2017

Unifying Static And Runtime Analysis In Declarative Distributed Systems

Chen Chen
University of Pennsylvania, cchen.upenn@gmail.com

Unifying Static And Runtime Analysis In Declarative Distributed Systems

Abstract
Today's distributed systems are becoming increasingly complex, due to the ever-growing number of network devices and their variety. The complexity makes it hard for system administrators to correctly configure distributed systems. This motivates the need for effective analytic tools that can help ensure correctness of distributed systems.

One challenge in ensuring correctness is that there does not exist one solution that works for all properties. One type of properties, such as security properties, are so critical that they demand pre-deployment verification (i.e., static analysis) which, though time-consuming, explores the whole execution space. However, due to the potential problem of state explosion, static verification of all properties is not practical, and not necessary. Violation of non-critical properties, such as correct routing with shortest paths, is tolerable during execution and can be diagnosed after errors occur (i.e., runtime analysis), a more light-weight approach compared to verification.

This dissertation presents STRANDS, a declarative framework that enables users to perform both pre-deployment verification and post-deployment diagnostics on top of declarative specification of distributed systems. STRANDS uses Network Datalog (NDlog), a distributed variant of Datalog query language, to specify network protocols and services. STRANDS has two components: a system verifier and a system debugger. The verifier allows the user to rigorously prove safety properties of network protocols and services, using either the program logic or symbolic execution we develop for NDlog programs. The debugger, on the other hand, facilitates diagnosis of system errors by allowing for querying of the structured history of network execution (i.e., network provenance) that is maintained in a storage-efficient manner.

We show the effectiveness of STRANDS by evaluating both the verifier and the debugger. Using the verifier, we prove path authenticity of secure routing protocols, and verify a number of safety properties in software-defined networking (SDN). Also, we demonstrate that our provenance maintenance algorithm achieves significant storage reduction, while incurring negligible network overhead.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Boon Loo

Second Advisor
Steve Zdancewic

Keywords
Declarative language, Distributed system, Logic, Provenance, Verification

Subject Categories
Computer Sciences

UNIFYING STATIC AND RUNTIME ANALYSIS IN DECLARATIVE DISTRIBUTED SYSTEMS

Chen Chen

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2017

Supervisor of Dissertation

Boon Thau Loo
Professor
Computer and Information Science

Co-Supervisor of Dissertation

Limin Jia
Assistant Research Professor
Electrical & Computer Engineering
Carnegie Mellon University

Graduate Group Chairperson

Lyle Ungar
Professor
Computer and Information Science

Dissertation Committee

Steve Zdancewic (Chair), Professor of Computer Science

Andre Scedrov, Professor of Mathematics and Computer Science

Andreas Haeberlen, Associate Professor of Computer Science

Wenchao Zhou, Assistant Professor of Computer Science (Georgetown University)

UNIFYING STATIC AND RUNTIME ANALYSIS
IN DECLARATIVE DISTRIBUTED SYSTEMS

To Ling Ding, my love.

ACKNOWLEDGMENT

This dissertation would not have been possible without the support and help of my advisors, committee members, collaborators, friends and family, who I would like to thank for their company and advice from the bottom of my heart.

First and foremost I would like to thank my advisor, Professor Boon Thau Loo, and my co-advisor, Professor Limin Jia (Carnegie Mellon University), who I closely collaborate with throughout my graduate life. Boon has been an extremely supportive advisor since my first day at the University of Pennsylvania. It is a privilege for me to work with Boon, whose passion for research is contagious. Boon is always actively involved my projects, whether it be providing insights into problems or evaluating the proposed solution. Boon also offers me comprehensive training in terms of programming, paper writing and oral presentation by working side-by-side with me throughout the development of research projects.

Though Limin advises me remotely, such collaboration never makes her help and advice any less. Limin patiently guided me through the first few years of my Ph.D life, when I gradually figured out, under her supervision, the initial research direction of applying formal methods to networking. This topic eventually turned into my dissertation. Limin also teaches me how to find, evaluate and tackle a problem in a systematic way, not to mention that she also gave tremendous help in improving my paper writing and experimental evaluation.

I am also very grateful for my dissertation committee members: Professor Steve Zdancewic, Professor Andre Scedrov, Professor Andreas Haeberlen, and Professor Wenchao Zhou. Steve is the committee chair for both my dissertation work and WPE-II examination, and provides constructive advice regarding my research orientation, write-up and presentation. Andre is always patient and cares much about the progress of my dissertation. Andreas has been extremely helpful in evaluating my dissertation research and providing insightful and important feedback to my dissertation. Wenchao is actually more than a committee member. In fact, Wenchao has been actively engaged in all my Ph.D projects and serves more like

an unofficial advisor. I learned a lot from Wenchao, through his advice, comments and his achievement as a professor.

I also want to express my sincere gratitude to all the collaborators. Sangeetha A.Jyothi was the first graduate student I collaborated at Upenn. She helped me understand the implementation of declarative programming languages quickly by answering almost all the questions I asked. Hao Xu is another graduate student I was lucky to collaborate with. Hao is an expert in C++ programming language, without whom I could not have been able to implement the demo for declarative SBGP so quickly. I also would like to thank Cheng Luo for implementing SCION with me and documenting all the experiments. Harshal Tushar Lehri joined the provenance project and was of tremendous help when evaluating our proposed solution. Our collaboration extended even after the paper was accepted. Lay Kuan Loh is an very important collaborator as well. We worked together for three years on two different projects. Her help with system implementation and proof of theorems was indispensable and I much appreciate her efforts and contribution to the work in this dissertation. Another person that I want to especially thank is Changbin Liu, who was my mentor during my internship at AT&T Labs Research. Changbin opened the door to Python for me, and worked side-by-side with me during that enjoyable three-month period at AT&T. In addition, I woud like to thank Suyog Mapara, Chen Zhu, Anupam Alur, Sibi Vijayakumar for contributing a lot to the work in this dissertation.

I am also lucky to make a lot of friends at Upenn. They are: Behnaz Arzani, Arthur Azevedo de Amorim, Saeed Abedi, Ang Chen, Loris D'Antoni, Yi Ge, Harjot Gill, Justin Hsu, Kai Hong, Radoslav Ivanov, Zhihao Jiang, Junyi Li, Xi Lin, Fei Miao, Gang Song, Xujie Si, Yifei Yuan, Zhepeng Yan, Dong Lin, Antonis Papadimitriou, Mukund Raghothaman, Nicu Stiurca, Nikos Vasilakis, Anduo Wang, Shaohui Wang, Yang Wu, Yinjun Wu, Zhiwei Wu, Meng Xu, Hongbo Zhang, Jianzhou Zhao, Mabel Zhang, Menglong Zhu, Mingchen Zhao, Nan Zheng, Qizhen Zhang, Yi Zhang, Zhuoyao Zhang, Xin Zhang.

Last but not the least, I want to thank my parents for their unrelenting support and

sacrifice during my pursuit of Ph.D. Without their understanding this journey could have been much harder. I am also extremely happy to have the company of Yang Li and Qianru Jia throughout my Ph.D life, who I always view as close family members.

Finally I dedicate this dissertation to my lovely girlfriend Ling Ding, who supports me and my work with her whole heart from the first day we met. My achievement is unimaginable without her continuing love and encouragement.

This dissertation is supported by the following funding: NSF CNS-1218066, NSF CNS-1117052, NSF CNS-1018061, NSF CNS-0845552, NSF ITR-1138996, NSF CNS-1115706, FA9550-12-1-0327, NSF CNS-1453392, NSF CNS-1513679, NSF CNS-1065130, NSF CNS-1513961, NSF CNS-1513734, AFOSR MURI 'Science of Cyber Security: Modeling, Composition, and Measurement' and AFOSR Young Investigator award.

ABSTRACT

UNIFYING STATIC AND RUNTIME ANALYSIS
IN DECLARATIVE DISTRIBUTED SYSTEMS

Chen Chen

Boon Thau Loo
Limin Jia

Today's distributed systems are becoming increasingly complex, due to the evergrowing number of network devices and their variety. The complexity makes it hard for system administrators to correctly configure distributed systems. This motivates the need for effective analytic tools that can help ensure correctness of distributed systems.

One challenge in ensuring correctness is that there does not exist one solution that works for all properties. One type of properties, such as security properties, are so critical that they demand pre-deployment verification (i.e., static analysis) which, though time-consuming, explores the whole execution space. However, due to the potential problem of state explosion, static verification of all properties is not practical, and not necessary. Violation of non-critical properties, such as correct routing with shortest paths, is tolerable during execution and can be diagnosed after errors occur (i.e., runtime analysis), a more light-weight approach compared to verification.

This dissertation presents STRANDS, a declarative framework that enables users to perform both pre-deployment verification and post-deployment diagnostics on top of declarative specification of distributed systems. STRANDS uses Network Datalog (NDlog), a distributed variant of Datalog query language, to specify network protocols and services. STRANDS has two components: a system verifier and a system debugger. The verifier allows the user to rigorously prove safety properties of network protocols and services, using either the program logic or symbolic execution we develop for NDlog programs. The

debugger, on the other hand, facilitates diagnosis of system errors by allowing for querying of the structured history of network execution (i.e., network provenance) that is maintained in a storage-efficient manner.

We show the effectiveness of STRANDS by evaluating both the verifier and the debugger. Using the verifier, we prove path authenticity of secure routing protocols, and verify a number of safety properties in software-defined networking (SDN). Also, we demonstrate that our provenance maintenance algorithm achieves significant storage reduction, while incurring negligible network overhead.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

Distributed systems today are playing an ever more important role in supporting a variety of services, such as distributed file systems [38]. As a result, failure in a distributed system is costly, especially for those that provide mission-critical services. Despite the importance of system correctness, it is difficult to manually ensure that a distributed system operates as expected, because a typical distributed system today has large scale (i.e., millions of servers), contains heterogeneous devices (e.g., routers, switches and middleboxes) and changes configuration frequently (e.g., due to virtual machine migration).

In light of this, researchers have proposed a variety of approaches to help network administrators and researchers analyze distributed systems. One branch of work applies static analysis to distributed systems to verify properties that are expected to hold during system execution ([56][9][47][50][13]). Static analysis requires the analyst to formally specify the system in question, and use techniques such as model checking [8], theorem proving [14], and symbolic execution [51] to rigorously analyze the desired properties of the specification. Static analysis could unveil design flaws before they manifest themselves at runtime, which could lead to undesirable and even disastrous consequences.

On the other hand, static analysis is not the panacea for system failure, for several reasons. First, static analysis has no guarantee over unproven properties. Certain properties (e.g., security guarantee) may not even occur to the analyst when he/she is designing the system. Second, a proven property could still be violated during runtime, due to bugs in low-level software (e.g., compilers) and hardware (e.g., processors). Last but not the least, static analysis is more suitable for an environment where system policies and configuration are relatively static. A dynamic environment – e.g., a distributed system with frequent virtual machine migration – changes so rapidly that the result of static analysis could become

outdated soon after verification is completed.

To overcome the deficiency of static analysis, network administrators also need tools to help analyze distributed systems at runtime – i.e., monitoring the system, and, when the system fails, identifying the root cause of the problem. For example, message logging [29] is a typical runtime analysis technique that is widely used by network administrators. Also, packet monitoring tools such as wireshark [1] are also popular in detecting abnormal traffic in networks. There is work such as NetSight [43] that records the packet history as well. Recently, the emergence of network provenance [95][92] has made it possible for network administrators to query the complete derivation history of network events. Network provenance captures the causal relationship among individual events in a network, even if these events may occur on different network devices. Optimization also has been made to maintain network provenance at low cost, such as only carrying the last-hop provenance information during system execution (i.e., reference-based provenance maintenance [95]).

The problem with runtime analysis is its low coverage over the space of all possible system execution traces. Subtle bugs, such as those caused by race conditions, may only happen occasionally, making it hard for the user to identify the root cause simply by monitoring execution traces. Furthermore, the overhead of enabling runtime analysis is high. To perform system monitoring tasks, for example, a distributed system often needs to be instrumented with monitoring software or hardware that inevitably interferes with normal execution. The storage overhead required for maintaining system meta-data is also a concern. For instance, in a distributed system where millions of packets are input every second, storing history (or provenance) of all the packets takes up much storage space.

This dissertation intends to bridge the gap between static analysis and runtime analysis through a unified declarative specification language – Network Datalog (NDlog) [54]. We show that not only popular static analysis techniques such as theorem proving and symbolic execution can be successfully applied to a distributed system modeled by NDlog, but runtime analysis, such as storage-optimized network provenance, can be supported easily as

well. Furthermore, with the help of NDlog, we try to combine static analysis and runtime analysis, so that they could complement each other in system verification and debugging.

An overview of the proposed framework – called STRANDS– can be found in Figure 1. Oval nodes in Figure 1 represent input by the user, including the declarative specification (i.e., NDLog programs) of distributed system and the properties to be verified. We choose to use NDLog as our specification language for two reasons. First, it is shown that declarative languages such as NDLog can specify a variety of network protocols concisely [55]. Second, NDLog is a specification language that allows for both verification and low-level implementation. As a result, static and runtime analysis can be performed in a unified framework.

The three shadow boxes represent the three components of this dissertation: (1) a program logic for verifying security properties of secure routing protocols [18] (Chapter 3); (2) a static analyzer of network protocols and services using symbolic execution of NDlog programs [23] (Chapter 4); and (3) a storage-efficient provenance maintenance engine [22] (Chapter 5). Specifically:

- A Hoare-style program logic is developed for verifying security properties of secure routing protocols, such as Secure BGP (SBGP) [75]. To capture the security primitives in those protocols, NDLog is enhanced with security-related user-defined functions (e.g., symmetric/asymmetric encryption), to create a new programming language SANDlog. The program logic is proved to be sound with regards to the semantics of SANDlog. In the case study, we demonstrate the effectiveness of the proposed logic by formally proving the path authenticity property over two secure routing proposals: SBGP and SCION [91].

- Though the program logic is effective in property verification, the manual proof process with proof assistants (e.g., Coq [79]) could be tedious and time-consuming for network administrators. Therefore, the second part of this dissertation aims at achieving

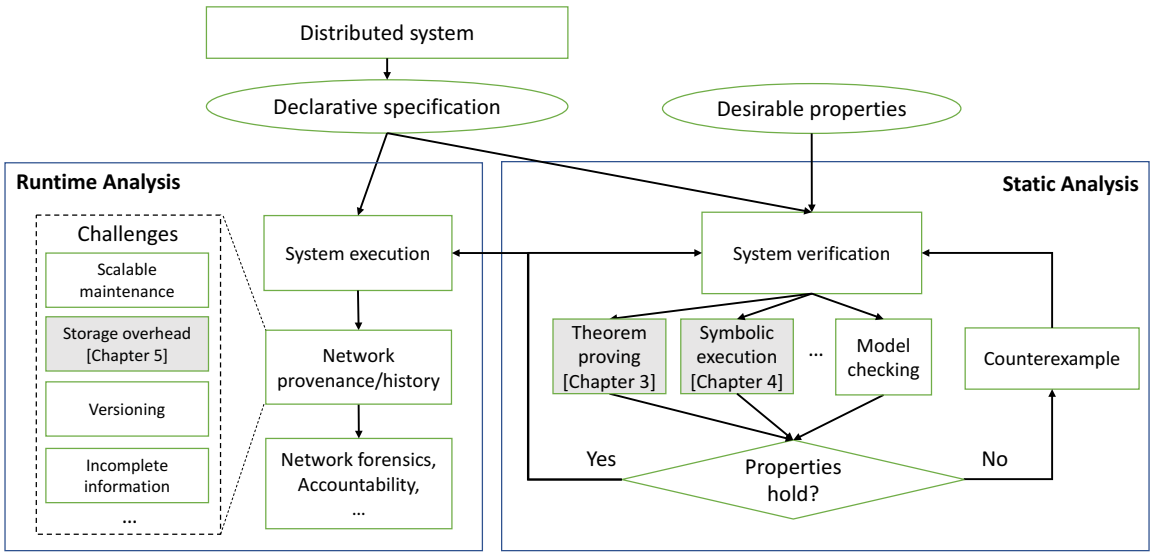


Figure 1: The overall architecture of STRANDS that unifies static and runtime analysis of distributed systems.

automation during static analysis of distributed systems. To do this, expressiveness of the property specification language is limited – i.e., the properties to be proved need to be specified in a restricted form of first-order logic. During verification, the specified property is checked, with a few novel optimization, against all possible execution traces of the distributed system. The execution traces are obtained through symbolic execution of the NDLog program specifying the system. With the help of an SAT solver, the verification process would further produce a counterexample when the property fails to hold. In the case study, a number of properties about software-defined networking applications are verified, with several subtle bugs identified.

- The third piece of this dissertation focuses on providing storage-efficient network provenance for runtime analysis of distributed systems. Network provenance [95] enables network administrators to query the execution history of network events. To compress provenance, an equivalence relation for provenance trees is defined and each equivalence class only maintains one concrete provenance copy that is shared by all the members. The compression process is efficient, as it is proved that the equiva-

lence of provenance trees could be identified by only examining the attributes of input tuples. Such efficiency is attributed to the introduction of Distributed Event-driven Linear Programs (DELP) – a constrained variant of NDLog programs – for modeling network protocols and services. Our experiments of packet forwarding and DNS resolution showed that the compression scheme achieves low storage consumption, low network overhead and low query latency.

Based on Figure 1, to perform analysis of a distributed system, a user first needs to specify the system in NDLog or its variants (e.g., SANDlog or DELP). The specification is readily subject to formal verification, either with the help of program logic or by symbolic execution. If the property is verified as true, the user could further prove other desirable properties, or execute the specification using a declarative networking engine (e.g., RapidNet [70]). If verification fails, the framework would be able to generate counterexamples to help the user identify and fix errors in the specification of either the program or the property. On the other hand, The user could perform runtime analysis after execution – e.g., network forensics or accountability – by querying the (compressed) provenance maintained by the provenance engine.

The structure of the dissertation is as follows, we present the program logic in Chapter 3, and show how to automatically verify safety properties of networking applications in Chapter 4. We present compressed provenance for runtime analysis in Chapter 5. The related work and the future work are summarized in Chapter 6 and Chapter 7 respectively.

CHAPTER 2

Background

We first provide the background of our unified specification language – Network Datalog (NDlog) [54], and its operational semantics. In each individual piece of work, we extend NDlog with a variety of features (e.g., security primitives) to cater to specific needs of different applications.

2.1. Network Datalog

2.1.1. Syntax

NDlog's syntax is summarized in Figure 2. A typical NDlog program is composed of a set of rules, each of which consists of a rule head and a rule body. The rule head is a predicate, or relation (we use predicate and relation interchangeably). A rule body consists of a list of body elements which are either relations or atoms (i.e. assignments and inequality constraints). The head relation supports aggregation functions as its arguments, whose semantics will be introduced in Section 2.1.2.

Atom	a	::=	$x := t \mid t_1 \ bop \ t_2$
Terms	t	::=	$x \mid c \mid \iota \mid f(\vec{t}) \mid f_c(\vec{t})$
Predicate	$pred$	::=	$p(agH) \mid p(agB)$
Body Elem	B	::=	$p(agB) \mid a$
Arg List	ags	::=	$\cdot \mid ags, x \mid ags, c$
Rule Body	$body$	::=	$\cdot \mid body, B$
Body Args	agB	::=	$@\iota, ags$
Rule	r	::=	$p(agH) :- body$
Head Args	agH	::=	$agB \mid @\iota, ags, F_{agr}\langle x \rangle, ags$
Base tp rules	b	::=	$p(agH).$
Program	$prog(\iota)$	::=	$b_1, \cdots, b_n, r_1, \cdots, r_k$

Figure 2: Syntax of NDlog

sp1 $\mathsf{path}(@s, d, c, p) :- \mathsf{link}(@s, d, c), p := [s, d].$
sp2 $\mathsf{path}(@z, d, c, p) :- \mathsf{link}(@s, z, c1), \mathsf{path}(@s, d, c2, p1), c := c1 + c2, p := z::p1.$
sp3 $\mathsf{bestPath}(@s, d, \mathsf{min}\langle c\rangle, p) :- \mathsf{path}(@s, d, c, p).$

Figure 3: A NDlog program for computing all-pair shortest paths

To support distributed execution, a NDlog program *prog* is parametrized over the node it runs on. Each relation in the program is supposed to have a location specifier, written $@\iota$, which specifies where a relation resides and serves as the first argument of a relation. A rule head can specify a location different from its body relations. When such a rule is executed, the derived tuple is sent to the remote node represented by the location specifier of the head tuple. We discuss the operational semantics of NDlog in detail in Section 2.1.2.

