wan Ngan, Dan Wallach, and Peter Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proceedings of International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[77] Old Nordstroem and Constantinos Dovrolis. Beware of BGP attacks. *ACM Computer Communications Review (CCR)*, Apr 2004.

[78] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Tim Carver, Matthew R. Pocock, and Anil Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:3045–3054, 2004.

[79] Adam J. Oliner and Alex Aiken. A query language for understanding component interactions in production systems. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2010.

[80] Hweehwa Pang and Kian-Lee Tan. Verifying Completeness of Relational Query Answers from Online Servers. *ACM Transactions on Information and System Security (TISSEC)*, 11(2):1–50, 2008.

[81] Hyunjung Park, Robert Ikeda, and Jennifer Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12), 2011.

[82] Quagga Routing Suite. `http://www.quagga.net/`.

[83] Anirudh Ramachandran, Kaushik Bhandankar, Mukarram Bin Tariq, and Nick Feamster. Packets with provenance. Technical Report GT-CS-08-02, Georgia Tech, 2008.

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

[85] Christopher Ré, Nilesh Dalvi, and Dan Suciu. Efcient top-k query evaluation on probabilistic data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2007.

[86] Christopher Ré and Dan Suciu. Approximate lineage for probabilistic databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2008.

[87] Yiqing Ren, Wenchao Zhou, Anduo Wang, Limin Jia, Alexander J.T. Gurney, Boon Thau Loo, and Jennifer Rexford. FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing. In *Proceedings of ACM SIGCOMM Conference on Data Communication (SIGCOMM) - demonstration*, 2011.

[88] Patrick Reynolds, Charles Edwin Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[89] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[90] RouteViews project. `http://www.routeviews.org/`.

[91] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical network support for ip traceback. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2000.

[92] Carlos Eduardo Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Cláudio T. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computing Graphics (TOVCG)*, 13(6):1560–1567, 2007.

[93] Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin. An implementation of a log-structured file system for unix. In *Proceedings of the USENIX Winter Conference (USENIX Winter)*, 1993.

[94] Kulesh Shanmugasundaram, Nasir Memon, Anubhav Savant, and Herve Bronnimann. ForNet: A distributed forensics network. In *Proceedings of International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS)*, 2003.

[95] Micah Sherr, Andrew Mao, William R. Marczak, Wenchao Zhou, Boon Thau Loo, and Matt Blaze. A3: An Extensible Platform for Application-Aware Anonymity. In *Proceedings of Network and Distributed System Security (NDSS)*, 2010.

[96] Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. Defending against the Eclipse attack in overlay networks. In *Proceedings of the ACM SIGOPS European workshop*, 2004.

[97] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT Protocols Under Fire. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[98] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using queries for distributed monitoring and forensics. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2006.

[99] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMMM)*, 2001.

[100] Workflow System, Ilkay Altintas, Oscar Barney, and Efrat Jaeger-frank. Provenance collection support in the kepler scientific workflow system. In *Proceedings of the International Provenance and Annotation Workshop (IPAW)*, 2006.

[101] Renata Teixeira and Jennifer Rexford. A measurement framework for pinpointing routing changes. In *Proceedings of the ACM SIGCOMM Network Troubleshooting Workshop*, 2004.

[102] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 1982.

[103] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. FSR:

Formal analysis and implementation toolkit for safe inter-domain routing. *IEEE/ACM Transactions on Networking (TON)*, 2012.

[104] The Stanford WebBase Project. `http://diglib.stanford.edu/~testbed/doc2/WebBase/`.

[105] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of Biennial Conference on Innovative Data System Research (CIDR)*, 2005.

[106] Yinglian Xie, Vyas Sekar, Mike Reiter, and Hui Zhang. Forensic analysis for epidemic attacks in federated networks. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2006.

[107] Mingchen Zhao, Wenchao Zhou, Alexander J. T. Gurney, Andreas Haeberlen, Micah Sherr, and Boon Thau Loo. Private and verifiable interdomain routing decisions. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMMM)*, 2012.

[108] Wenchao Zhou, Ling Ding, Andreas Haeberlen, Zachary Ives, and Boon Thau Loo. TAP: Time-aware provenance for distributed systems. In *Proceedings of the USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011.

[109] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[110] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. Technical Report MS-CIS-11-14, University of Pennsylvania, 2011.

[111] Wenchao Zhou, Qiong Fei, Shengzhi Sun, Tao Tao, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. NetTrails: A declarative platform for provenance maintenance and querying in distributed systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD) - demonstration*, 2011.

[112] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed time-aware provenance. 2012. In submission to VLDB.

[113] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.