In Figure 3, we show an example program for computing the shortest path between each pair of nodes in a network. s is the location parameter of the program, representing the ID of the node where the program is executing. Each node stores three kinds of tuples: $\mathsf{link}(@s, d, c)$ means that there is a direct link from s to d with cost c; $\mathsf{path}(@s, d, c, p)$ means that p is a path from s to d with cost c; and $\mathsf{bestPath}(@s, d, c, p)$ states that p is the lowest-cost path between s and d. Here, link is a base tuple, whose values are determined by the concrete network topology. path and bestPath are derived tuples. Figure 3 only shows the rules common to all network nodes. Rules for initializing the base tuple link depend on the topology and are omitted from the figure.

In the program, rule *sp1* computes all one-hop paths based on direct links. Rule *sp2* expresses that if there is a link from s to z of cost *c1* and a path from s to d of cost *c2*, then there is a path from z to d with cost *c1+c2* (for simplicity, we assume links are symmetric, i.e. if there is a link from s to d with cost c, then a link from d to s with the same cost c also exists). The generated path tuple will be sent to z, as indicated by the head of *sp2*. Finally, rule *sp3* aggregates all paths with the same pair of source and destination (s and d) to compute the shortest path. The arguments that appear before the aggregation denotes the group-by keys.

To execute the program, a user provides rules for initializing base tuples. For example, if we would like to run the shortest-path program over the topology given in Figure 4, the following rules will be included in the program. Rules $rb1$ lives at node A, rules $rb2$ and $rb3$ live at node B, and rule $rb4$ lives at node C.

$rb1$ $\mathsf{link}(@A, B, 1).$ $rb3$ $\mathsf{link}(@B, C, 1).$

$rb2$ $\mathsf{link}(@B, A, 1).$ $rb4$ $\mathsf{link}(@C, B, 1).$

2.1.2. Operational Semantics

The operational semantics of NDlog adopts a distributed state transition model. Each node runs a designated NDlog program, and maintains a database of derived tuples as its local state. Nodes can communicate with each other by sending tuples over the network, which is represented as a global network queue. The evaluation of the NDlog programs follows the PSN algorithm [54], and updates the database incrementally. The semantics introduced here is similar, except that we make explicit which tuples are derived, which are received, and which are sent over the network. This addition is crucial to specifying and proving protocol properties.

At a high-level, each node computes its local fixed-point by firing the rules on newly-derived tuples. The fixed-point computation can also be triggered when a node receives tuples from the network. When a tuple is derived, it is sent to the node specified by its location specifier. Instead of blindly computing the fixed-point, we make sure that only rules whose body tuples are updated are fired. The operational semantics also support deletion of tuples. A deletion is propagated through the rules similar to an insertion.

More formally, the constructs needed for defining the operational semantics of NDlog are presented below.

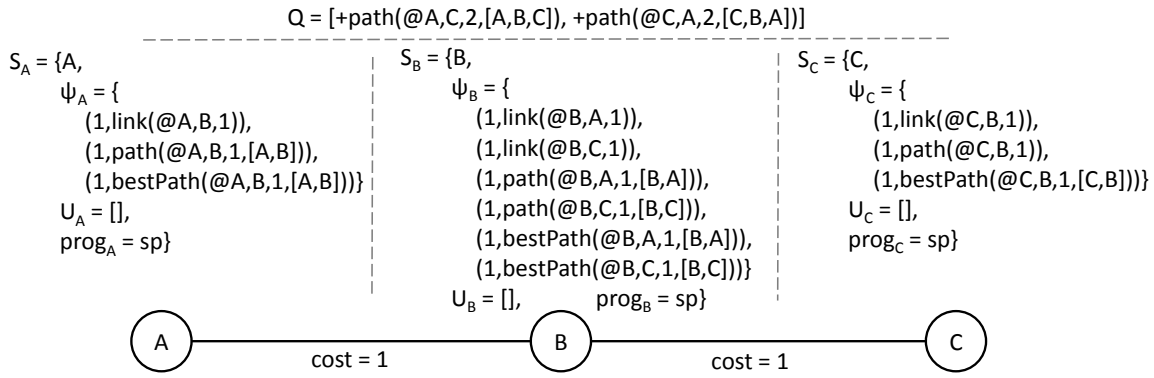


Figure 4: An Example Scenario.

$$
\begin{array}{llllllll}
\textit{Table} & \Psi & ::= & \cdot \mid \Psi, (n, P) & \textit{Network Queue} & \mathcal{Q} & ::= & \mathcal{U} \\
\textit{Update} & u & ::= & -P \mid +P & \textit{Local State} & \mathcal{S} & ::= & (\iota, \Psi, \mathcal{U}, \textit{prog}(\iota)) \\
\textit{Update List} & \mathcal{U} & ::= & [u_1, \cdots, u_n] & \textit{Configuration} & \mathcal{C} & ::= & \mathcal{Q} \rhd \mathcal{S}_1, \cdots, \mathcal{S}_n \\
\textit{Trace} & \mathcal{T} & ::= & \xrightarrow{\tau_0} \mathcal{C}_1 \xrightarrow{\tau_1} \mathcal{C}_2 \cdots \xrightarrow{\tau_n} \mathcal{C}_{n+1} & & & &
\end{array}
$$

We write P to denote tuples. The database for storing all derived tuples on a node is denoted Ψ. Because there could be multiple derivations of the same tuple, we associate each tuple with a reference count n, recording the number of valid derivations for that tuple. An update is either an insertion of a tuple, denoted $+P$, or a deletion of a tuple, denoted $-P$. We write $\mathcal{U}$ to denote a list of updates. A node's local state, denoted $\mathcal{S}$, consists of the node's identifier ι, the database Ψ, a list of unprocessed updates $\mathcal{U}$, and the program *prog* that ι runs. A configuration of the network, written $\mathcal{C}$, is composed of a network update queue $\mathcal{Q}$, and the set of the local states of all the nodes in the network. The queue $\mathcal{Q}$ models the update messages sent across the network. Finally, a trace $\mathcal{T}$ is a sequence of time-stamped (i.e. τ_i) configuration transitions.

Figure 4 presents an example scenario of executing the shortest-path program in Figure 3. The network consists of three nodes, A, B and C, connected by two links with cost 1. Each node's local state is displayed right above the node. For example, the local state of the node A is given by S_A above it. The network queue Q is presented at the top of Figure 4. In the current state, all three nodes are aware of their direct neighbors, i.e., link tuples are

in their databases Ψ_A, Ψ_B and Ψ_C. They have constructed paths to their neighbors (i.e., the corresponding path and bestPath tuples are stored). The current network queue Q stores two tuples: +path(@A,C,2,[A,B,C]) and +path(@C,A,2,[C,B,A]), waiting to be delivered to their destinations (node A and C respectively). These two tuples are the result of running *sp2* at node B. We will explain further how configurations are updated based on the updates in the network queue when introducing the transition rules.

Top-level transitions. The small-step operational semantics of a node is denoted $\mathcal{S} \hookrightarrow \mathcal{S}', \mathcal{U}$. From state $\mathcal{S}$, a node takes a step to a new state $\mathcal{S}'$ and generates a set of updates $\mathcal{U}$ for other nodes in the network. The small-step operational semantics of the entire system is denoted $\mathcal{C} \rightarrow \mathcal{C}'$, where $\mathcal{C}$ and $\mathcal{C}'$ respectively represent the states of all nodes along with the network queue before and after the transition. Figure 5 defines the rules for system state transition.

- **Global state transition ($\mathcal{C} \rightarrow \mathcal{C}'$).**

  Rule NodeStep states that the system takes a step when one node takes a step. As a result, the updates generated by node i are appended to the end of the network queue. We use $\circ$ to denote the list append operation. Rule DeQueue applies when a node receives updates from the network. We write $\mathcal{Q}_1 \oplus \mathcal{Q}_2$ to denote a merge of two lists. Any node can dequeue updates sent to it and append those updates to the update list in its local state. Here, we overload the $\circ$ operator, and write $\mathcal{S} \circ \mathcal{Q}$ to denote a new state, which is the same as $\mathcal{S}$, except that the update list is the result of appending $\mathcal{Q}$ to the update list in $\mathcal{S}$.

- **Local state transition ($\mathcal{S} \hookrightarrow \mathcal{S}', \mathcal{U}$).** Rule Init applies when the program starts to run. Here, only base rules—rules that do not have a rule body—can fire. The auxiliary function *BaseOf*(*prog*) returns all the base rules in *prog*. In the resulting state, the internal update list ($\mathcal{U}_{in}$) contains all the insertion updates located at ι, and the external update list ($\mathcal{U}_{ext}$) contains only updates meant to be stored at a

$\boxed{\mathcal{S} \hookrightarrow \mathcal{S}', \mathcal{U}}$

$$\frac{\mathcal{U}_{in} = [+p_1(@\iota, \vec{t}_1), ..., +p_m(@\iota, \vec{t}_m)] \qquad [p_1(@\iota, \vec{t}_1), \cdots, p_m(@\iota, \vec{t}_m)] = \mathit{BaseOf}(\mathit{prog})}{(\iota, \varnothing, [], \mathit{prog}) \hookrightarrow (\iota, \varnothing, \mathcal{U}_{in}, \mathit{prog}), []} \textsc{Init}$$

$$\frac{(\mathcal{U}_{in}, \mathcal{U}_{ext}) = \mathit{fireRules}(\iota, \Psi, u, \Delta \mathit{prog})}{(\iota, \Psi, u :: \mathcal{U}, \mathit{prog}) \hookrightarrow (\iota, \Psi \uplus u, \mathcal{U} \circ \mathcal{U}_{in}, \mathit{prog}), \mathcal{U}_{ext}} \textsc{RuleFire}$$

$\boxed{\mathcal{C} \rightarrow \mathcal{C}'}$

$$\frac{\mathcal{S}_i \hookrightarrow \mathcal{S}'_i, \mathcal{U} \qquad \forall j \in [1, n] \wedge j \neq i, \mathcal{S}'_j = \mathcal{S}_j}{\mathcal{Q} \rhd \mathcal{S}_1, \cdots \mathcal{S}_n \rightarrow \mathcal{Q} \circ \mathcal{U} \rhd \mathcal{S}'_1, \cdots \mathcal{S}'_n} \textsc{NodeStep}$$

$$\frac{\mathcal{Q} = \mathcal{Q}' \oplus \mathcal{Q}_1 \cdots \oplus \mathcal{Q}_n \qquad \forall j \in [1, n] \qquad \mathcal{S}'_j = \mathcal{S}_j \circ \mathcal{Q}_j}{\mathcal{Q} \rhd \mathcal{S}_1, \cdots \mathcal{S}_n \rightarrow \mathcal{Q}' \rhd \mathcal{S}'_1, \cdots \mathcal{S}'_n} \textsc{DeQueue}$$

$\boxed{\mathit{fireRules}(\iota, \Psi, u, \Delta \mathit{prog}) = (\mathcal{U}_{in}, \mathcal{U}_{ext})}$

$$\frac{}{\mathit{fireRules}(\iota, \Psi, u, []) = ([], [])} \textsc{Empty}$$

$$\frac{\mathit{fireSingleR}(\iota, \Psi, u, \Delta r) = (\Psi', \mathcal{U}_{in1}, \mathcal{U}_{ext1}) \qquad \mathit{fireRules}(\iota, \Psi', u, \Delta \mathit{prog}) = (\mathcal{U}_{in2}, \mathcal{U}_{ext2})}{\mathit{fireRules}(\iota, \Psi, u, (\Delta r, \Delta \mathit{prog})) = (\mathcal{U}_{in1} \circ \mathcal{U}_{in2}, \mathcal{U}_{ext1} \circ \mathcal{U}_{ext2})} \textsc{Seq}$$

Figure 5: Operational Semantics

node different from ι. In this case, it is empty. Rule RuleFire (Figure 5) computes new updates based on the program and the first update in the update list. It uses a relation *fireRules*, which processes an update u, and returns a pair of update lists, one for node ι itself, the other for other nodes. The last argument for *fireRules*, $\Delta \mathit{prog}$, transforms every rule r in the program *prog* into a *delta* rule, Δr, for r, which we explain when we discuss incremental maintenance. After u is processed, the database of ι is updated with the update u ($\Psi \uplus u$). The $\uplus$ operation increases (decreases) the reference count of P in Ψ by one, when u is an insertion (deletion) update $+P$ ($-P$). The update list in the resulting state is augmented with the new updates generated from processing u.

- **Fire rules** ($\mathit{fireRules}(\iota, \Psi, u, \Delta \mathit{prog}) = (\mathcal{U}_{in}, \mathcal{U}_{ext})$). Given one update, we fire rules in

the program *prog* that are affected by this update. Rule Empty is the base case where all rules have been fired, so we directly return two empty sets. Given a program with at least one rule $(\Delta r, \Delta prog)$, rule Seq first fires the rule Δr, then recursively calls itself to process the rest of the rules in $\Delta prog$. The resulting updates are the union of the updates from firing Δr and $\Delta prog$.

Given the example scenario in Figure 4, at this moment node A dequeues the update +path(@A,C,2,[A,B,C]) from the network queue Q at the top of Figure 4, and puts it into the unprocessed update list $\mathcal{U}_A$ (rule DeQueue). Node A then locally processes the update by firing all rules that are triggered by the update, and generates new updates $\mathcal{U}_{in}$ and $\mathcal{U}_{ext}$. In the resulting state, the local state of node A (Ψ_A) is updated with path(@A,C,2,[A,B,C]), and $\mathcal{U}_A$ now includes $\mathcal{U}_{in}$. The network queue is also updated to include $\mathcal{U}_{ext}$ (rule NodeStep).

Our operational semantics does not specify the time gaps between two consecutive reductions and, therefore, does not determine time points as associated with a concrete trace—such as $\mathcal{C} \xrightarrow{\tau} \mathcal{C}'$, where τ represents the time at which a concrete transition takes place. Instead, a trace (without time points) generated by the operational semantics—e.g., $\mathcal{C} \rightarrow \mathcal{C}'$—is an abstraction of all its corresponding annotations with time points that satisfy monotonicity. In our assertions and proofs, we use time points only to specify a relative order between events on a specific trace, so their concrete values are irrelevant.

Incremental maintenance. Now we explain in more detail how the database of a node is maintained incrementally by processing updates in its internal update list $\mathcal{U}_{in}$ one at a time. Following the strategy proposed in declarative networking [54], the rules in a NDlog program are rewritten into Δ *rules*, which can efficiently generate all the updates triggered by one update. For any given rule r that contains k body tuples, k Δ rules of the following form are generated, one for each $i \in [1, k]$.

$$\Delta p(agH) :- p_1^{\nu}(agB_1), ..., p_{i-1}^{\nu}(agB_{i-1}), \Delta p_i(agB_i), p_{i+1}(agB_{i+1}), ..., p_k(agB_k), a_1, ..., a_m$$

Δp_i in the body denotes the update currently being considered. Δp in the head denotes new updates that are generated as the result of firing this rule. Here p_i^ν denotes a tuple of name p_i in the database Ψ or the internal update list $\mathcal{U}_{in}$. In comparison, p_i (without ν) denotes a tuple of name p_i only in Ψ. For example, the Δ rules for *sp2* are:

sp2a $\Delta\mathsf{path}(@z, d, c, p) :- \Delta\mathsf{link}(@s, z, c1), \mathsf{path}(@s, d, c2, p1), c := c1 + c2, p := z::p1.$

sp2b $\Delta\mathsf{path}(@z, d, c, p) :- \mathsf{link}^\nu(@s, z, c1), \Delta\mathsf{path}(@s, d, c2, p1), c := c1 + c2, p := z::p1.$

Rules *sp2a* and *sp2b* are Δ rules triggered by updates of the link and path relation respectively. For instance, when node A processes +path(@A,C,2,[A,B,C]), only rule *sp2b* is fired. In this step, path^ν includes the tuple path(@A,C,2,[A,B,C]), while path does not. On the other hand, link^ν and link denote the same set of tuples, because $\mathcal{U}_{in}$ does not contain any tuple of name link. The rule evaluation then generates +path(@B,C,3,[B,A,B,C]), which will be communicated to node B and further triggers rule *sp2b* at node B. Such update propagates until no further new tuples are generated.

Rule Firing. We present in Figure 6 the set of rules for firing a single Δ rule given an insertion update. We write Ψ^ν to denote the table resulted from updating Ψ with the current update: $\Psi^\nu = \Psi \uplus u$.

Rule InsExists specifies the case where the tuple to be inserted (i.e. $q_i(\vec{t})$) already exists. We do not need to further propagate the update. Rule InsNew handles the case where new updates are generated by firing rule r. In order to fire a rule r, we need to map its bodies to concrete tuples in the database or the update list. We use an auxiliary function $\rho(\Psi^\nu, \Psi, r, i, \vec{t})$ to extract the complete list of substitutions for variables in the rule. Here i and $\vec{t}$ indicate that $q_i(\vec{t})$ is the current update, where q_i is the i^{th} body tuple of rule r. Every substitution σ in that set is a general unifier of the body tuples and constraints. Formally:

(1) $\vec{t} = \sigma(agB_i)$,

(2) $\forall j \in [1, i-1], \exists \vec{s}, \vec{s} = \sigma(agB_j)$ and $q_j(\vec{s}) \in \Psi^\nu$

$\boxed{\mathit{fireSingleR}(\iota, \Psi, u, \Delta r) = (\Psi', \mathcal{U}_{in}, \mathcal{U}_{ext})}$

$$\frac{(n, q_i(\vec{t})) \in \Psi}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi, [], [])} \textsc{InsExists}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota_1, ags) :- \cdots, \Delta q_i(agB_i) \cdots \\ q_i(\vec{t}) \notin \Psi \qquad ags \text{ does not contain any aggregate} \\ \Sigma = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \qquad \Sigma' = sel(\Sigma, \Psi^\nu) \qquad \mathcal{U} = genUpd(\Sigma, \Sigma', p, \Psi^\nu) \\ \text{if } \iota_1 = \iota \text{ then } \mathcal{U}_i = \mathcal{U}, \mathcal{U}_e = [] \text{ otherwise } \mathcal{U}_i = [], \mathcal{U}_e = \mathcal{U}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi, \mathcal{U}_i, \mathcal{U}_e)} \textsc{InsNew}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :- \cdots, \Delta q_i(agB_i) \cdots \qquad q_i(\vec{t}) \notin \Psi \\ ags \text{ contains an aggregate } F_{agr} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \uplus \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agr}, \Psi') = p(@\iota, \vec{s}) \qquad p(@\iota, \vec{s}) \in \Psi\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi', [], [])} \textsc{InsAggSame}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :- \cdots, \Delta q_i(agB_i) \cdots \qquad q_i(\vec{t}) \notin \Psi \\ ags \text{ contains an aggregate } F_{agr} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \uplus \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agr}, \Psi') = p(@\iota, \vec{s}) \\ p(@\iota, \vec{s}_1) \in \Psi \qquad \vec{s} \text{ and } \vec{s}_1 \text{ share the same key but different aggregate value}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi', [-p(@\iota, \vec{s}_1), +p(@\iota, \vec{s})], [])} \textsc{InsAggUpd}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :- \cdots, \Delta q_i(agB_i) \cdots \\ q_i(\vec{t}) \notin \Psi \qquad ags \text{ contains an aggregate } F_{agr} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, \vec{t}) \\ \Psi' = \Psi \uplus \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agr}, \Psi') = p(@\iota, \vec{s}) \qquad \nexists p(@\iota, \vec{s'}) \in \Psi \\ \text{such that } \vec{s} \text{ and } \vec{s'} \text{ share the same key but different aggregate value}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi', [+p(@\iota, \vec{s})], [])} \textsc{InsAggNew}$$

Figure 6: Insertion rules for evaluating a single Δ rule

(3) $\forall j \in [i+1, n], \exists \vec{s}, \vec{s} = \sigma(agB_j)$ and $q_j(\vec{s}) \in \Psi$

(4) $\forall k \in [1, m], \sigma[a_k]$ is true

We write $[a]$ to denote the constraint that a represents. When a is an assignment (i.e., $x := f(\vec{t})$), $[a]$ is the equality constraint $x = f(\vec{t})$; otherwise, $[a]$ is a.

When multiple tuples with the same key are derived using a rule, a selection function *sel* is introduced to decide which substitution to propagate. In NDlog run time, similar to a relational database, a key value of a stored tuple $p(\vec{t})$ uniquely identifies that tuple. When a different tuple $p(\vec{t'})$ with the same key is derived, the old value $p(\vec{t})$ and any tuple derived using it need to be deleted. For instance, we can demand that each pair of nodes in the network have a unique path between them. This is equivalent to designating the first two arguments of path as its key. As a result, path(A,B,1,[A,B]) and path(A,B,2,[A,D,B]) cannot both exist in the database.

We also use a *genUpd* function to generate appropriate updates based on the selected substitutions. It may generate deletion updates in addition to an insertion update of the new value. For example, assume that path(A,B,3,[A,C,D,B]) is in Ψ^{ν}. If we were to choose path(A,B,1,[A,B]) because it appears earlier in the update list, then *genUpd* returns $\{+\text{path(A,B,1,[A,B])}, -\text{path(A,B,3,[A,C,D,B])}\}$. We leave the definitions of *sel* and *genUpd* abstract here, as there are many possible strategies for implementing these two functions. Aside from the strategy of picking the first update in the queue (illustrated above), another possible strategy is to pick the last, as it is the freshest. Once the strategy of *sel* is fixed, *genUpd* is also fixed. However, the only relevant part to the logic we introduce later is that the substitutions used for an insertion update come from the ρ function, and that the substitutions satisfy the property we defined above. In other words, our program logic can be applied to a number of different implementation of *sel* and *genUpd*.

The rest of the rules in Figure 6 deal with generating an aggregate tuple. Rule InsAggNew applies when the aggregate is generated for the first time. We only need to insert the

new aggregate value to the table. Additional rules (i.e. InsAggSame and InsAggUpd) are required to handle aggregates where the new aggregate is the same as the old one or replaces the old one.

To efficiently implement aggregates, for each tuple p that has an aggregate function in its arguments, there is an internal tuple p_{agg} that records all candidate values of p. When there is a change to the candidate set, the aggregate is re-computed. For example, $\mathsf{bestpath}_{agg}$ maintains all candidate path tuples.

We also require that the location specifier of a rule head containing an aggregate function be the same as that of the rule body. With this restriction, the state of an aggregate is maintained in one single node. If the result of the aggregate is needed by a remote node, we can write an additional rule to send the result after the aggregate is computed.

Rule InsAggSame applies when the new aggregates is the same as the old one. In this case, only the candidate set is updated, and no new update is propagated. Rule InsAggUpd applies when there is a new aggregate value. In this case, we need to generate a deletion update of the old tuple before inserting the new one.

Figure 7 summaries the deletion rules. When the tuple to be deleted has multiple copies, we only reduce its reference count. The rest of the rules are the dual of the corresponding insertion rules.

We revisit the example in Figure 4 to illustrate how incremental maintenance is performed on the shortest-path program. Upon receiving +path(@A,C,2,[A,B,C]), Δ rule *sp2b* will be triggered and generate a new update +path(@B,C,3,[B,A,B,C]), which will be included in $\mathcal{U}_{ext}$ as it is destined to a remote node B (rule InsNew). The Δ rule for *sp3* will also be triggered, and generate a new update +bestPath(@A,C,2,[A,B,C]), which will be included in $\mathcal{U}_{in}$ (rule InsAggNew). After evaluating the Δ rules triggered by the update +path(@A,C,2,[A,B,C]), we have $\mathcal{U}_{in} = \{+\mathsf{bestPath(@A,C,2,[A,B,C])}\}$ and $\mathcal{U}_{ext} = \{+\mathsf{path(@B,C,3,[B,A,B,C])}\}$. In addition, $\mathsf{bestpath}_{agg}$, the auxiliary relation that maintains all candidate tuples for bestpath, is

$$\frac{(n, q_i(\vec{t})) \in \Psi \qquad n > 1}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi, [], [])}\ \textsc{DelExists}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota_1, ags) :- \cdots, \Delta q_i(agB_i) \cdots \qquad (1, q_i(\vec{t})) \in \Psi \\ ags \text{ does not contain any aggregate} \qquad \{\sigma_1, \cdots, \sigma_k\} = sel(\rho(\Psi^\nu, \Psi, r, i, \vec{t}), \Psi^\nu) \\ \mathcal{U} = [-p(@\iota_1, \sigma_1(ags)), \cdots, -p(@\iota_1, \sigma_k(ags))] \\ \text{if } \iota_1 = \iota \text{ then } \mathcal{U}_i = \mathcal{U}, \mathcal{U}_e = [] \text{ otherwise } \mathcal{U}_i = [], \mathcal{U}_e = \mathcal{U}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi, \mathcal{U}_i, \mathcal{U}_e)}\ \textsc{DelNew}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :- \cdots, \Delta q_i(agB_i) \cdots \qquad (1, q_i(\vec{t})) \in \Psi \\ ags \text{ contains an aggregate } F_{agr} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \backslash \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agr}, \Psi') = p(@\iota, \vec{s}) \qquad p(@\iota, \vec{s}) \in \Psi\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi', [], [])}\ \textsc{DelAggSame}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :- \cdots, \Delta q_i(agB_i) \cdots \qquad (1, q_i(\vec{t})) \in \Psi \\ ags \text{ contains an aggregate } F_{agr} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \backslash \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agr}, \Psi') = p(@\iota, \vec{s}) \\ p(@\iota, \vec{s}_1) \in \Psi \qquad \vec{s} \text{ and } \vec{s}_1 \text{ share the same key but different aggregate value}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi', [-p(@\iota, \vec{s}_1), +p(@\iota, \vec{s})], [])}\ \textsc{DelAggUpd}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :- \cdots, \Delta q_i(agB_i) \cdots \qquad (1, q_i(\vec{t})) \in \Psi \\ ags \text{ contains an aggregate } F_{agr} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \backslash \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agr}, \Psi') = \mathsf{NULL}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi', [-p(@\iota, \vec{s'})], [])}\ \textsc{DelAggNone}$$

Figure 7: Deletion rules for evaluating a single Δ rule

also updated to reflect that a new candidate tuple has been generated. It now includes bestpath(@A,C,2,[A,B,C]).

Discussion. The semantics introduced here will not terminate for programs with a cyclic derivation of the same tuple, even though set-based semantics will. Most routing protocols do not have such issue (e.g., cycle detection is well-adopted in routing protocols). Prior work [63] has proposed improvements to solve this issue, which is not crucial for the soundness of the proposed program logic.

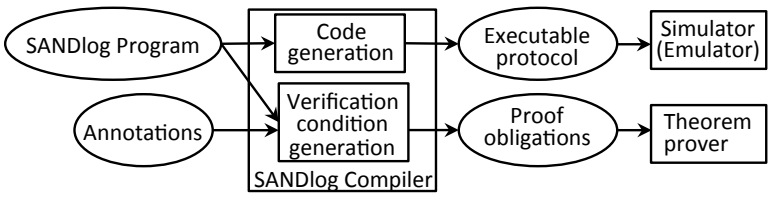
CHAPTER 3

Theorem Proving with Program Logic

In recent years, we have witnessed an explosion of services provided over the Internet. These services are increasingly transferring customers' private information over the network and used in mission-critical tasks. Central to ensuring the reliability and security of these services is a secure and efficient Internet routing infrastructure. Unfortunately, the Internet infrastructure, as it stands today, is highly vulnerable to attacks. The Internet runs the *Border Gateway Protocol* (BGP), where routers are grouped into Autonomous Systems (*AS*) administrated by Internet Service Providers (*ISP*s). Individual ASes exchange route advertisements with neighboring ASes using the *path-vector* protocol. Each originating AS first sends a route advertisement (containing a single AS number) for the IP prefixes it owns. Whenever an AS receives a route advertisement, it adds itself to the AS *path*, and advertises the best route to its neighbors based on its routing policies. Since these route advertisements are not authenticated, ASes can advertise non-existent routes or claim to own IP prefixes that they do not. These faults may lead to long periods of interruption of the Internet; best epitomized by recent high-profile attacks [25, 67].

In response to these vulnerabilities, several new Internet routing architectures and protocols for a more secure Internet have been proposed. These range from security extensions of BGP (Secure-BGP (S-BGP) [49], ps-BGP [81], so-BGP [82]), to "clean-slate" Internet architectural redesigns such as SCION [91] and ICING [60]. However, *none* of the proposals formally analyzed their security properties. These protocols are implemented from scratch, evaluated primarily experimentally, and their security properties shown via informal reasoning.

Existing protocol analysis tools [12, 27, 31] are rarely used in analyzing routing protocols because they are considerably more complicated than cryptographic protocols: they often



The round objects are code (proofs), which are the input or output of the framework. The rectangular objects are software components of the framework.

Figure 8: Architecture of a unified framework for implementing and verifying secure routing protocols.

compute local states, are recursive, and their security properties need to hold on arbitrary network topologies. As the number of models is infinite, model-checking-based tools in general cannot be used to prove the protocol secure.

To overcome this limitation, STRANDS explores a novel proof methodology to verify these protocols. First, STRANDS augments NDLog with cryptographic libraries to provide compact encoding of secure routing protocols. We call our language SANDlog (stands for *Secure and Authenticated Network Datalog*). We develop a program logic for reasoning about SANDlog programs that execute in an adversarial environment. The properties proved on a SANDlog program hold even when the program interact with potentially malicious programs in the network.

Based on the program logic, we implement a verification condition generator (VCGen), which takes as inputs the SANDlog program and user-provided annotations, and outputs intermediary proof obligations as a Coq file, where proof can be filled. VCGen is integrated into the SANDlog compiler, an cryptography-augmented extension to the declarative networking engine RapidNet [70]. The compiler is able to translate our SANDlog specification into executable code, which is amenable to implementation and evaluation.

We summarize our technical contributions:

1. We define a program logic for verifying SANDlog programs in the presence of adversaries (Section 3.2). We prove that our logic is sound.

2. We implement VCGen for automatically generating proof obligations and integrate

VCGen into a compiler for SANDlog (Section 3.3).

3. We encode S-BGP and SCION in SANDlog, verify path authenticity properties of these protocols, and run them in simulation (Section 3.4).

3.1. SANDlog

We specify secure routing protocols in a distributed declarative programming language called SANDlog. SANDlog inherits the expressiveness of NDLog, and is augmented with security primitives (e.g. asymmetric encryption) necessary for specifying secure routing protocols. The security primitives are encoded as user-defined functions in SANDlog programs. Figure 9 gives detailed explanation of these functions. Users can add additional cryptographic primitives to SANDlog based on their needs.

We can construct a more secure variant of the shortest path protocol by deploying signature authentication in the rules involving inter-node communications. For example, the following rule *sp2′* is extended from rule *sp2* in Figure 3 with inter-node communication encrypted.

$$
\begin{aligned}
sp2' \quad & \mathsf{path}(@z, d, c, p, sig) :- \\
& \mathsf{link}(@s, z, c1), \mathsf{path}(@s, d, c2, p1, sig1), c := c1 + c2, p := z::p1, \\
& \mathsf{pubK}(@s, d, pk), \mathsf{f\_verify}(p1, sig1, pk) = 1, \mathsf{privK}(@s, sk), sig := \mathsf{f\_sign}(p, sk).
\end{aligned}
$$

In rule *sp2′*, a signature *sig* for the path becomes an additional argument to the path tuple. When node s receives such a tuple, it verifies the signature of the path $\mathsf{f\_verify}(p1, sig, pk)$. When s sends out a path to its neighbor, it generates a signature by assigning $sig := \mathsf{f\_sign}(p, sk)$. Here f_sign and f_verify are user-defined asymmetric cryptographic functions,

Function	Description
f_sign_asym(*info*, *key*)	Create a signature of *info* using *key*
f_verify_asym(*info*, *sig*, *key*)	Verify that *sig* is the signature of *info* using *key*
f_mac(*info*, *key*)	Create a message authentication code of *info* using *key*
f_verifymac(*info*, *MAC*, *key*)	Verify *info* against *MAC* using *key*

Figure 9: Cryptographic functions in SANDlog

such as RSA.

The semantics of SANDlog is similar to that of NDLog. The execution of security primitives is identical to user-defined functions in NDLog.

3.2. A Program Logic for SANDlog

To verify correctness of secure routing protocols encoded in SANDlog, we introduce a program logic for SANDlog. The program logic enables us to prove program invariants—that is, properties holding throughout the execution of SANDlog programs—even if the nodes running the program interact with potential attackers, whose behaviors are unpredictable. In our case study, we show that a large number of desirable properties of secure routing protocols that we are interested in are safety properties and can be proved by introducing appropriate programs' invariant properties.

Attacker model. We assume *connectivity-bound* network attackers, a variant of the Dolev-Yao network attacker model. The attacker can perform cryptographic operations with correct keys, such as encryption, decryption, and signature generation, but is not allowed to eavesdrop or intercept packets. This attacker model manifests itself in our formal system in two places: (1) the network is modeled as connected nodes, some of which run the SANDlog program that encodes the prescribed protocol and others are malicious and run arbitrary SANDlog programs; (2) safety of cryptography is admitted as axioms.

Syntax. We use first-order logic formulas, denoted φ, as property specifications. The atoms, denoted A, include predicates and term inequalities. The syntax of the logic formulas is shown below.

$$
\begin{array}{llll}
\textit{Atoms} & A & ::= & P(\vec{t})@(\iota,\tau) \mid \mathsf{send}(\iota,\mathsf{tp}(P,\iota',\vec{t}))@\tau \mid \mathsf{recv}(\iota,\mathsf{tp}(P,\vec{t}))@\tau \\
& & \mid & \mathsf{honest}(\iota,\mathit{prog},\tau) \mid t_1 \ \mathit{bop}\ t_2 \\
\textit{Formulas} & \varphi & ::= & \top \mid \bot \mid A \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \supset \varphi_2 \mid \neg\varphi \mid \forall x.\varphi \mid \exists x.\varphi \\
\textit{Variable Ctx} & \Sigma & ::= & \cdot \mid \Sigma, x \qquad\qquad \textit{Logical Ctx} \quad \Gamma ::= \cdot \mid \Gamma, \varphi
\end{array}
$$

Predicate $P(\vec{t})@(\iota,\tau)$ means that tuple $P(\vec{t})$ is derived at time τ by node ι. The first

$$\frac{\Sigma;\Gamma\vdash\varphi \qquad \Sigma;\Gamma,\varphi\vdash\varphi'}{\Sigma;\Gamma\vdash\varphi'}\ \textsc{Cut} \qquad \frac{\varphi\in\Gamma}{\Sigma;\Gamma\vdash\varphi}\ \textsc{Init} \qquad \frac{\Sigma;\Gamma,\varphi\vdash\cdot}{\Sigma;\Gamma\vdash\neg\varphi}\ \neg\text{I} \qquad \frac{\Sigma;\Gamma\vdash\neg\varphi}{\Sigma;\Gamma,\varphi\vdash\cdot}\ \neg\text{E}$$

$$\frac{\Sigma;\Gamma\vdash\varphi_1 \qquad \Sigma;\Gamma\vdash\varphi_2}{\Sigma;\Gamma\vdash\varphi_1\wedge\varphi_2}\ \wedge\text{I} \qquad \frac{i\in[1,2],\Sigma;\Gamma\vdash\varphi_1\wedge\varphi_2}{\Sigma;\Gamma\vdash\varphi_i}\ \wedge\text{E} \qquad \frac{i\in[1,2],\Sigma;\Gamma\vdash\varphi_i}{\Sigma;\Gamma\vdash\varphi_1\vee\varphi_2}\ \vee\text{I}$$

$$\frac{\Sigma;\Gamma\vdash\varphi_1\vee\varphi_2 \qquad \Sigma;\Gamma,\varphi_1\vdash\varphi \qquad \Sigma;\Gamma,\varphi_2\vdash\varphi}{\Sigma;\Gamma\vdash\varphi}\ \vee\text{E}$$

$$\frac{\Sigma,x;\Gamma\vdash\varphi}{\Sigma;\Gamma\vdash\forall x.\varphi}\ \forall\text{I} \qquad \frac{\Sigma;\Gamma\vdash\forall x.\varphi}{\Sigma;\Gamma\vdash\varphi[t/x]}\ \forall\text{E} \qquad \frac{\Sigma;\Gamma\vdash\varphi[t/x]}{\Sigma;\Gamma\vdash\exists x.\varphi}\ \exists\text{I}$$

$$\frac{\Sigma;\Gamma\vdash\exists x.\varphi \qquad \Sigma,a;\Gamma,\varphi[a/x]\vdash\varphi' \qquad a \text{ is fresh}}{\Sigma;\Gamma\vdash\varphi'}\ \exists\text{E}$$

Figure 10: Rules in first-order logic.

element in $\vec{t}$ is a location identifier ι', which may be different from ι. When a tuple $P(\iota', ...)$ is derived at node ι, it is sent to ι'. This *send* action is captured by predicate $\mathsf{send}(\iota, \mathsf{tp}(P, \iota', \vec{t}))@\tau$. Correspondingly, predicate $\mathsf{recv}(\iota, \mathsf{tp}(P, \vec{t}))@\tau$ denotes that node ι has received a tuple $P(\vec{t})$ at time τ. A user could determine send and recv tuples by inspecting rules whose head tuple locates differently from body tuples. For example, the head tuple $\mathsf{path}(@z, d, c, p)$ in the rule *sp2* of the shortest-path program (Figure 3) corresponds to a tuple $\mathsf{send}(s, \mathsf{tp}(path, z, (z, d, c, p)))@t$ in our logic. $\mathsf{honest}(\iota, prog(\iota), \tau)$ means that node ι starts to run program $prog(\iota)$ at time τ. Since predicates take time points as an argument, we are effectively encoding linear temporal logic (LTL) in first-order logic [45]. The domain of the time points is the set of natural numbers. Each time point represents the number of clock ticks from the initialization of the system.

Logical judgments. The logical judgments in our program logic use two contexts: context Σ containing all the free variables; and context Γ containing logical assumptions.

(1) $\Sigma;\Gamma\vdash\varphi$ (2) $\Sigma;\Gamma\vdash prog(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e)$

Judgment (1) states that φ is provable given the assumptions in Γ. Judgment (2) is an

$\boxed{\Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e)}$

$$\begin{array}{l}\forall r \in \mathit{rlOf}(\mathit{prog}), \qquad (r = h(\vec{v}) :- p_1(\vec{s}_1), ..., p_m(\vec{s}_m), q_1(\vec{u}_1), ..., q_n(\vec{u}_n), a_1, ..., a_k) \\ \quad \Sigma;\Gamma \vdash \forall i, \forall t, \forall \vec{y}, \qquad (\ \vec{y} = \mathsf{fv}(r)) \\ \qquad \bigwedge_{j\in[1,m]} (p_j(\vec{s}_j)@(i,t) \wedge \varphi_{p_j}(i,t,\vec{s}_j)) \wedge \\ \qquad \bigwedge_{j\in[1,n]} \mathsf{recv}(i, \mathsf{tp}(q_j, \vec{u}_j))@t \wedge \qquad \supset \qquad \varphi_h(i,t,\vec{v}) \\ \qquad \bigwedge_{j\in[1,k]} [a_j]\end{array}$$

$$\frac{\forall p \in \mathit{hdOf}(\mathit{prog}),\ \varphi_p \text{ is closed under trace extension}}{\Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}.\ \bigwedge_{p\in \mathit{hdOf}(\mathit{prog})} \forall t, \forall \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i,t) \supset \varphi_p(i,t,\vec{x})}\ \textsc{Inv}$$

$\boxed{\Sigma;\Gamma \vdash \varphi}$

$$\frac{\Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e) \quad \Sigma;\Gamma \vdash \mathsf{honest}(\iota, \mathit{prog}(\iota), t)}{\Sigma;\Gamma \vdash \forall t', t' > t, \varphi(\iota, t, t')}\ \textsc{Honest}$$

Figure 11: Rules in program logic

assertion about SANDlog programs, i.e., a program invariant. We write $\varphi(\vec{x})$ when $\vec{x}$ are free in φ. $\varphi(\vec{t})$ denotes the resulting formula of substituting $\vec{t}$ for $\vec{x}$ in $\varphi(\vec{x})$. Recall that *prog* is parametrized over the identifier of the node it runs on. The program invariant is parametrized over not only the node ID i, but also the starting point of executing the program (y_b) and a later time point y_e. Judgment (2) states that any trace $\mathcal{T}$ containing the execution of a program *prog* by a node ι, starting at time τ_b, satisfies $\varphi(\iota, \tau_b, \tau_e)$, for any time point τ_e later than τ_b. Note that the trace could also contain threads that run malicious programs. Since τ_e is any time after τ_b (the time *prog* starts), φ is an invariant property of *prog*.

Inference rules. The inference rules of our program logic include all standard first-order logic ones (e.g. Modus ponens), shown in Figure 10. Reasoning about the ordering between time points are carried out in first-order logic using theory on natural numbers (in Coq, we use Omega). We choose first-order logic because it is better supported by proof assistants,

such as Coq.

In addition, we introduce two key rules (Figure 11) into our proof system. Rule Inv proves an invariant property of a program *prog*. The program invariant takes on a specific form as the conjunction of all the invariants of the tuples derived by *prog*, and means that if any head tuple is derived by *prog*, then its associated property should hold; formally: $\forall t, \forall \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i,t) \supset \varphi_p(i,t,\vec{x})$, where p is the name of the head tuple, and $\varphi_p(i,t,\vec{x})$ is an invariant property associated with $p(\vec{x})$. For example, p can be path, and $\varphi_p(i,t,\vec{x})$ be that every link in argument *path* must have existed in the past. In the INV rule, the function *rlOf*(*prog*) returns rules generating derivation tuples for a given program, and the function $\mathsf{f}v(r)$ returns all free variables in a given rule.

Intuitively, the premises of Inv need to establish that *each* derivation rule's body tuples and its associative invariants together imply the invariant of the rule's head tuple. For each derivation rule r in *prog*, we assume that the body of r is arranged so that the first m tuples (i.e. $p_1(\vec{s}_1), ..., p_m(\vec{s}_m)$) are derived by *prog*, the next n tuples (i.e. $q_1(\vec{u}_1), ..., q_n(\vec{u}_n)$) are received from the network, and constraints (i.e. $a_1, ..., a_k$) constitute the rest of the body. For tuples derived by *prog* (i.e. p_j's), we can safely assume that their invariants φ_{p_j} hold at time t. On the other hand, properties of received tuples (i.e. q_j) are excluded from the premises, as in adversarial environment, these messages are not trusted by default.

Each premise of the INV rule provides the strongest assumption that allows us to prove the conclusion in that premise. In most cases, arithmetic constraints are enough for proving the invariant. But in some special cases—for example, the invariant explicitly specifies the existence of a received tuple—the predicate representing the action of a tuple receipt is needed in the assumption. In other words, φ_{p_j} is the inductive hypothesis in this inductive proof. In our case study, we frequently need to invoke the inductive hypothesis to complete the proof.

We make sure that each tuple in an SANDlog program is either derived locally or received

from the network, but not both. For a program that violates this property, the user can rewrite the program by creating a copy tuple of a different name for the tuple that can be both derived locally or received from the network. For example, the path tuple in the shortest-path program in Figure 3 could be both derived locally (rule *sp1*) and received from a remote node (rule *sp2*). The user could rewrite the head tuple path in *sp2* to recvPath to differentiate it from path. In this way, the invariant property associated with the path tuple can be trusted and used in the proof of the program invariant.

We also require that an invariant φ_p be closed under trace extension. Formally: if $\mathcal{T} \models \varphi(\iota, t, \vec{s})$ and $\mathcal{T}$ is a prefix of $\mathcal{T}'$, then $\mathcal{T}' \models \varphi(\iota, t, \vec{s})$. For instance, the property that node ι has received a tuple p before time t is closed under trace extension, while the property that node ι never sends p to the network is not closed under trace extension.

We do not allow invariants to be specified over base tuples. The INV rule cannot be used to derive properties of base rules (e.g., link), because the function $rlOf()$ only returns rules for derivation tuples.

As an example, we use INV to prove a simple program invariant of the shortest-path program in Figure 3. The property is specified as

$$
\begin{aligned}
&\varphi_{sp} = prog(x) : \{x, y_b, y_e\}.\\
&\qquad (\forall t, \forall y, \forall c, \forall pt, y_b \leqslant t < y_e \wedge\\
&\qquad\quad \mathsf{path}(x, y, c, pt)@(x, t) \supset\\
&\qquad\quad\quad (\exists z, c', \mathsf{link}(x, z, c')@(x, t) \vee\\
&\qquad\quad\quad\quad \mathsf{link}(z, x, c'@(z, t))) \wedge\\
&\qquad (\forall t, \forall y, \forall c, \forall pt, y_b \leqslant t < y_e \wedge\\
&\qquad\quad \mathsf{bestPath}(x, y, c, pt)@(x, t) \supset \mathsf{true}
\end{aligned}
$$

Intuitively, φ_{sp} specifies an invariant property for the path tuple, which says a path tuple must imply a path tuple to/from the direct neighbor. φ_{sp} also assigns true as the invariant property for bestPath tuples. The proof is established using INV (Figure 12). The whole proof has three premises, each corresponding to a rule in the shortest-path program in

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \forall s, \forall d, \forall c, \forall t, \\ \quad (\mathsf{link}(s,d,c)@(s,t) \wedge p = [s,d]) \supset \\ \quad\quad (\exists z, c', \mathsf{link}(s,z,c')@(s,t) \vee \mathsf{link}(z,s,c')@(z,t)) \\ \\ \Sigma;\Gamma \vdash \forall s, \forall d, \forall c1, \forall c2, \forall p1, \forall z, \forall t, \\ \quad (\mathsf{link}(s,z,c1)@(s,t) \wedge \mathsf{recv}(s, \mathsf{tp}(\mathsf{path}, s, d, c2, p1))@t \wedge \\ \quad c = c1 + c2 \wedge p = z{::}p1) \supset \\ \quad\quad (\exists z'', c'', \mathsf{link}(z,z'',c'')@(z,t) \vee \mathsf{link}(z'',z,c')@(z'',t)) \\ \\ \Sigma;\Gamma \vdash \mathsf{true}\end{array}}{\Sigma;\Gamma \vdash \varphi_{sp}}\ \textsc{Inv}$$

Figure 12: Proof of φ_{sp}

Figure 3. For example, in the second premise corresponding to *sp2*, we include the local link tuple and the received path as well as constraints in the assumption, while leaving out the invariant property of the path tuple, because a received path tuple should not be trusted in an adversarial environment.

The Honest rule proves properties of the entire system based on the program invariant. If $\varphi(i, y_b, y_e)$ is the invariant of *prog*, and a node ι runs the program *prog* at time t_b, then any trace *containing* the execution of this program satisfies $\varphi(\iota, t_b, t_e)$, where t_e is a time point after t_b. SANDlog programs never terminate: after the last instruction, the program enters a stuck state. The Honest rule is applied to honest principles (nodes) that execute the prescribed protocols. The invariant property of an honest node holds even when it interacts with other malicious nodes in the network, which is required by the soundness of the inference rules, as explained below.

Soundness. We prove the soundness of our logic with regard to the trace semantics. First, we define the trace-based semantics for our logic and judgments in Figure 13. Different from semantics of first-order logic, in our semantics, formulas are interpreted on a trace $\mathcal{T}$. We

$\mathcal{T} \vDash P(\vec{t})@(\iota, \tau)$ iff $\exists \tau' \leqslant \tau$, $\mathcal{C}$ is the configuration on $\mathcal{T}$ prior to time τ',
$(\iota, \Psi, \mathcal{U}, prog(\iota)) \in \mathcal{C}$, at time τ', $(\iota, \Psi, \mathcal{U}, prog(\iota)) \hookrightarrow (\iota, \Psi', \mathcal{U}' \circ \mathcal{U}_{in}, prog(\iota)), \mathcal{U}_e$,
and either $P(\vec{t}) \in \mathcal{U}_{in}$ or $P(\vec{t}) \in \mathcal{U}_e$

$\mathcal{T} \vDash \mathsf{send}(\iota, \mathsf{tp}(P, \iota', \vec{t}))@\tau$ iff $\mathcal{C}$ is the configuration on $\mathcal{T}$ prior to time τ,
$(\iota, \Psi, \mathcal{U}, prog(\iota)) \in \mathcal{C}$, at time τ, $(\iota, \Psi, \mathcal{U}, prog(\iota)) \hookrightarrow \mathcal{S}', \mathcal{U}_e$ and $P(@\iota', \vec{t}) \in \mathcal{U}_e$

$\mathcal{T} \vDash \mathsf{recv}(\iota, \mathsf{tp}(P, \vec{t}))@\tau$ iff $\exists \tau' \leqslant \tau$, $\mathcal{C} \xrightarrow{\tau'} \mathcal{C}' \in \mathcal{T}$,
$\mathcal{Q}$ is the network queue in $\mathcal{C}$, $P(\vec{t}) \in \mathcal{Q}$, $(\iota, \Psi, \mathcal{U}, prog(\iota)) \in \mathcal{C}'$ and $P(\vec{t}) \in \mathcal{U}$

$\mathcal{T} \vDash \mathsf{honest}(\iota, prog(\iota), \tau)$ iff at time τ, node ι's local state is (ι, [], [], *prog* (ι))

$\Gamma \vDash prog(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e)$ iff Given any trace $\mathcal{T}$ such that $\mathcal{T} \vDash \Gamma$,
and at time τ_b, node ι's local state is (ι, [], [], $prog(\iota)$)
given any time point τ_e such that $\tau_e \geqslant \tau_b$, it is the case that $\mathcal{T} \vDash \varphi(\iota, \tau_b, \tau_e)$

Figure 13: Trace-based semantics

elide the rules for first-order logic connectives. A tuple $P(\vec{t})$ is derivable by node ι at time τ, if $P(\vec{t})$ is either an internal update or an external update generated at a time point τ' no later than τ. A node ι sends out a tuple $P(\iota', \vec{t})$ if that tuple was derived by node ι. Because ι' is different from ι, it is sent over the network. A *received tuple* is one that comes from the network (obtained using DeQueue). Finally, an honest node ι runs *prog* at time τ, if at time τ and the local state of ι at time τ is the initial state with an empty table and update queue.

The semantics of invariant assertion states that if a trace $\mathcal{T}$ contains the execution of *prog* by node ι (formally defined as the node running *prog* is one of the nodes in the configuration $\mathcal{C}$), then given any time point τ_e after τ_b, the trace $\mathcal{T}$ satisfies $\varphi(\iota, \tau_b, \tau_e)$. Here, the semantic definition requires that the invariant of an honest node holds in the presence of attackers, because we examine all traces that include the honest node in their configurations. This means that those traces can contain arbitrary other nodes, some of which are malicious.

Our program logic is proven to be sound with regard to the trace semantics:

Theorem 1 (Soundness)**.** *1. If $\Sigma; \Gamma \vdash \varphi$, then for all grounding substitution σ for Σ, given any trace $\mathcal{T}$, $\mathcal{T} \vDash \Gamma\sigma$ implies $\mathcal{T} \vDash \varphi\sigma$;*

2. If $\Sigma; \Gamma \vdash prog(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e)$, then for all grounding substitution σ for Σ, $\Gamma\sigma \vDash (prog)\sigma(i) : \{i, y_b, y_e\}.(\varphi(i, y_b, y_e))\sigma$.

Proof. By mutual induction on the derivation $\mathcal{E}$. The rules for standard first-order logic formulas are straightforward. We focus on the case when $\mathcal{E}$ ends in the Honest rule.

Case: The last step of $\mathcal{E}$ is Honest.

$$\mathcal{E} = \frac{\mathcal{E}_1 :: \Sigma; \Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e) \qquad \mathcal{E}_2 :: \Sigma; \Gamma \vdash \mathsf{honest}(\iota, \mathit{prog}(\iota), t)}{\Sigma; \Gamma \vdash \forall t', t' > t, \varphi(\iota, t, t')} \textsc{Honest}$$

Given σ, $\mathcal{T}$ s.t. $\mathcal{T} \vDash \Gamma\sigma$, by I.H. on $\mathcal{E}_1$ and $\mathcal{E}_2$

(1) $\Gamma\sigma \vDash (\mathit{prog})\sigma(i) : \{i, y_b, y_e\}.(\varphi(i, y_b, y_e))\sigma$

(2) $\mathcal{T} \vDash (\mathsf{honest}(\iota, \mathit{prog}(\iota), t))\sigma$

By (2),

(3) at time $t\sigma$, $\iota\sigma$ starts to run program $((\mathit{prog})\sigma)$

By (1) and (3), given any T s.t. $T > t\sigma$

(4) $\mathcal{T} \vDash \varphi\sigma(\iota\sigma, t\sigma, T)$

Therefore,

(5) $\mathcal{T} \vDash (\forall t', t' > t, \varphi(\iota, t, t'))\sigma$

Case: $\mathcal{E}$ ends in Inv rule.

Given $\mathcal{T}$, σ such that $\mathcal{T} \vDash \Gamma\sigma$, and at time τ_b, node ι's local state is (ι, [], [], $\mathit{prog}(\iota)$), given any time point τ_e such that $\tau_e \geqslant \tau_b$,

let $\varphi = (\bigwedge_{p \in hdOf(prog)} \forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$

we need to show $\mathcal{T} \vDash \varphi$

By induction on the length of $\mathcal{T}$

subcase: $|\mathcal{T}| = 0$, $\mathcal{T}$ has one state and is of the form $\xrightarrow{\tau} \mathcal{C}$

By assumption (ι, [], [], $[\mathit{prog}]_\iota$) $\in \mathcal{C}$

Because the update list is empty, $\nexists\sigma_1$, s.t. $\mathcal{T} \vDash (p(\vec{x})@(\iota, t))\sigma\sigma_1$

Therefore, $\mathcal{T} \vDash \varphi$ trivially.

subcase: $\mathcal{T} = \mathcal{T}' \xrightarrow{\tau} \mathcal{C}$

We examine all possible steps allowed by the operational semantics.

To show the conjunction holds, we show all clauses in the conjunction are true by

construct a generic proof for one clause.

case: DeQueue is the last step.

Given a substitution σ_1 for t and $\vec{x}$ s.t. $\mathcal{T} \vDash (\tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t))\sigma\sigma_1$

By the definitions of semantics, and DeQueue merely moves messages around

(1) $(p(\vec{x}))\sigma\sigma_1$ is on trace $\mathcal{T}'$

(2) $\mathcal{T}' \vDash (\tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t))\sigma\sigma_1$

By I.H. on $\mathcal{T}'$,

(3) $\mathcal{T}' \vDash (\forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$

By (2) and (3)

(4) $\mathcal{T}' \vDash \varphi_p(\iota, t, \vec{x})\sigma\sigma_1$

By φ_p is closed under trace extension and (4),

$\mathcal{T} \vDash \varphi_p(\iota, t, \vec{x})\sigma\sigma_1$

Therefore, $\mathcal{T} \vDash \varphi$ by taking the conjunction of all the results for such p's.

case: NodeStep is the last step. Similar to the previous case, we examine every tuple p generated by *prog* to show $\mathcal{T} \vDash \varphi$. When p was generated on $\mathcal{T}'$, the proof proceeds in the same way as the previous case. We focus on the cases where p is generated in the last step.

We need to show that $\mathcal{T} \vDash (\forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$

Assume the newly generated tuple is $(p(\vec{x})@(\iota, \tau_p))\sigma\sigma_1$, where $\tau_p \geqslant \tau$

We need to show that $\mathcal{T} \vDash (\varphi_p(\iota, \tau_p, \vec{x}))\sigma\sigma_1$

subcase: Init is used

In this case, only rules with an empty body are fired ($r = h(\vec{v}) :- .$).

By expanding the last premise of the Inv rule, and $\vec{v}$ are all ground terms,

(1) $\mathcal{E}_1 :: \Sigma; \Gamma \vdash \forall i, \forall t, \varphi_h(i, t, \vec{v})$

By I.H. on $\mathcal{E}_1$

(2) $\mathcal{T} \vDash (\forall i, \forall t, \varphi_h(i, t, \vec{v}))\sigma$

By (2)

$\mathcal{T} \vDash (\varphi_h(\iota, \tau_p, \vec{y}))\sigma\sigma_1$

subcase: RuleFire is used.

We show one case where p is not an aggregate and one where p is.

subsubcase: InsNew is fired

By examine the Δr rule,

(1) exists $\sigma_0 \in \rho(\Psi^\nu, \Psi, r, k, \vec{s})$ such that $(p(\vec{x})@(\iota, t))\sigma\sigma_1 = (p(\vec{v})@(\iota, t))\sigma_0$

(2) for tuples (p_j) derived by node ι, $(p_j(\vec{s_j}))\sigma_0 \in \Psi^\nu$ or $(p_j(\vec{s_j}))\sigma_0 \in \Psi$

By operational semantics, p_j must have been generated on $\mathcal{T}'$

(3)$\mathcal{T}' \vDash (p_j(\vec{s}_j)@(\iota, \tau_p))\sigma_0$

By I.H. on $\mathcal{T}'$ and (3), the invariant for p_j holds on $\mathcal{T}'$

(4) $\mathcal{T}' \vDash (\varphi_{p_j}(\iota, \tau_p, \vec{s}_j))\sigma_0$

By φ_p is closed under trace extension

(5) $\mathcal{T} \vDash (p_j(\vec{x}_j)@(\iota, \tau_p) \wedge \varphi_{p_j}(\iota, \tau_p, \vec{s}_j))\sigma_0$

For tuples (q_j) that are received by node ι, using similar reasoning as above

(6) $\mathcal{T} \vDash (\mathsf{recv}(i, \mathsf{tp}(q_j, \vec{s}_j))@\tau_p)\sigma_0$

(7) For constraints (a_j), $\mathcal{T} \vDash a_j\sigma_0$

By I.H. on the last premise in Inv and (5) (6) (7)

(8) $\mathcal{T} \vDash (\varphi_p(i, \tau_p, \vec{v}))\sigma_0$

By (1) and (8), $\mathcal{T} \vDash (\varphi_p(i, \tau_p, \vec{y}))\sigma\sigma_1$

subsubcase: InsAggNew is fired.

When p is an aggregated predicate, we additionally prove that

every aggregate candidate predicate p_{agg} has the same invariant as p.

That is (1) $\mathcal{T} \vDash (\forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p_{agg}(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$

The reasoning is the same as the previous case.

We additionally show that (1) is true on the newly generated $p_{agg}(\vec{t})$.

The intuition behind the soundness proof is that the invariant properties φ_p specified for the predicate p are local properties that will not be affected by the attacker. For instance, we can specify basic arithmetic constraints of arguments derived by the honest node and the existence of base tuples. These invariants can be checked by examining the program of the honest node and are not affected by how the honest node interacts with the rest of the nodes in the network. We never use any invariant of received tuples, because they could be sent from an attacker, and the attacker does not need to generate those tuples following protocols. However, we can use the fact that those received tuples must have arrived at the honest node; otherwise, the rule will not fire. In other words, we trust the runtime of an honest node.

Discussion. Our program logic enables us to prove invariant properties that hold even in adversarial environment. The network trace $\mathcal{T}$ in Theorem 1 could involve attacker threads who run arbitrary malicious programs. For example, a trace may contain attacker threads who keep propagating invalid route advertisement for a non-existent destination. Properties proved with our logic, however, still hold in such traces. The key observation here is that in the rule Inv, the correctness of the program property does not rely on received tuples, which could have been manipulated by malicious attackers. This guarantee is further validated by our logic semantics and soundness, where we demand that a proved conclusion should hold in *any* trace.

Our program logic could possibly prove false program invariants for SANDlog programs only generating empty network traces. An example program is as follows:

$$r1 \quad \mathsf{p}(@a) :- \mathsf{q}(@a).$$
$$r2 \quad \mathsf{q}(@a) :- \mathsf{p}(@a).$$

A user could assign false to both p and q, and prove the program invariant with the rule Inv. However, this program, when executing in bottom-up evaluation, produces an empty set of

tuples. The Inv rule is still sound in this case as there is no trace that generates tuples p and q. Instead, a SANDlog program should have rules of the form "p :-" to generate base tuples. If a false program invariant is given for such a program, the user is obliged to prove $\vdash$ false in the logic, which is impossible.

3.3. Verification Condition Generator

Our prototype implementation is built on top of RapidNet [59], a declarative networking engine. RapidNet takes a SANDlog program as input, rewrites each rule into a set of Δ rules (see Section 2.1.2), and compiles each Δ rule into a series of relational operators, such as projection, selection and join.

We augmented RapidNet with libraries handling cryptographic functions. We also equipped RapidNet with a verification condition generator (VCG), which extracts proof obligations from SANDlog programs into Coq's logic (i.e., shallow embedding). We could target other interactive theorem provers such as Isabelle HOL as well.

Library extensions to RapidNet are implemented similarly to those described in prior work [93]. Thus, we focus on describing the implementation of VCG, which generates lemmas corresponding to the last premise of the rule Inv. VCG takes as input, the abstract syntax tree of a SANDlog program *sp* and its user-defined type annotation *tp*, and outputs a Coq file that contains (1) definitions for types, predicates, and functions; (2) lemmas for rules in the SANDlog program; and (3) axioms based on Honest rule. The annotation *tp* is supposed to be provided by the programmer, and contains typing information of the form $(arg : type)$, for all arguments in relations and functions. *tp* is necessary for VCG, because SANDlog is untyped. Next we illustrate the process of shallow embedding in detail.

Generating definitions. VCG uses Algorithm 1 to generate four kinds of definitions: types, predicates, functions and invariant properties. VCG scans through the types in the type annotation file and inserts one type definitions for each distinct type (lines 4 – 9). For instance, given $(IPAddress : string)$ in the annotation, VCG generates Variable string: Type.

Definition for the type time is hard-coded .

Lines 10 to 20 generate definitions for relations. For each distinct relation name p in a SANDlog program, we define a predicate of the same name, whose arguments are composed of the arguments of p, a node identifier, and a time point. For example, with link$(@n, m)$ and the type annotation:(n:string) (m:string), VCG outputs the following predicate definition. It represents the predicate link$(n, m)@(n, t)$.

```
Variable link: string → string → string → time → Prop.
```

Lines 21 to 29 generate definitions for user-defined functions. For each user-defined function, we define a data constructor of the same name, unless it corresponds to a Coq's built-in operator. We also include a time point as an additional argument. For example, VCG generates the following definition for $f\_sign(info, key)$ with typing information: ($info$:string)(key:string) (f_sign():string).

```
Variable fsign: string → string → time → string.
```

VCG also defines a skeleton for invariant properties of the relations in each rule head (lines 30 – 38). The arguments of an invariant property are the same as those in the relation, plus a location specifier and a time point. VCG generates a question mark as the place holder for the concrete definition of the invariant properties, which will be provided by the user. As an example, consider the rule head of *sp1* in Figure 3 of Chapter 2. The invariant property of the relation *path(@s, d, c, p)*, with typing information (s:string) (d:string) (c:nat)(p:string),is defined as follows:

```
Definition p-path: (attr_0:string)(attr_1:string)(attr_2:nat)(attr_3:string)

                   (loc:string)(t:time): Prop:=?
```

Generating lemma statements. Let us use the program in Figure 3 of Chapter 2 as an example. The lemma generated for *sp1* is shown below, where p-path represents the

invariant associated with the path tuple. Next we explain how it is generated in detail. The pseudo-code is shown in Algorithm 2. The algorithm writes to a Coq file using the keyword Write. All the arguments to Write are strings. For simplicity, we omit the double quotes around strings and string operations.

Lemma r1: forall(s:node)(d:node)(c:nat)(p:list node)(t:time),

```
link s d c s t → p = cons (s (cons d nil)) → p-path s t s d c p t.
```

VCG first fetches a rule (lines 5 – 6). Next, it generates a unique name for the lemma and universal quantification of free variables in that rule (lines 7 – 11). In this phase, VCG generates the first line of the lemma shown above, assuming the annotated *sp1* is:

$\mathsf{path}(@(s : \mathsf{node}), (d : \mathsf{node}), (c : \mathsf{nat}), (p : \mathsf{list[node]})) :- \mathsf{link}(@s, d, c), p := [s, d].$

Next, VCG processes the rule body by applying the function ParsingBody (lines 20 – 38) to each element in it. Elements are separated by commas. The input to ParsingBody is either a relation, an assignment, or a binary operation. When processing a relation, VCG generates (1) the predicate that corresponds to the relation (lines 22 – 25) and (2) the invariant associated with the relation (lines 26 – 31). For assignments and binary operations, VCG translates them into corresponding operations in Coq (lines 32 – 37).

ParseBody function uses ParsingTerm (Algorithm 3) to process expressions in the body elements. There are four forms of expressions: variables, constances, functions, and arithmetic operations. A variable or constance is translated as it is (lines 2 – 5). A function (lines 6 – 15) can either have a correspondent in Coq libraries or be defined by VCG; VCG generates the definition for a function only when no correspondent can be found in Coq libraries. For the former case, VCG translates the function into the corresponding form in Coq (e.g. $f\_removeFirst(l) \Rightarrow tl(l)$); for the latter one, VCG simply applies the defined function to the function arguments (e.g. $f\_sign(info, key) \Rightarrow f\_sign\ info\ key\ t$). In both cases, each argument itself is an expression, and is processed recursively with ParsingTerm. As an example, the second line of Lemma $r1$ before p-path is the result of processing the rule body

of *sp1*.

Finally, VCG generates the conclusion of the lemma – which is the invariant associated with the rule head (lines 14 – 18) – and complete the translation process. Users can instruct VCG to generate multiple sets of lemmas over the same SANDlog program, if there are multiple invariants to be proved.

Axioms. For each invariant φ_p of a rule head p, VCG produces an axiom (Algorithm 4) , in the form $\forall i, t, \vec{x}, \mathsf{Honest}(i) \supset p(\vec{x})@(i,t) \supset \varphi_p(i, \vec{x})$, where $\mathsf{Honest}(n) \triangleq \mathsf{honest}(n, prog, -\infty)$. For example, consider the rule *sp1* in Figure 3 of Chapter 2. The corresponding axiom generated by VCG is as follows:

```
Axiom geneProp-path: (attr_0:string)(attr_1:string) (attr_2:nat)

                     (attr_3:string)(loc:string)(t:time),

                     Honest loc → path attr_0 attr_1 attr_2 attr_3 loc t →

                     p-path attr_0 attr_1 attr_2 attr_3 loc t.
```

These axioms are conclusions derived from the Honest rule after invariants are verified. Soundness of these axioms is backed by the soundness theorem (Theorem 1). Since we always assume that the program starts at time $-\infty$, the condition that $t > -\infty$ is always true, thus omitted.

3.4. Case Studies

In this section, we investigate two proposed secure routing solutions: S-BGP (Section 3.4.1) and SCION (Section 3.4.2). We encode both solutions in SANDlog and prove that they preserve route authenticity, a key property stating that route announcements are trustworthy. Our case studies not only demonstrate the effectiveness of our program logic, but provide a formal proof supporting the informal guarantees given by the solution designers. Interested readers can find SANDlog specification and formal verification of both solutions

Algorithm 1 Generate Proof Obligation – Definitions

function GeneDefition(sp, tp)
    (* *sp: SANDlog program* *)
    (* *tp: Type annotation* *)
    (* *time is a type* *)
    Write Definition $time$:= nat.
    (* *Type definition* *)
    for all $(v : type) \in tp$ **do**
        if "Variable *type*" not defined **then**
            Write Variable *type*: Type.
    (* *Predicate definition* *)
    Write Variable *Honest*: *string* $\rightarrow$ *Prop*.
    for all $p(@\iota, \vec{x}) \in sp$ **do**
        if "Variable p" not defined **then**
            Write Variable p:
            $id\_type' \leftarrow tp(\iota)$
            Write $id\_type' \rightarrow$
            for all $arg \in \vec{x}$ **do**
                $type' \leftarrow tp(arg)$
                Write $type' \rightarrow$
            Write $id\_type' \rightarrow time \rightarrow Prop$.
    (* *Function definition* *)
    for all $(f\_name(\vec{y})) \in sp$ **do**
        if f_name has no correspondents in Coq library &
          "Variable f_name" not defined **then**
            Write Variable f_name:
            for all $arg \in \vec{y}$ **do**
                $type'' \leftarrow tp(arg)$
                Write $type'' \rightarrow$
            $type''' \leftarrow tp(f\_name(\vec{y}))$
            Write $type'''$.
    (* *Invariant definition* *)
    for all $p(@\iota, \vec{x}) \in HeadTuple(sp)$ **do**
        Write Definition p-p:
        $id\_type' \leftarrow tp(\iota)$
        Write (attr_0 : $type'$)
        for $i \leftarrow 1, len(\vec{x})$ **do**
            $type' \leftarrow tp(\vec{x}(i))$
            Write (attr_i : $type'$)
        Write $(loc : id\_type')(t : time) : Prop :=?$
end function

Algorithm 2 Generate Proof Obligation – Lemma statements.

```
1: function GeneLemma(sp, tp)
2:     (* sp: SANDlog program *)
3:     (* tp: Type annotation *)
4:     for i ← 1, NumberOfRules(sp) do
5:         (H :− body) ← Rules(sp)(i)
6:         Where H = ph(@ι′, y⃗)
7:         Write Lemma ri: forall
8:         (* Quantify variables *)
9:         for all arg ∈ fv(H, body) do
10:            type′ ← tp(arg)
11:            Write (arg:type)
12:        for all ele ∈ body do
13:            ParsingBody(ele)
14:        Write p-ph
15:        for all arg ∈ y⃗ do
16:            ParsingTerm(arg)
17:        (* add node id and time as arguments *)
18:        Write ι t.
19: end function
20: function ParsingBody(body_piece)
21:     if body_piece = p(@ι, x⃗) then
22:         Write pred_p
23:         for all arg ∈ (ι, x⃗) do
24:             ParsingTerm(arg)
25:         Write ι t →
26:         if p(@ι, x̃) ∈ HeadTuple(sp) &
27:            p(@ι, x⃗) not a received tuple then
28:             Write prop_p
29:             for all arg ∈ (ι, x⃗) do
30:                 ParsingTerm(arg)
31:             Write ι t →
32:     else if body_piece = (a := b) or
33:                          (a bop b) then
34:         ParsingTerm(a)
35:         if a := b then Write =
36:         else Write bop
37:         ParsingTerm(b)
38: end function
```

Algorithm 3 Generate Proof Obligation – Function ParsingTerm()

```
1: function ParsingTerm(exp)
2:     if exp is variable x then
3:         Write x
4:     else if exp is constance c then
5:         Write "c"
6:     else if exp = fname(y⃗) then
7:         if fname is defined then
8:             Write "fname "
9:             for all arg ∈ y⃗ do
10:                ParsingTerm(arg)
11:            Write t → "
12:        else
13:            Write Corresponding built-in Coq function
14:            for all arg ∈ y⃗ do
15:                ParsingTerm(arg)
16: end function
```

online (`http://netdb.cis.upenn.edu/secure_routing/`.)

3.4.1. S-BGP

Secure Border Gateway Protocol (S-BGP) [75] is a comprehensive solution that aims to eliminate security vulnerabilities of BGP, while maintaining compatibility with original BGP specifications. S-BGP requires that each node sign the route information (route attestation) using asymmetric encryption (e.g. RSA [73]) before advertising the message

Algorithm 4 Generate Proof Obligation – Axioms

```
1: function GeneAxiom(sp, tp)
2:     (* sp: SANDlog program *)
3:     (* tp: Type annotation *)
4:     (* Invariant definition *)
5:     for all p(@ι, x⃗) ∈ HeadTuple(sp) do
6:         Write Axiom geneProp-p:forall
7:         Where (ι : id_type′) ∈ tp
8:         Write (attr_0 : type′)
9:         for i ← 1, len(x⃗) do
10:            Where (x⃗(i) : type′) ∈ tp
11:            Write (attr_i:type')
12:        Write (loc : id_type′)(t : time),
13:        Write pred-p
14:        for i ← 0, len(x⃗) do
15:            Write attr_i
16:        Write loc t →
17:        Write Honest loc →
18:        Write p-p
19:        for i ← 0, len(x⃗) do
20:            Write attr_i
21:        Write loc t.
22: end function
```

to its neighbor. The route information is supposed to include the destination address (represented by an IP prefix), the known path to the destination, and the identifier of the neighbor to whom the route information will be sent. The sender also attaches a signature list to the route information, containing all signatures received from the previous neighbors. A node receiving the route attestation would not trust the routing information unless all signatures inside are properly checked.

Encoding. Figure 14 presents our encoding of S-BGP in SANDlog. The meaning of tuples in the program can be found in Figure 15.

In rule r1 of Figure 14, when a node N receives an advertise tuple from its neighbor Nb, it generates a verifyPath tuple, which serves as an entry point for recursive signature verification. In rule 2, N recursively verifies all signatures in OsI, which stands for "original signature list". SI in verifyPath is a sub-list of OsI, representing the signatures that have not been

```
r1 verifyPath(@N,Nb,Pfx,Pvf,
              Sl,OrigP,Osl) :-
    advertise(@N,Nb,Pfx,RcvP,Sl),
    link(@N,Nb),
    Pvf := f_prepend(N,RcvP),
    OrigP := Pvf,
    Osl := Sl,
    f_member(RcvP,N) == 0,
    Nb == f_first(RcvP).
r2 verifyPath(@N,Nb,Pfx,PTemp,
               Sl1,OrigP,Osl) :-
    verifyPath(@N,Nb,Pfx,Pvf,
               Sl,OrigP,Osl),
    publicKeys(@N,Nd,PubK),
    f_size(Sl) > 0,
    f_size(Pvf) > 1,
    PTemp := f_removeFirst(Pvf),
    Nd := f_first(PTemp),
    SigM := f_first(Sl),
    MsgV := f_prepend(Pfx,Pvf),
    f_verify(MsgV,SigM,PubK) == 1,
    Sl1 := f_removeFirst(Sl).
r3 route(@N,Pfx,C,OrigP,Osl) :-
    verifyP(@N,Nd,Pfx,Pvf,
            Sl,OrigP,Osl),
    f_size(Sl) == 0,
    f_size(Pvf) == 1,
    C:= f_size(OrigP) - 1.
r4 route(@N,Pfx,C,P,Sl) :-
    prefixs(@N,Pfx),
    List := f_empty(),
    C := 0,
    P := f_prepend(N,List),
    Sl := f_empty().
r5 bestRoute(@N,Pfx,a_MIN<C>,P,Sl) :-
    route(@N,Pfx,C,P,Sl).
r6 signature(@N,Msg,Sig) :-
    bestRoute(@N,Pfx,C,BestP,Sl),
    privateKeys(@N,PriK),
    link(@N,Nb),
    Pts := f_prepend(Nb,BestP),
    Msg := f_prepend(Pfx,Pts),
    Sig := f_sign(Msg,PriK).
r7 advertise(@Nb,N,Pfx,BestP,NewSl) :-
    bestRoute(@N,Pfx,C,BestP,Sl),
    link(@N,Nb),
    Pts := f_prepend(Nb,BestP),
    Msg == f_prepend(Pfx,Pts),
    signature(@N,Msg,Sig),
    NewSl := f_prepend(Sig,Sl).
```

Figure 14: S-BGP encoding

checked. When all signatures have been verified — this is ensured by "$f\_size(Sl) == 0$" in rule 3 — N accepts the route and stores the path as a route tuple in the local database. Rule 4 also allows N to generate a route tuple storing the path to its self-owned IP prefixes (i.e. prefix(@N, Pfx)). Given a specific destination Pfx, in rule 5, N aggregates all route tuples storing paths to Pfx, and computes a bestPath tuple for the shortest path. The bestPath is intended to be propagated to downstream ASes. Before propagation, however, S-BGP requires N to sign the path information. This is captured in rule 6, where N uses its private key (i.e. privateKeys(@N, PriK)) to generate a signature based on the selected bestPath tuple. Finally, in rule 7, N embeds the routing information (i.e. bestPath) along with its signature (i.e. signature) into a new route advertisement (i.e. advertise), and propagates the message to its neighbors.

$\mathsf{link}(@n, n')$	there is a link between n and n'.
$\mathsf{route}(@n, d, c, p, sl)$	p is a path to d with cost c.
	sl is the signature list associated with p.
$\mathsf{prefix}(@n, d)$	n owns prefix (IP addresses) d.
$\mathsf{bestRoute}(@n, d, c, p, sl)$	p is the best path to d with cost c.
	sl is the signature list associated with p.
$\mathsf{verifyPath}(@n, n', d, p, sl,$	a path p to a destination d is verified against signature list sl.
$pOrig, sOrig)$	p is a sub-path of $pOrig$, and s is a sub-list of $sOrig$.
$\mathsf{signature}(@n, m, s)$	n creates a signature s of message m with private key.
$\mathsf{advertise}(@n', n, d, p, sl)$	n advertises path p to neighbor n' with signature list sl.

Figure 15: Tuples for $prog_{sbgp}$

Property specification. Route authenticity of S-BGP ensures that no route announcement can be tampered with by an attacker. In other words, it requires that any route announcement *accepted* by a node is authentic. We encode it as φ_{auth1} below.

$$\varphi_{auth1} = \forall n, m, t, d, p, sl, \\ \mathsf{Honest}(n) \wedge \mathsf{advertise}(m, n, d, p, sl)@(n, t) \supset \mathsf{goodPath}(t, d, p)$$

φ_{auth1} is a general topology-independent security property. It asserts that whenever an honest node n, denoted as $\mathsf{Honest}(n)$, sends out an advertise tuple to its neighbor m, the property $\mathsf{goodPath}(t, d, p)$ holds. $\mathsf{Honest}(n)$ means that n runs S-BGP and n's private key is not compromised. Formally:

$$\mathsf{Honest}(n) \triangleq \mathsf{honest}(n, prog_{sbgp}(n), -\infty).$$

Here, the starting time is set to be the earliest possible time point. SANDlog's semantics allows a node to begin execution at any time after the specified starting time, so using $-\infty$ gives us the most flexibility. $\mathsf{goodPath}(t, d, p)$ is recursively defined in Figure 16, which asserts that all links in the path p towards the destination d exist no later than t. Each link (m, n) is represented by two tuples: `link(@n, m)` and `link(@m, n)`. These two tuples reside on two endpoints respectively.

To be more specific, the definition of $\mathsf{goodPath}(t, d, p)$ involves three cases (Figure 16). The base case is when p contains only one node. We require that d be one of the prefixes owned

$$\frac{\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{prefix}(n, d)@(n, t')}{\mathsf{goodPath}(t, d, n :: nil)}$$

$$\frac{\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \qquad \mathsf{goodPath}(t, d, n :: nil)}{\mathsf{goodPath}(t, d, n' :: n :: nil)}$$

$$\frac{\begin{array}{l}\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \wedge \exists t'', t'' \leqslant t \wedge \mathsf{link}(n, n'')@(n, t'') \\ \mathsf{goodPath}(t, d, n :: n'' :: p'')\end{array}}{\mathsf{goodPath}(t, d, n' :: n :: n'' :: p'')}$$

Figure 16: Definitions of goodPath

by n (i.e., the prefix tuple is derivable). When p has two nodes n' and n, we require that the link from n to n' exist from n's perspective, assuming that n is honest, but impose no constraint on n''s database, because n' has not received the advertisement. The last case is when the length of p is larger than two; we check that both links (from n to n' and from n to n'') exist from n's perspective, assuming n is honest. In the last two rules, we also recursively check that the subpath also satisfies goodPath.

goodPath can serve as a template for a number of useful properties. For example, by substituting link (n,n'') with announce_link (n, n''), we are able to express whether a node is willing to let its neighbor know of that link. We can also require each subpath be authorized by the sender.

Axiom of signature. To use the authenticity property of signatures in the proof of φ_{auth1}, we include the following axiom A_{sig} in the logical context Γ. This axiom states that if a signature s is verified by the public key of a node n', and n' is honest, then n' must have generated a signature tuple. Predicate $\mathsf{verify}(m, s, k)@(n, t)$ means that node n verifies, using key k at time t, that s is a valid signature of message m.

$$A_{sig} = \forall m, s, k, n, n', t, \mathsf{verify}(m, s, k)@(n, t) \wedge \mathsf{publicKeys}(n, n', k)@(n, t) \wedge \mathsf{Honest}(n') \supset \exists t', t' < t \wedge \mathsf{signature}(n', m, s)@(n', t')$$

Verification. Our goal is to prove that φ_{auth1} is an invariant property that holds on all possible execution traces. However, directly proving φ_{auth1} is hard, as it involves verification over all the traces. Instead, we take the indirect approach of using our program logic to prove a program invariant, which is stronger than φ_{auth1}, and, more importantly, whose validity implies the validity of φ_{auth1}. To be concrete, we show that $prog_{sbgp}$ has the following invariant property φ_I:

$$\text{(a) } \cdot;\cdot \vdash prog_{sbgp}(i) : \{i, y_b, y_e\}.\varphi_I(i, y_b, y_e)$$

where φ_I is defined as:

$$\varphi_I(i, y_b, y_e) = \bigwedge_{p \in hdOf(prog_{sbgp})} \forall t\ \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i, t) \supset \varphi_p(i, t, \vec{x})$$

Every φ_p in φ_I denotes the invariant property associated with each head tuple in $prog_{sbgp}$, and needs to be specified by the user. Table 1 gives the invariants associated with all head tuples in the program. Especially, the invariant associated with the advertise tuple (goodPath) is the same as the conclusion of φ_{auth1}.

We prove (a) using the Inv rule in Section 3.2, by showing that all the premises hold. The Inv rule has two types of premises: (1) Premises that ensure each rule of the program maintains the invariant of its rule head; and (2) Premises that ensure all invariants for head tuples are closed under trace extension. Premises of the second type are guaranteed through manual inspection of all the invariants, thus omitted in the formal proof. In terms of premises of the first type, since $prog_{sbgp}$ has seven rules, this corresponds to seven premises to be proved. For example, the premise corresponding to rule 2 is represented by (a_0), shown below.

Rule	**Head Tuple**	**Invariant**
r1,r2	verifyPath (N,Nb,Pfx,Pvf, Sl,OrigP,Osl)@(N,t)	$\exists l, \mathsf{Osl} = l{+}{+}\mathsf{Pvf} \wedge$ (goodPath(t,Pfx,Pvf) $\supset$ goodPath(t,Pfx,Osl))
r3,r4	route (N,Pfx,C,OrigP,Osl)@(N,t)	goodPath (t,Pfx,OrigP)
r5	bestRoute (N,Pfx,C,P,Sl)@(N,t)	goodPath (t,Pfx,OrigP)
r6	signature (N,Msg,Sig)@(N,t)	$\exists$p, m, pfx, Msg = pfx :: nei :: p
r7	advertise (Nb,N,Pfx,BestP,NewSl)	goodPath (t,Pfx,Nb::BestP)

Table 1: Tuple invariants in φ_I for S-BGP route authenticity

(a_0) $\cdot;\cdot \vdash \forall \mathsf{N}, \forall \mathsf{Nb}, \forall \mathsf{Pfx}, \forall \mathsf{Pvf}, \forall \mathsf{Sl}, \forall \mathsf{Sl1}, \forall \mathsf{OrigP}, \forall \mathsf{Osl}, \forall \mathsf{t}, \forall \mathsf{Nd},$

$\forall \mathsf{PubK}, \forall \mathsf{m}, \forall \mathsf{p}, \forall \mathsf{SigM}, \forall \mathsf{MsgV}, \forall \mathsf{PTemp}, \forall \mathsf{Osl},$

verifyPath(N,Nb,Pfx,Pvf,SL,OrigP,Osl)@(N,t) $\wedge$

$\exists l, \mathsf{Osl} = l{+}{+}\mathsf{Sl} \wedge$

(goodPath(t,Pfx,Pvf) $\supset$ goodPath(t,Pfx,Osl)) $\wedge$

publicKeys(N,Nd,PubK)@(N,t) $\wedge$

length(Sl) $> 0 \wedge$

length(Pvf) $> 0 \wedge$

Pvf = m :: Nd :: p $\wedge$

PTemp = Nd :: p $\wedge$

Sl = SigM :: Sl1 $\wedge$

MsgV = Pfx :: Pvf $\wedge$

verify(MsgV,SigM,PubK)@(N,t) $\supset$

($\exists l, \mathsf{Osl} = l{+}{+}\mathsf{Sl1} \wedge$

(goodPath(t,Pfx,PTemp) $\supset$ goodPath(t,Pfx,Osl)))

Here, $(\exists l, \mathit{Osl} = l{+}{+}\mathit{Sl1} \wedge (\mathit{goodPath}(\mathit{t,Pfx,PTemp}) \supset \mathit{goodPath}(\mathit{t,Pfx,Osl})))$ is the invariant of rule 2's head tuple verifyPath (Figure 1). Other rule-related premises are constructed in a similar way. We prove all the premises in Coq, thus proving (a).

After (a) is proved, by applying the Honest rule to (a), we can deduce $\varphi = \forall n\ t, \mathsf{Honest}(n) \supset \varphi_I(n, t, -\infty)$. φ_I can then be injected into the assumptions (Γ) by VCGen (as do φ_{I1}) and is safe to be used as a theorem in proving other properties. Finally, φ_{auth1} is proved by

discharging $\varphi \supset \varphi_{auth1}$ in Coq with standard elimination rules.

Proof details. Among the others, the premise corresponding to rule 2 in the program turns out to be the most challenging one, as it involves recursion and signature verification. Recursion in rule 2 makes it hard to find the proper invariant specification for the head tuple verifyPath, as the invariant needs to maintain correctness for both the head tuple and the body tuple, which have different arguments. In our specification, we specify the invariant in a way that reversely verify the signature list by checking the signature for the longest path first. More concretely, we use an implication, stating that if the path to be verified satisfies the invariant goodPath, then the entire path satisfies the invariant goodPath (Table 1).

Another challenge in proving the invariant for verifyPath is to reason about the existence of *link* tuples at the previous nodes. We solve the problem in two steps: (1) we prove a stronger auxiliary program invariant (a_1), which asserts the existence of the local *link* tuple when a node signs the path information. (2) we then use the axiom A_{sig} to allow a node who verifies a signature to assure the existence of the *link* tuple at the remote node who signs the signature.

More concretely, (a_1) is defined as:

$$(\mathrm{a}_1)\ \cdot;\cdot \vdash prog_{sbgp}(i) : \{i, y_b, y_e\}.\varphi_{I1}$$

In (a_1), all head tuples p other than *signature* and *advertise* takes on the same invariant $\varphi_{link1}(p, n, d, t)$:

$$\begin{aligned}\varphi_{link1}(p, n, d, t) = \exists p',\ & \\ & p = n :: p' \wedge (p' = \mathsf{nil} \supset \mathsf{prefix}(n, d)@(n, t)) \wedge \\ & \forall p'', m', p' = m' :: p'' \supset \mathsf{link}(n, m')@(n, t)\end{aligned}$$

It states that node n is the first element in path p, and the *link* tuple from n to its neighbor in p exists in n's database.

For *signature* and *advertise*, we introduce another property:

$$\varphi_{link2}(p,n,d,n',t) = \mathsf{link}(n,n')@(n,t) \wedge \exists p',\ p = n :: p' \wedge (p' = \mathsf{nil} \supset \mathsf{prefix}(n,d)@(n,t)) \wedge \forall p'', m', p' = m' :: p'' \supset \mathsf{link}(n,m')@(n,t)$$

$\varphi_{link2}(p,n,d,n',t)$ extends $\varphi_{link1}(p,n,d,t)$ by including the receiving node n' as an argument, asserting that the link between n and n' also exists. And the invariants of *signature* and *advertise* are:

$$\varphi_{signature}(i,t,n,m,s) = \exists n', d, m = d :: n' :: p \wedge \varphi_{link2}(p,n,d,n',t)$$
$$\varphi_{advertise}(i,t,n',n,d,p,sl) = \varphi_{link2}(p,n,d,n',t)$$

We prove (a_1) using the Inv rule. Then, by applying Honest rule to (a_1) and only keeping the clause in φ_{I2} related to signature, we derive the following:

$$(\mathrm{a}_2)\ \cdot;\cdot \vdash \forall n, \forall t, \forall m, \mathsf{Honest}(n) \wedge \mathsf{signature}(n,m,s)@(n,t) \supset \exists n', d, pm = d :: n' :: p \wedge \varphi_{link2}(p,n,d,n',t)$$

(a_2) connects an honest node's signature to the existence of related link tuples at a previous node in the path p.

Next, we use (a_2) along with A_{sig} to prove (a_0). Applying A_{sig} to tuples publicKeys and verify in (a_0), we can get:

$$(\mathrm{a}_3)\ \cdot;\cdot \vdash \forall \mathsf{Nd}, \forall \mathsf{MsgV}, \forall \mathsf{SigM}, \forall \mathsf{t}, \mathsf{Honest}(\mathsf{Nd}) \supset \exists \mathsf{t'}, \mathsf{t'} < \mathsf{t} \wedge \mathsf{signature}(\mathsf{Nd},\ \mathsf{MsgV},\mathsf{SigM})@(\mathsf{Nd},\mathsf{t'})$$

We further apply (a_2) to (a_3) to obtain:

$$(\mathrm{a}_4)\ \cdot;\cdot \vdash \forall \mathsf{Nd}, \forall \mathsf{t}, \forall \mathsf{MsgV}, \exists n', d, p, \mathsf{Nd} = d :: n' :: p \wedge \varphi_{link2}(p,\mathsf{Nd},d,n',\mathsf{t})$$

Combining (a_4) and the assumptions in (a_0), we are able to prove the conclusion of (a_0). Other premises can be proved similarly. For non-recursive rules, the premises for them are

$$\frac{\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{prefix}(n, d)@(n, t')}{\mathsf{goodPath2}(t, d, n :: nil)}$$

$$\frac{\mathsf{Honest}(n) \supset \exists t', c, s, t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \wedge \mathsf{route}(n, d, c, n :: nil, sl)@(n, t') \quad \mathsf{goodPath2}(t, d, n :: nil)}{\mathsf{goodPath2}(t, d, n' :: n :: nil)}$$

$$\frac{\begin{array}{l}\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \wedge \\ \qquad \exists t'', c, s, t'' \leqslant t \wedge \mathsf{link}(n, n'')@(n, t'') \wedge \\ \qquad \mathsf{route}(n, d, c, n :: n'' :: p'', sl)@(n, t') \\ \mathsf{goodPath2}(t, d, n :: n'' :: p'')\end{array}}{\mathsf{goodPath2}(t, d, n' :: n :: n'' :: p'')}$$

Figure 17: Definitions of goodPath2

straightforward. The detailed proof can be found online.

Discussion. φ_{auth1} is a general template for proving different kinds of route authenticity. For example, S-BGP satisfies a stronger property that guarantees authentication of each subpath in a given path p. The property, called $\mathsf{goodPath2}(t, d, p)$, is defined in Figure 17. The meaning of the variables remains the same as before.

Compared with goodPath, the last two rules of goodPath2 additionally assert the existence of a route tuple. The predicate $\mathsf{route}(n, d, c, n :: p', sl)@(n, t')$ states that node n generates a route tuple for path $n :: p'$ at time t', and that sl is the signature list that authenticates the path $n :: p'$. This property ensures that an attacker cannot use n's route advertisement for another path p', which happens to share the two direct links of n. More specifically, given $p = n1 :: n :: n2 :: p1$ and $p' = n1 :: n :: n2 :: p2$, with $p1 \neq p2$, an attacker could not replace p with p' without being detected. However, a protocol that only requires a node n to sign the links to its direct neighbors would be vulnerable to such attack.

3.4.2. SCION

SCION [91] is a clean-slate design of Internet routing architecture that offers more flexible route selection and failure isolation along with route authenticity. Our case study focuses on the routing mechanism proposed by SCION. We only provide high-level explanation of SCION. Detailed encoding can be found under the following link (`http://netdb.cis.upenn.edu/secure_routing/`).

In SCION, Autonomous Domains (AD) — a concept similar to Autonomous Systems (AS) in BGP — are grouped into different Trust Domains (TD). Inside each Trust Domain, top-tier ISP's are selected as the TD core, which provide routing service inside and across the border of TD. Figure 18 presents an example deployment of SCION with two TD's. Each AD can communicate with its neighbors. The direction of direct edges represents provider-customer relationship in routing; the arrow goes from a provider to its customer.

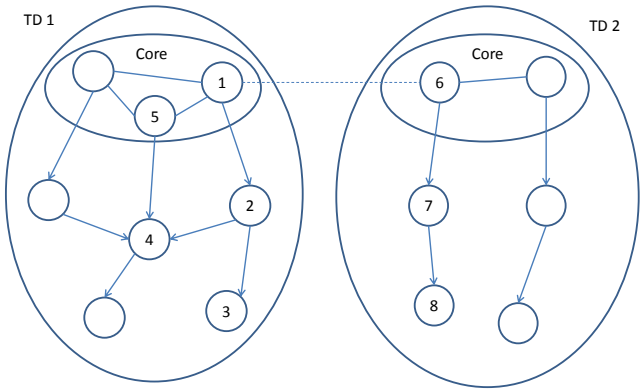


Figure 18: An example deployment of SCION

To initiate the routing process, a TD core periodically generates a path construction announcement, called a beacon, to all its customer ADs. Each non-core AD, upon receiving a beacon, (1) verifies the information inside the beacon, (2) attaches itself to the path inside the received beacon to construct a new beacon, and (3) forwards the new beacon to its customer ADs. Each beacon represents a path towards the TD core (e.g. path "1-2-3" in Figure 18). After receiving k beacons , an downstream AD selects m paths out of k and

uploads them to the TD core, thus finishing path construction (k and m can be set by the administrator). When later an AD n intends to send a packet to another AD n', it first queries the TD core for the paths that n' has uploaded, and then constructs a forwarding path combining its own path to the TD core with the query result. For example, in Figure 18, when node 4 wants to communicate with node 3, it would query from the TD core for path "1-2-3", and combine it with its own path to the TD core (i.e. "4-5"), to get the desired path "4-5-1-2-3".

In Table 2, we summarize the SANDlog encoding of the path construction phase in SCION. Definitions of important tuples can be found in Figure 19. The path construction beacon plays an important role in SCION routing mechanism. A beacon is composed of four fields: an interface field, a time field, an opaque field and a signature. The interface field in SCION is identical to the announced path in S-BGP. An interface field contains a list of AD identifiers representing the routing path. As its name suggests, the interface field also includes each AD's interfaces to direct neighbors in the path — SCION calls the interface to an AD's provider as *ingress* and the one to a customer as *egress*. Each AD attaches his own identifier along with its *ingress* and *egress* to the end of the received interface field, generating the new interface field. For example, in Figure 18, assume the ingress interface of AD 2 against AD1 is "a", and the egress interface of AD 2 against AD 4 is "b". Given an interface field {c::1} from AD 1 — c represents the egress interface of AD 1 against AD 2 — the newly generated interface field at AD 2 targeting AD 4 will be {c::1::a::b::2}.

The time field is a list of time stamps which record the arrival time of the beacon at each AD. The opaque field adds a message authentication code (MAC) on each AD's *ingress* and *egress* fields using the AD's private key, for the purpose of path authentication during data forwarding. The final part is called the signature list. Each AD constructs a signature by signing the above threes fields (i.e. the interface field, the opaque field and the time field) along with the signature received from preceding ADs. The newly generated signature is appended to the end of the signature list.

Rule	Summary	Head Tuple
b1:	TD core generates a signature.	signature(@*core*, *info*, *sig*, *time*)
b2:	TD core signs beacon global information.	signature(@*core*, *info*, *sig*, *time*)
b3:	TD core initiates an opaque field.	mac(@*core*, *info*, *hash*)
b4:	TD core initiates global info of beacon.	beaconPrep(@*core*, *glb*, *sigG*, *time*)
b5:	TD core sends a new beacon to neighbor.	beaconIni(@*nei*, *core*, *td*, *itf*, *tl*, *ol*, *sl*, *sigG*)
b6:	AD receives a beacon from TD core.	beaconRev(@*ad*, *td*, *td*, *itf*, *tl*, *ol*, *ing*, *sigG*)
b7:	AD receives a beacon from non-core AD.	beaconRev(@*ad*, *td*, *ing*, *itf*, *tl*, *ol*, *sl*, *sigG*)
b8:	AD verifies global information.	beaconToVeri(@*ad*, *td*, *itf*, *l*, *ol*, *sl*, *sigG*, *itfv*, *pos*)
b9:	AD recursively verifies signatures.	beaconToVeri(@*ad*, *td*, *itf*, *tl*, *ol*, *sl*, *sigG*, *itfv*, *pos*)
b10:	AD validates a beacon.	verifiedBeacon(@*ad*, *td*, *ing*, *itf*, *tl*, *ol*, *sl*, *sigG*)
b11:	AD creates signature for new beacon.	signature(@*ad*, *info*, *sig*, *time*)
b12:	AD initiates opaque field for new beacon.	mac(@*ad*, *info*, *hash*)
b13:	AD sends the new beacon to neighbor.	beaconFwd(@*nei*, *ad*, *td*, *itf*, *tl*, *ol*, *sl*, *sigG*)
pc1:	AD extracts path information.	upPath(@*ad*, *td*, *itf*, *ol*, *tl*)
pc2:	AD initiates path upload.	pathUpload(@*nei*, *ad*, *src*, *core*, *itf*, *ol*, *op*, *pos*)
pc3:	AD sends path to upstream neighbor.	pathUpload(@*nei*, *ad*, *src*, *core*, *itf*, *ol*, *op*, *pos*)
pc4:	TD core stores received path.	downPath(@*core*, *src*, *itf*, *op*)

Table 2: SANDlog encoding of path construction in SCION

coreTD($@n, c, td, ctf$)	c is the core of TD td with certificate ctf attesting to that fact
provider($@n, m, ig$)	m is n's provider, with traffic into n through interface ig
customer($@n, m, eg$)	m is n's customer, with traffic out of n through interface eg
beaconIni($@m, n, td,$ itf, tl, ol, sl, sg)	itf, containing a path, is initialized by n and sent to m. tl is a list of time stamps, ol is a list of opaque fields, whose meaning is not relevant here. sl is list of signatures for route attestation. sg is a signature for certain global information, which is not relevant here.
verifiedBeacon($@n, td,$ itf, tl, ol, sl, sg)	itf is the stored interface fields from n to the TD core in td. Rest of the fields have the same meaning as those in beaconIni
beaconFwd($@m, n, td,$ itf, tl, ol, sl, sg).	itf is forwarded to m with corresponding signature list sl Rest of the fields have the same meaning as those in beaconIni
upPath($@n, td, itf,$ $opqU, tl$)	$opqU$ is a list of opaque fields indicating a path. Rest of the fields have the same meaning as those in beaconIni.
pathUpload($@m, n,$ $src, c, itf, opqD,$ $opqU, pt$)	src is the node (AD) who initiated the path upload process. c is TD core of an implicit TD. $opqD$ is the opaque fields uploaded. pt indicates the next opaque field in $opqU$ to be checked. itf and $opqU$ have the same meaning as those in upPath.

Figure 19: Tuples for SCION

SCION also satisfies similar route authenticity properties as S-BGP. Each path in SCION is composed of two parts: a path from the sender to the TD core (called "up path") and a path from the TD core to the receiver (called "down path"). We only prove the properties for the up paths. The proof for the down paths can be obtained similarly by switching the role of provider and customer. Tuples provider and customer in SCION can be seen as counterparts of the link tuple in S-BGP, and tuple beaconIni and tuple beaconFwd correspond to tuple advertise. The definition of route authenticity in SCION, denoted φ_{authS}, is defined as:

$$
\begin{aligned}
\varphi_{authS} = \forall & n, m, t, td, itf, tl, ol, sl, sg, \\
& \mathsf{honest}(n) \wedge \\
& (\mathsf{beaconIni}(@m, n, td, itf, tl, ol, sl, sg)@(n, t) \vee \\
& \mathsf{beaconFwd}(@m, n, td, itf, tl, ol, sl, sg)@(n, t)) \\
& \supset \mathsf{goodInfo}(t, td, n, sl, itf)
\end{aligned}
$$

$$\frac{\mathsf{coreTD}(ad, c, td, ctf)@(ad, t) \quad \mathsf{Honest}(c) \supset \exists t', t' \leqslant t \wedge \mathsf{customer}(c, n, ceg)@(c, t')}{\mathsf{goodInfo}(t, td, ad, nil, c :: ceg :: n :: nil)}$$

$$\frac{\begin{array}{l}\mathsf{coreTD}(ad, c, td, ctf)@(ad, t) \\ \mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{provider}(n, c, nig)@(n, t') \wedge \mathsf{customer}(n, m, neg)@(n, t') \\ \qquad \wedge \exists td', tl, ol, sg, s, \mathsf{verifiedBeacon}(n, td', c :: ceg :: n :: nig :: nil, tl, \\ \qquad\qquad ol, s :: nil)@(n, t') \\ \mathsf{goodInfo}(t, td, ad, nil, (c :: ceg :: n :: nil))\end{array}}{\mathsf{goodInfo}(t, td, ad, s :: nil, c :: ceg :: n :: nig :: neg :: m :: nil)}$$

$$\frac{\begin{array}{l}\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{provider}(n, h, nig)@(n, t') \wedge \mathsf{customer}(n, m, meg)@(n, t') \\ \qquad \wedge \exists td', tl, ol, sg, s, sl, \mathsf{verifiedBeacon}(n, td', p' + +h :: hig :: heg :: n :: nig, \\ \qquad\qquad tl, ol, s :: sl)@(n, t'). \\ \mathsf{goodInfo}(t, td, ad, sl, p' + +h :: hig :: heg :: n :: nil)\end{array}}{\mathsf{goodInfo}(t, td, n, s :: sl, \quad p' + +h :: hig :: heg :: n :: nig :: neg :: m :: nil)}$$

Figure 20: Definitions of goodInfo

Formula φ_{authS} asserts a property $\mathsf{goodInfo}(t, td, n, sl, itf)$ on any beacon tuple generated by node n, which is either a TD core or an ordinary AD. The definition of goodInfo is shown in Figure 20. Predicate $\mathsf{goodInfo}(t, td, n, sl, itf)$ takes five arguments: t represents the time, td is the identity of the TD that the path lies in, n is the node that verifies the beacon containing the interface field itf, and sl is the signature list associated with the path. $\mathsf{goodInfo}(t, td, n, sl, itf)$ makes sure that each AD present in the interface field itf does have the specified links to its provider and customer respectively. Also, for each non-core AD, there always exists a verified beacon corresponding to the path from the TD core to it.

More concretely, the definition of goodInfo considers three cases. The base case is when a TD core c initializes an interface field $c :: ceg :: n :: nig :: nil$ and sends it to AD n. We require that c be a TD core and n be its customer. The next two cases are similar, they both require the current AD n have a link to its preceding neighbor, represented by provider, as well as one to its downstream neighbor, represented by customer. In addition, a verifiedBeacon tuple should exist, representing an authenticated route stored in the database,

with all inside signatures properly verified. The difference between these two cases is caused by two possible types of an AD's provider: TD core and non-TD core.

The proof strategy is exactly the same as that used in proof of goodPath about S-BGP. To prove φ_{authS}, we first prove $prog_{scion}$ has a stronger program invariant φ_I:

$$\text{(b) } \cdot;\cdot \vdash prog_{sci}(n) : \{i, y_b, y_e\}.\varphi_I(i, y_b, y_e)$$

where $\varphi_I(i, y_b, y_e)$ is defined as:

$$\varphi_I(i, y_b, y_e) = \bigwedge_{p \in hdOf(sci)} \forall t, \forall \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i, t) \supset \varphi_p(i, t, \vec{x}).$$

Especially, φ_p for beaconIni and beaconFwd are as follows:

$$\varphi_{beaconIni}(i, t, m, n, td, itf, tl, ol, sl, sg) = \mathsf{goodInfo}(t, td, n, sl, itf)$$
$$\varphi_{beaconFwd}(i, t, m, n, td, itf, tl, ol, sl, sg) = \mathsf{goodInfo}(t, td, n, sl, itf)$$

As in S-BGP, (b) can be proved using Inv rule, whose premises are verified in Coq. After (b) is proved, we can deduce $\varphi' = \forall n, t', \mathsf{Honest}(n) \supset \varphi_I(n, t', -\infty)$ by applying Honest rule to (b). Finally, φ_{authS} is proved by showing that $\varphi' \supset \varphi_{authS}$, which is straightforward.

At the end of the path construction phase, an AD needs to upload its selected paths to the TD core for future queries (i.e. rules $pc1 - pc4$ in Table 2). The uploading process uses the forwarding mechanism in SCION, which provides hop-by-hop authentication. An AD who wants to send traffic to another AD attaches each data packet with the opaque field extracted from a beacon received during the path construction phase. The opaque field contains MACs of the ingress and egress of all ADs on the intended path. When the data packet is sent along the path, each AD en-route re-computes the MAC of intended ingress and egress using its own private key. This MAC is compared with the one contained in the opaque field. If they are the same, the AD knows that it has agreed to receiving/sending packets from/to its neighbors during path construction phase and forwards the packet further along the path. Otherwise, it drops the data packet.

The formal definition of data path authenticity in SCION is defined as:

$$\begin{aligned}\varphi_{authD} = \forall m, n, t, src, core, itf, opqD, opqU, pt,\\ \mathsf{honest}(n) \wedge\\ \mathsf{pathUpload}(@m, n, src, core, itf, opqD, opqU, pt)@(n, t) \supset\\ \mathsf{goodFwdPath}(t, n, opqU, pt)\end{aligned}$$

Formula φ_{authD} asserts property $\mathsf{goodFwdPath}(t, n, opqU, pt)$ on any tuple pathUpload sent by a customer AD to its provider. $\mathsf{goodFwdPath}(t, n, opqU, pt)$ has four arguments: t is the time. n is the node who sent out pathUpload tuple. $opqU$ is a list of opaque fields for forwarding. pt is a pointer to $opqU$, indicating the next opaque field to be checked. Except time t, all arguments in $\mathsf{goodFwdPath}(t, n, opqU, pt)$ are the same as those in pathUpload, as described in Figure 19. $\mathsf{goodFwdPath}(t, n, opqU, pt)$ states that whenever an AD receives a packet, it has direct links to its provider and customer as indicated by the opaque field in the packet. In addition, it must have verified a beacon with a path containing this neighboring relationship.

The definition of $\mathsf{goodFwdPath}(t, n, opqU, pt)$ is given in Figure 21. There are four cases. The base case is when pt is 0, which means nothing has been verified. In this case goodFwdPath holds trivially. If pt is equal to the length of opaque field list, meaning all opaque fields have been verified already, then based on SCION specification, the last opaque field should be that of the TD core. Being a TD core requires a certificate (coreTD), and a neighbor customer along the path (customer). When pt does not point to the head or the tail of the opaque field list, node n should have a neighbor provider(provider), and a neighbor customer(customer). It must also have received and processed a verifiedBeacon during path construction. The second and third cases both cover this scenario. The second case applies when a node n's provider is TD core, while in the third case, n's provider and customer are both ordinary TDs.

SCION uses MAC for integrity check during data forwarding, so we use the following axiom about MAC. It states that if a node n verifies a MAC, using n''s key k, there must have

$$\frac{pt = 0}{\mathsf{goodFwdPath}(t, n, opqU, pt)}$$

$$\frac{\begin{array}{r}\mathsf{Honest}(n) \supset \exists t', m, td, ctf, t' \leqslant t \\ \wedge\ \mathsf{coreTD}(n, n, td, ctf)@(n, t') \\ \wedge\ \mathsf{customer}(n, m, ceg)@(n, t')\end{array}}{\begin{array}{r}\mathsf{goodFwdPath}(t, n, opq' {+\!+} [ceg :: mac :: nil] :: nil, \\ length(opq' {+\!+} [ceg :: mac :: nil] :: nil))\end{array}}$$

$$\frac{\begin{array}{l}0 < pt \wedge pt < length(opq' {+\!+} [nig :: neg :: mac' :: nil] :: [ceg :: mac :: nil] :: nil \wedge \\ \mathsf{Honest}(n) \supset \exists t', h, m, t' \leqslant t \\ \qquad \wedge\ \mathsf{provider}(n, h, nig)@(n, t') \\ \qquad \wedge\ \mathsf{customer}(n, m, meg)@(n, t') \\ \qquad \wedge\ \exists td', tl, sg, sl, \mathsf{verifiedBeacon}(n, td', h :: ceg :: n :: nig, \\ \qquad\qquad tl, [ceg :: mac :: nil] :: nil, sl, sg)@(n, t')\end{array}}{\mathsf{goodFwdPath}(t, n, opq' {+\!+} [nig :: neg :: mac' :: nil] :: [ceg :: mac :: nil] :: nil, pt)}$$

$$\frac{\begin{array}{l}0 < pt \wedge \\ pt < length(opq' {+\!+} [nig :: neg :: mac' :: nil] :: [hig :: heg :: mac :: nil] {+\!+} opq'') \wedge \\ \mathsf{Honest}(n) \supset \exists t', h, m, t' \leqslant t \\ \qquad \wedge\ \mathsf{provider}(n, h, nig)@(n, t') \\ \qquad \wedge\ \mathsf{customer}(n, m, meg)@(n, t') \\ \qquad \wedge\ \exists td', tl, sg, sl, p', p'', \mathsf{verifiedBeacon}(n, td', p' {+\!+} h :: hig :: heg :: n :: nig, \\ \qquad\qquad tl, p'' {+\!+} [hig :: heg :: mac :: nil] :: nil, sl, sg)@(n, t')\end{array}}{\mathsf{goodFwdPath}(t, n, opq' {+\!+} [nig :: neg :: mac' :: nil] :: [hig :: heg :: mac :: nil] {+\!+} opq'', pt)}$$

Figure 21: Definitions of goodFwdPath

been a node n'' who created the MAC at an earlier time t'.

$$\begin{array}{l}A_{mac} = \forall msg, m, k, n, n', t, \\ \qquad \mathsf{verifyMac}(msg, m, k)@(n, t) \wedge \mathsf{privateKeys}(n, n', k)@(n, t) \wedge \\ \qquad \exists n'', \mathsf{Honest}(n'') \supset \exists t', t' < t \wedge \mathsf{privateKeys}(n'', n', k)@(n'', t) \wedge \\ \qquad\qquad \mathsf{mac}(n'', msg, m)@(n'', t')\end{array}$$

In SCION, each node should not share its own private key with other nodes. This means, for each specific MAC, only the node who generated it can verify its validity. This fact simplifies the axiom:

$$\begin{array}{l}A'_{mac} = \forall msg, m, k, n, t, \\ \qquad \mathsf{verifyMac}(msg, m, k)@(n, t) \wedge \mathsf{privateKeys}(n, k)@(n, t) \wedge \\ \qquad \mathsf{Honest}(n) \supset \exists t', t' < t \wedge \mathsf{mac}(n, msg, m)@(n, t')\end{array}$$

The rest of the proof follows the same strategy as that of goodPath and goodInfo. Interested readers can refer to our proof online for details.

3.4.3. Comparison between S-BGP and SCION

In this section, we compare the difference between the security guarantees provided by S-BGP and SCION. In terms of practical route authenticity, there is little difference between what S-BGP and SCION can offer. This is not surprising, as the kind of information that S-BGP and SCION sign at path construction phase is very similar. Though both use layered-signature to protect the routing information, signatures in S-BGP are not technically layered—ASes in S-BGP only sign the path information, not including previous signatures. On the other hand, ADs in SCION sign the previous signature so signatures in SCION are nested. Consider an AS n in S-BGP that signed the path p twice, generating two signatures: s and s'. An attacker, upon receiving a sequence of signatures containing s, can replace s with s' without being detected. This attack is not possible in SCION, as attackers cannot extract signatures from a nested signature.

SCION also provides stronger security guarantees than S-BGP in data forwarding. Though S-BGP does not explicitly state the process of data forwarding, we can still compare its IP-based forwarding to SCION's forwarding mechanism. Like BGP, an AS running S-BGP maintains a routing table on all BGP speaker routers that connect to peers in other domains. The routing table is an ordered collection of forwarding entries, each represented as a pair of $\langle$IP prefix, next hop$\rangle$. Upon receiving a packet, the speaker searches its routing table for IP prefix that matches the destination IP address in the IP header of the packet, and forwards the packet on the port corresponding to the next hop based on table look-up. This next hop must have been authenticated, because only after an S-BGP update message has been properly verified will the AS insert the next hop into the forwarding table.

However, SCION provides stronger security guarantee over S-BGP regarding the last hop of the packet. An AS n running S-BGP has no way of detecting whether a received packet

is from legitimate neighbor ASes who are authorized to forward packets to n. Imagine that n has two neighbor ASes, m and m'. n knows a route to an IP prefix p and is only willing to advertise the route to m. Ideally, any packet from m' through n to p should be rejected by n. However, this may not happen in practice for AS's who run S-BGP for routing. As long as its IP destination is p, a packet will be forwarded by n, regardless of whether it is from m or m'. On the other hand, SCION routers would discard such packets by verifying the MAC in the opaque field, since m cannot forge the MAC embedded in the beacon.

CHAPTER 4

Automated Verification and Debugging with Symbolic Execution

The program logic introduced in Chapter 3 is powerful, as its specification language –i.e., first-order logic – is expressive enough to specify most common properties in distributed systems. However, to verify properties specified in our logic, a tedious and laber-intensive manual proof is required. It is also beyond the ability of a system manager to use proof assistants efficiently. What is worse is that when the proofs cannot be constructed, it is nontrivial to find out what went wrong. Either there are bugs in the program, or the invariants used in the proofs are not correct. There is little tool support for identifying problems under these circumstances. Therefore, in this part of the dissertation, we aims at developing a static analysis-based technique to analyze the safety properties of NDlog programs automatically. When properties do not hold, our tool provides a concrete counterexample to further aid program debugging. The properties that we are interested in include invariants of the network and desirable behavior of nodes in the network. For instance, we would like to know if every forward entry corresponds to a route announcement packet, or if a successfully delivered packet indicates proper forwarding table setup in the switches that the packet traverses. One observation we have is that a large fragment of the interesting properties of networks can be expressed in a simple fragment of first-order logic. Leveraging this limited expressive power, we are able to develop static analysis for NDlog programs.

Our static analysis examines the structure of the NDlog program and builds a summary data structure for all derivations of that program. Properties specified in the restricted format of first-order logic are checked on the summary data structure with the help of the SMT solver Z3 [88]. The challenge is how to deal with recursive programs. For such programs, the number of possible derivations for recursive predicates is infinite. We use

a concise representation for recursive predicates, so all possible derivations can be finitely represented. To evaluate our analysis, we built a prototype tool, and verified several safety properties of a number of SDN controller programs, where the SDN's controller program and switch logic are specified in NDlog.

The proposed static analysis makes the following technical contributions.

- We developed algorithms for automatically analyzing a class of safety properties of NDlog programs.
- We proved the soundness and completeness of our algorithms for non-recursive programs, and the soundness of our algorithms for recursive programs.
- We implemented a prototype tool and verified a number of safety properties of SDN controller programs.

As our driving example, we will use the *erroneous* program in Figure 22. The non-recursive set of rules in the program computes one-, two-, and three-hop reachability information within a network. There is an error in rule `r2`, where `onehop X Z C2` should be `onehop Z Y C2`, thus this program cannot derive three-hop paths.

```
ThreeHops (With a deliberate error in r2):
r1 onehop(@X,Y,C) :- link(@X,Y,C).
r2 twohops(@X,Z,C) :- link(@X,Z,C1),
                      onehop(@X,Z,C2),C = C1+C2.
r3 threehops(@X,Y,C) :- onehop(@X,Z,C1),
                        twohops(@Z,Y,C2),C=C1+C2.
r4 threehops(@X,Y,C) :- twohops(@X,Z,C1),
                        onehop(@Z,Y,C2),C=C1+C2.
```

Figure 22: An erroneous NDlog program for demonstration purpose. The rule `r2` is wrong as it uses `onehop X Z C2` as its body relation, which should be `onehop Z Y C2`

4.1. Overview

We first present an overview of our solution. The static analysis mainly consists of two processes: a process that summarizes all derivations of predicates in an auxiliary data structure, which we call a *derivation pool*, and a process that queries properties on the

derivation pool. NDlog programs are represented abstractly as dependency graphs. Recursive programs are more complicated than non-recursive programs, so we explain the algorithms for non-recursive programs first, before we discuss extensions to support recursive programs. The dependency graph and the properties to be checked are of the same form for both recursive and non-recursive programs. Next, we formally define the dependency graph and the format of the properties.

Dependency graph A dependency graph can be formally defined as follows:

$$
\begin{array}{lll}
\textit{Predicate type} & \tau & ::= \mathsf{Pred} \mid \mathsf{bt} \supset \tau \\
\textit{Dependency graph} & \mathcal{G} & ::= (Np\ \mathsf{List}, Nr\ \mathsf{List}, E\ \mathsf{List}) \\
\textit{Predicate node} & Np & ::= (nID, p{:}\tau, \mathsf{c}cyc) \mid (nID, p{:}\tau, \mathsf{nc}yc) \\
\textit{Rule node} & Nr & ::= (rID, hd, body, c) \\
\textit{Edge} & E & ::= (rID, nID) \mid (nID, rID) \\
\textit{Rule head} & hd & ::= p(\vec{x}) \\
\textit{Rule body} & body & ::= p_1(\vec{x_1}), \cdots, p_n(\vec{x_n}) \\
\textit{Rule constraints} & c & ::= e_1\ bop\ e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \exists x.c
\end{array}
$$

A dependency graph has two types of nodes, predicate nodes, denoted Np, and rule nodes, denoted Nr. Each predicate node corresponds to a tuple in the program. A predicate node consists of a unique ID for the node, the name of the predicate and its type, and a tag indicating whether the predicate is on a cycle in the graph. The tag $\mathsf{c}cyc$ means that the node is on a cycle and $\mathsf{nc}yc$ means the opposite. Each rule node corresponds to a rule in the program. A rule node consists of a unique ID, the head of the rule, the body of the rule, which is a list of predicates, and the constraints. The edges, denoted E, are directional. Each edge points either from a rule node to the predicate node which is the head of that rule node, or from a predicate node to a rule node where the predicate is in the rule body.

To make variable substitution easier, each predicate takes unique variables as arguments.

For instance, the following two NDlog rules are equivalent, but `r1` is the normal form.

```
r1: p(x,y) :- q(x1), s(y1), x1=y1, x=x1, y=y1.
r2: p(x,y) :- q(x), s(y), x=y.
```

The dependency graph for ThreeHops is shown in Figure 23, where boxes represent nodes in the graph and arrows represent edges in the graph.

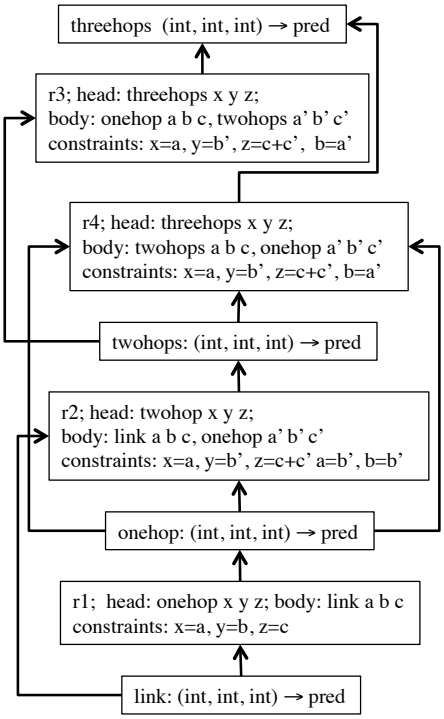


Figure 23: A dependency graph for the ThreeHops program(buggy)

Properties We focus on safety properties, which state that bad things have not happened yet. We use trace-based semantics of NDlog [63, 19]. The advantage of trace-based semantics over fixed point semantics is that the order in which predicates are derived can be clearly specified using traces. Fixed point semantics only care about what is derivable in the end, and are not precise enough to capture transient faults that appear only in the middle of the execution of network protocols.

To allow for automated analysis, we restrict the form of properties to be the following:
$\varphi = \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1} \cdots \vec{x_n}) \supset$

$$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1} \cdots \vec{x_n}, \vec{y_1} \cdots \vec{y_m})$$

The meaning of the property is the following: if all of the predicates p_i are derivable, and their arguments satisfy constraint c_p, then each of the predicate q_j must be in one of the derivations of p_i, and the constraint c_q must be true. We implicitly require q_is to be derived before p_is. A lot of the correctness properties can be specified using formulas of this form. For instance, we can specify the following three properties of our ThreeHops program:

Q1: $\forall x, y, z, \mathsf{threehops}\ x\ y\ z \supset \exists x', z', \mathsf{twohops}\ x\ x'\ z'$

Q2: $\forall x, y, z, \mathsf{threehops}\ x\ y\ z$

$\supset \exists x_1, x_2, z_1, z_2, z_3, \mathsf{link}\ x\ x_1\ z_1 \wedge \mathsf{link}\ x_1\ x_2\ z_2$

$\wedge\ \mathsf{link}\ x_2\ y\ z_3$

Q3: $\exists x, y, z, \mathsf{threehops}\ x\ y\ z$

Q1 states that to derive threehops $x\ y\ z$, it is necessary to derive twohops $x\ x'\ z'$, for some x' and z'. Q1 does not hold because there are two ways to derive threehops and one of them does not contain such a twohops tuple as a sub-derivation. Q2 states that to derive a threehops tuple, three links connecting those two nodes are necessary. Q2 should hold. Q3 states that threehops tuple is derivable for some x, y, and z.

4.2. Analyzing Non-recursive Programs

In this section, we first explain how to compute the derivation pool for a non-recursive NDlog program. Then, we show how to check properties. Next, we show how to incorporate network constraints into our property checking algorithm. Finally, we prove the correctness of our algorithm and analyze its time complexity.

4.2.1. Derivation Pool Construction

For a non-recursive program, its derivation pool maps each predicate to the set of all derivation trees rooted at that predicate. It is formally defined as follows.

$$
\begin{array}{lllll}
\textit{Derivation pool} & \textit{dpool} & ::= & \cdot \mid \textit{dpool}, (nID, p{:}\tau) \mapsto \Delta \\
\textit{Entries} & \Delta & ::= & \cdot \mid \Delta, (c, \mathcal{D}) \\
\textit{Derivation} & \mathcal{D} & ::= & (\mathsf{B}T, p(\vec{x})) \mid (rID, p(\vec{x}), \mathcal{D}\ \mathsf{List})
\end{array}
$$

We write *dpool* to denote derivation pools. We write Δ to denote lists of pairs of a constraint and a derivation tree, denoted $\mathcal{D}$. At a high-level, $\mathcal{D}$ can be instantiated to be a valid derivation of $p(\vec{t})$ using rules in the program, if c is satisfiable. A derivation tree, $\mathcal{D}$, is inductively defined. The base tuples, denoted $(\mathsf{B}T, p(\vec{x}))$, are the leaf nodes. A non-leaf node consists of the unique rule ID of the last rule of the derivation, the conclusion of that rule ($p(\vec{x})$), and the list of derivation trees for the body predicates of that rule ($\mathcal{D}$ List). We write *dpool*(p) to denote *dpool*($nID, p{:}\tau$), which returns Δ.

Figure 24 and 25 present the main functions used for constructing a derivation pool from a dependency graph. The top-level function GenDPool is defined in Figure 24. This function follows the topological order of the nodes in the dependency graph $\mathcal{G}$. We keep track of a working set P, which is the set of nodes whose derivations can be summarized currently. We also keep track of the set of edges that the function has not traversed yet. The function terminates when all of the edges in the dependency graph have been traversed and the derivations for all of the predicates in the dependency graph are built. In the body of GenDPool, we remove one predicate node p from P, and build all derivations for it. A base tuple's only possible derivation is one with itself as the leaf node. The constraint associated with this derivation is the trivial true constraint $\top$ (Line 8). When p is not a base tuple, derivations for tuples that p's derivations depend on have been stored in *dpool*. The GenDs function constructs derivations for p given the dependency graph and the current derivation pool (explained later).

```
1: function GenDPool(G)
2:     E ← G's edges
3:     P ← G's predicate nodes that have no incoming edges
4:     while E ≠ empty || P ≠ empty do
5:         remove (nID, p:τ) from P
6:         x⃗ ← fresh(p:τ)
7:         if p is a base tuple then
8:             dpool← dpool[(nID, p) ↦ {(⊤, (BT, p(x⃗)))}]
9:         else
10:            d ← GenDs(G, dpool, (nID, p:τ))
11:            dpool← dpool ∪ d
12:        (* done processing p, remove edges *)
13:        P, E ←removeEdges(P, E, G, nID)
14:    end while
15: end function
16:
17: function removeEdges(P, E, G, nID)
18:    remove outgoing edges of nID from E
19:    for each rID with no edges of form (_, rID) in E do
20:        remove edges (rID, nID) from E
21:        for each (nID, p:τ) with no incoming edges in E do
22:            add (nID, p:τ) to P
23: end function
```

Figure 24: Construct derivation pools for non-recursive programs

After the derivations for a predicate p are constructed, outgoing edges from p are removed (Line 13), so predicates that depend on p can be processed in later iterations. Function RemoveEdges removes outgoing edges from p, and outgoing edges from rule nodes that now do not have incoming edges. This may result in predicates enqueued into P for the next iteration of processing.

Function GenDs (Figure 25) takes the dependency graph, the derivation pool that has been constructed so far, and a predicate p, as arguments, and returns all derivation pool entries for p. The body of GenDs calls GenDRule to construct derivations for each rule that derives p. The function GenDRule makes use of List map and fold operations to construct all possible derivations of p from a rule of the form $r:\ p(\vec{x})\texttt{:-}q_1(\vec{y_1}), ..., q_n(\vec{y_n}), c$. *dpool* has already stored all possible derivations for each q_i. We need to compute all combinations of

```
1: function GenDs(G, dpool, (nID, p:τ))
2:     Δ ← {}
3:     for each rule with ID rID where (rID, nID) in G do
4:         Δ ← Δ∪GenDRule(G, dpool, (nID, p:τ),rID)
       return Δ
5: end function
6:
7: function GenDRule(G, dpool, (nID, p:τ), rID)
8:     (p(y⃗), Q, c_r) ← G(rID)
9:     D ← List.map (LookUp dpool) Q
10:    D′ ←List.FoldRight MergeDLL D nil
11:    x⃗ ← fresh(p(y⃗))
12:    return List.Map (completeD c_r rID p(y⃗) x⃗) D′
13: end function
14:
15: function MergeD(dc_i, dc_2i)
16:    (σ_2i, c_2i, d_2i) ← dc_2i
17:    (σ_i, c_i, d_i) ← dc_i
18:    (σ′_i, c′_i, d′_i) ← fresh(c_i, d_i)
19:    return (σ_i σ′_i ∪ σ_2i, c′_i ∧ c_2i, d′_i::d_2i)
20: end function
21:
22: function LookUp(dpool, q(x⃗))
23:    return List.Map (extractD x⃗) dpool(q)
24: end function
25:
26: function extractD(x⃗, (c, d))
27:    (rID, p(y⃗), dl) ← d
28:    return (y⃗/x⃗, c, d)
29: end function
30:
31: function completeD(c_r, rID, p(y⃗), x⃗, d)
32:    (σ, c, dl) ← d
33:    return (c ∧ c_r[x⃗/y⃗]σ, (rID, p(x⃗), dl))
34: end function
```

Figure 25: Generate derivation pool for one predicate

the derivations for q_is. The LookUp function on line 11 collects the list of derivations for one body tuple and the list map function returns the list of derivations for all body tuples. More precisely, the LookUp function returns a list of tuples of the form (σ, c, d), where d is a derivation, c is the constraint associated with that derivation, and σ is a variable substitution. The domain of σ is q_i's arguments in the rule node, and the range of σ is

q_i's arguments in the conclusion of the derivations. We need these substitutions because we alpha-rename the derivations. The constraint in the rule node needs to use the correct variables. Line 12 uses list fold operation to generate all possible derivations. Function MergeDLL and MergeDL are helper functions to generate the list of derivations. Function MergeD is the function that takes as arguments, the list of derivations from q_m to q_{i+1} and one derivation for q_i, and prepends the derivation for q_i to the list of derivations from q_m up to q_i. Here, the substitutions need to be merged and the resulting constraint is the conjunction of the two constraints. Finally on line 14, function Completed generates a well-formed derivation for p using the rule ID and the list of derivations for q_is. The constraint associated with this derivation of p is the conjunction of constraints for the derivation of q_i and the constraint in the rule body. The substitutions are applied to the constraint c, because all derivations are alpha-renamed and use fresh variables.

Example Constraint Pool A simplified derivation pool for onehop, twohops, and threehops is shown below. To ease presentation, we rewrite the derivation pool using equality constraints. onehop has only one derivation, using rule R1. A derivation $\mathcal{D}$ is a tuple consisting of four fields: the name of the last rule in the derivation; the conclusion of the derivation; the constraint associated with this derivation; and the list of derivations of the premises of the last rule. We instantiate the rules with concrete variables. The constraint in $\mathcal{D}$ is true, denoted $\top$; as there is no constraint in R1. The predicate twohops also has only one derivation, using R2. The premises of R2 are link and onehop. Since link is a base tuple, we simply represent its derivation as the tuple itself. The sub-derivation of onehop is the same as in the previous case. The constraint for deriving onehop is the conjunction of three constraints: c_1 is the constraint for deriving onehop, c_2 for the base tuple link, and c_3 the rule constraint of rule R2. Here c_2 is true, because no constraint is imposed on base tuples.

onehop

$\mathcal{D}$: $(\text{R1}, \text{onehop}\ x_1\ x_2\ x_3, \{\text{link}\ x_1\ x_2\ x_3\})$

$c = \top$

twohops

$\mathcal{D}$: ($\text{R}2, \text{twohops}\ x_1\ x_2\ x_3$

$\{\text{link}\ x_1\ x_2\ y_3, (\text{R}1, \text{onehop}\ x_1\ x_2\ z_3, \{\text{link}\ x_1\ x_2\ z_3\}\})$

$c = \top \wedge \top \wedge x_3 = y_3 + y_3$

threehops

$\mathcal{D}_1$: ($\text{R}3, \text{threehops}\ x_1\ x_2\ x_3$,

$\{(\text{R}1, \text{onehop}\ x_1, y_2, y_3, \{\text{link}\ x_1\ y_2\ y_3\})$

($\text{R}2, \text{twohops}\ y_2\ x_2\ s_3$,

$\{\text{link}\ y_2\ x_2\ t_3, (r1, \text{onehop}\ y_2\ x_2\ u_3, \{\text{link}\ y_2\ x_2\ u_3\})\})\})$

$c = \top \wedge \top \wedge \top \wedge s_3 = t_3 + u_3 \wedge x_3 = y_3 + s_3$

$\mathcal{D}_2$: ($\text{R}4, \text{threehops}\ x_1\ x_2\ x_3$,

$\{(\text{R}2, \text{twohops}\ x_1\ y_1\ s_3$,

$\{\text{link}\ x_1\ y_1\ t_3, (\text{R}1, \text{onehop}\ x_1\ y_1\ u_3, \{\text{link}\ x_1\ y_1\ u_3\}\})$

($\text{R}1, \text{onehop}\ y_1, x_2, y_3, \{\text{link}\ y_1\ x_2\ y_3\}$)})

$c = \top \wedge \top \wedge \top \wedge s_3 = t_3 + u_3 \wedge c_5 = x_3 = y_3 + s_3$

Tuple threehops has two derivations, one uses R3, the other uses R4. Both derivations contain sub-derivations of onehop and twohops. The constraints for deriving threehops include constraint for deriving twohops, onehop, and the rule constraint of R3 (R4).

4.2.2. Property Query

Figure 26 shows the property query algorithm for non-recursive programs. The top-level function CkProp takes the derivation pool and the property as arguments. On line 3, we separate the property into the list of predicates to the left of the implication (P), the constraint to the left of the implication (c_p), the list of predicates to the right of the implication (Q), and the constraint to the right of the implication (c_q). Next, similar to the derivation pool construction, we construct all possible combinations of the derivations of all the p_is in P between lines 5 to 9. We omit the definition of MergeDerivation, as it

```
1:  function CkProp(dpool, φ)
2:      (P, c_p, Q, c_q) ← φ
3:      L ← LookUpRec(dpool, P)
4:      D ←MergeDerivation L
5:      for each (σ, c_d, d) in D do
6:          z ←CkPropD(c_d, c_pσ, d, Q, c_qσ)
7:          if z = invalid(d, σ_r) then
8:              return invalid(d, σ_r)
9:      return valid
10: end function
11:
12: function CkPropD(c_d, c_p, d, Q, c_q)
13:     if Check sat c_d ∧ c_p = (sat, σ_p) then
14:         (* find all occurrences of q in d *)
15:         Σ ←List.map (Unify d) Q
16:         if nil ∈ Σ then
17:             (* some q_i does not appear in d *)
18:             return invalid(d, σ_p)
19:         else
20:             Σ_q ←MergeLL Σ
21:             c'_q ←Conj(Σ_q, ¬c_q)
22:             c_a ← c_p ∧ c_d ∧ c'_q
23:             if Check sat c_a = (sat, σ_a) then
24:                 return invalid(d, σ_a)
25:             else
26:                 return valid
27:     else
28:         (* Constraints for p_1 ... p_n and c_p are unsat *)
29:         return valid
30: end function
```

Figure 26: Property query

is similar to MergeDLL. The only difference is that we do not need to alpha-rename the derivations. Next, we check that for each possible derivation of p_is in D, all of q_is appear in the derivation, and the constraint c_q holds (lines 10 to 14) using function CkPropD. If for all possible derivations of p_is, we can always find derivations of q_is such that the constraint c_q holds, φ holds (line 14).

The function CkPropD checks that in the list of derivations d, with constraints c_d, whether all the predicates in Q appear in d, and c_q is true. On Line 18, we first check whether all

```
1: function CkPropDC(c_d, c_p, d, Q, c_q, β, c_b)
2:     if Check sat c_d ∧ c_p = (sat, σ_p) then
3:         Σ_b ←List.map (Unify d) β
4:         Σ'_b ←MergeLL Σ_b
5:         (* Given Σ'_b = σ_b1 :: ··· ::σ_bμ, c'_b = ⋀_{ℓ=1}^{μ} c_b σ_bℓ *)
6:         c'_b ←Conj(Σ'_b, c_b)
7:         (* find all occurrences of q in d *)
8:         Σ ←List.map (Unify d) Q
9:         if nil∈ Σ then
10:            (* check network constraints *)
11:            if Check sat c_d ∧ c_p ∧ (c'_b) = (sat, σ_c) then
12:                (* Network constraints are met *)
13:                return invalid(d, σ_c)
14:            else
15:                return valid
16:        else
17:            Σ_q ←MergeLL Σ
18:            c'_q ←Conj(Σ_q, ¬c_q)
19:            c_s ← c_d ∧ c_p ∧ c'_q ∧ c'_b
20:            if Check sat c_s = (sat, σ_s) then
21:                (* Network constraints are met *)
22:                return invalid(d, σ_s)
23:            else
24:                return valid
25:    else
26:        (* Constraints for p_1 ... p_n and c_p are unsat *)
27:        return valid
28: end function
```

Figure 27: Property query with network constraints

the p_is are derivable and constraint c_p is satisfiable. If the conjunction of the derivation constraint c_d and c_p is not satisfiable, then the precedent of φ is false, so φ is trivially true for that derivation. So, we return valid in the else branch (line 40). If the conjunction is satisfiable, then there are substitutions for variables so that all the p_is are derivable and the constraint c_p is satisfiable. Next, we need to check whether all q_is are derivable. On line 20, function Unify identifies a list of occurrences of q_i in the derivation d. That is, for each $q_i(\vec{y_i})$ appearing in d, Unify returns the list of substitutions: $(\vec{y_1}/\vec{x})::(\vec{y_2}/\vec{x})::\cdots::(\vec{y_n}/\vec{x})::\mathsf{nil}$, where $\vec{x}$ is q_i's arguments in φ. The list map function returns the list of the list of occurrences for all the q_is in Q. We call it "UNIFY" because we unify the variables that are q_i's arguments in φ with q_i's arguments in the derivation d. This substitution will be applied to constraint c_q later. If some q_i does not appear in d, then Unify will return an empty list nil. Therefore, on line 21, we check whether each q_i will appear at least once in d. If it is not the case, then we return invalid with the current derivation and one satisfying substitution that makes p_is true for constructing a counterexample. Otherwise, we check whether the constraint c_q can be satisfied. Before doing so, on line 30, we first compute the list of all possible combinations of occurrences of q_is. Again, the function MergeLL is similar to MergeDLL and we omit the details. Now on line 32, for each possible appearance of q_is in d, Σ_q is a list of substitutions, each of which, when applied to c_q, makes c_q use the same variables as those in the derivation. We ask whether the negation of c_q together with the derivation constraint and the constraint on the arguments of p_is are satisfiable. If this is not satisfiable, then we know that there exists a substitution for variables so that the property φ holds. Otherwise, we return the derivation and the satisfying substitution that makes p_is and q_is derivable, but c_q false for counterexample construction.

4.2.3. Network Constraints

Sometimes, the network being analyzed has certain *network constraints* constraints; for instance, every node in the network has only one outgoing link. Our property query algorithm needs to take into consideration these network constraints. If we ignore these constraints,

the counterexample generated by the tool may not be useful as the counterexample could violate the network constraints.

Network constraints that our analysis can handle have similar form as the properties: $\forall \vec{x_1}.b_1(\vec{x_1}) \wedge \cdots \wedge \forall \vec{x_k}.b_k(\vec{x_k}) \supset c_b(\vec{x_1} \cdots \vec{x_k})$, where b_i is a base tuple. Figure 27 shows the algorithm for checking properties on networks with constraints. For clarity, we explain the case with one network constraint. Extending the algorithm to handle multiple constraints is straightforward.

The top-level function CkPropC (omitted here) is almost the same as CkProp, except that it takes a network constraint (φ_{net}) as an additional argument and uses the function CkPropDC, which additionally checks network constraints compared to CkPropD. The function CkPropDC takes as additional arguments, a list base tuples B and the constraint c_b in the network constraint. In the body of CkPropDC, we first check whether the constraint on p_is is satisfiable (line 2). If it is not, then this derivation does not violate the property we are checking (line 37). Next, between lines 3 to 10, we find all occurrences of the base tuples in the constraint φ_{net}. We find all possible combinations of substitutions for arguments of these base tuples as they appear in the derivation d. For each occurrence of the base tuples, the constraint c_b needs to be true, so we compute the conjunction of all the c_bs. To given an example, if the constraint is $\forall x, b(x) \supset x > 0$. If d has two occurrences of b, $b(y)$ and $b(z)$, then $c'_b = y > 0 \wedge z > 0$.

Next, we collect the list of the occurrences of q_is, the same as before (line 12). If some q_is do not appear in d (line 13), we additionally check whether this derivation d satisfies the network constraint (line 15). If it is the case, then we find a counterexample. Otherwise, d does not violate the property being checked.

Then, we compute the combination of all possible occurrences of q_is in derivation d (line 26) as usual, and find the substitutions that make q_is appear in d. We compute the conjunction of all $\neg c_q$s (line 28). If the conjunction of c_d, c_p, and the conjunction of all the c_bs are found

in lines 3 - 10, and the conjunction of all the $\neg c_q$s is satisfiable, then networks constraints are met although d does not satisfy the property being checked, and we report an error (lines 30 - 34).

4.2.4. Analysis of the Algorithms

Correctness. We first prove that our derivation pool construction is correct. Lemma 2 states that an entry for a predicate p in the derivation pool maps to a valid derivation of p if the constraints of that derivation is satisfiable; and that if a predicate p is derivable, then there must be a corresponding entry in the derivation pool. The function DGraph generates a dependency graph for *prog*, which can be straightforwardly defined. The semantics of NDlog programs are bottom up, so a set of base tuples B is needed to start the execution of the program. We write $\sigma' \geqslant \sigma$ to mean that σ' extends σ. B denotes a set of ground base tuples of *prog*. We write $prog, B \vDash d{:}p(\vec{t})$ to mean that d is a derivation of $p(\vec{t})$ using program *prog* and base tuples B. We write $(c, d'{:}p(\vec{x})) \in dpool(p)$ to mean that (c, d') is an entry in the derivation pool *dpool* for the predicate p and that d' is a derivation tree with $p(\vec{x})$ as the root.

Lemma 2 (Correctness of derivation pool construction)**.**
$\mathrm{D}\mathit{Graph}(prog) = \mathcal{G}$ *and* $\mathrm{G}\mathit{enDPool}(\mathcal{G}) = dpool$

1. *If* $prog, B \vDash d'{:}p(\vec{t})$, *then* $\exists\sigma$ *s.t.* $(c(\vec{x_c}), d(\vec{x_d}){:}p(\vec{x})) \in dpool(p)$, $d(\vec{x_d})\sigma = d'$ *and* $\vDash c(\vec{x_c})\sigma$.

2. *If* $(c(\vec{x_c}), d(\vec{x_d}){:}p(\vec{x})) \in dpool(p)$ *and* $\vDash c(\vec{x_c})\sigma$ *where* $\mathsf{dom}(\sigma) = \vec{x_c}$, *then* $\forall\sigma'$ *s.t.* $\sigma' \geqslant \sigma$ *and* $\mathsf{dom}(\sigma') = \vec{x_d}$, $\exists B$ *s.t.* $B = \{b \mid b$ *is a base tuple and appears in* $d(\vec{x_d})\sigma'\}$ *and* $prog, B \vDash d(\vec{x_d})\sigma'{:}p(\vec{x})\sigma'$.

Using the result of Lemma 2, we prove our property checking algorithm is correct with regard to the formula semantics.

Theorem 3 (Correctness of property query)**.**
$\varphi = \forall\vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall\vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1} \cdots \vec{x_n}) \supset$

$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1} \cdots \vec{x_n}, \vec{y_1} \cdots \vec{y_m})$ $\mathrm{D}\mathit{Graph}(\mathit{prog}) = \mathcal{G}$ *and* $\mathrm{G}\mathit{enDPool}(\mathcal{G}) = \mathit{dpool}$,

1. $\mathrm{C}\mathit{kProp}(\mathit{dpool}, \varphi) = \mathit{valid}$ *implies* $\forall B$, $\mathit{prog}, B \vDash \varphi$.

2. $\mathrm{C}\mathit{kProp}(\mathit{dpool}, \varphi) = \mathit{invalid}(d, \sigma)$, *implies* $\exists B$ *s.t.* $\mathit{prog}, B \nvDash \varphi$.

When network constraints are provided, we prove that the property checking algorithm is correct with regard to the network constraints on base tuples.

Theorem 4 (Correctness of property query with constraints)**.**

$\varphi = \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1} \cdots \vec{x_n}) \supset$

$\exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1} \cdots \vec{x_n}, \vec{y_1} \cdots \vec{y_m})$

$\varphi_{net} = \forall \vec{u_1}.b_1(\vec{u_1}) \wedge \cdots \wedge \forall \vec{u_k}.b_n(\vec{u_k}) \supset c_b(\vec{u_1} \cdots \vec{u_k})$

$\mathrm{D}\mathit{Graph}(\mathit{prog}) = \mathcal{G}$ *and* $\mathrm{G}\mathit{enDPool}(\mathcal{G}) = \mathit{dpool}$,

(1) $\mathrm{C}\mathit{kPropC}(\mathit{dpool}, \varphi_{net}, \varphi) = \mathit{valid}$ *implies* $\forall B$, *either* $\mathit{prog}, B \vDash \varphi$ *or* $B \nvDash \varphi_{net}$.

(2) $\mathrm{C}\mathit{kPropC}(\mathit{dpool}, \varphi_{net}, \varphi) = \mathit{invalid}(d, \sigma)$ *implies* $\exists B$ *s.t.* $\mathit{prog}, B \nvDash \varphi$ *and* $B \vDash \varphi_{net}$.

Time complexity. We give an upper bound on the time complexity of the property query algorithm (Figure 26). Given an NDlog program with R rules; each rule contains at most W body tuples. Also assume $|Q| = m$ and $|P| = n$. The time complexity of our algorithm is $O((R^{nW^R})n^m W^{Rn})$. In practice, R and W are usually small. For example, in our case study, R is bounded by 11 and W is bounded by 5. In this case, R and W can be viewed as constants.

In CkPropD, we assume $|Q| = m$, and d contains at most D_q instances for each q_i in Q. Also, assume each query of Z3 takes a constant time t. Therefore, the size of Σ_q in line 29 is bounded by $(D_q)^m$, and the running time of the loop (line 30 - line 34) is bounded by $(D_q)^m t$. So the time complexity of CkPropD is $O((D_q)^m t)$.

The time complexity of CkProp in Figure 26 is dominated by the loop in the algorithm,

i.e., line 10 - line 13. Suppose each p_i in P has at most D_p instances in d, and $|P| = n$, then for D in line 10, we have $|D| \leqslant (D_p)^n$. Based on the above discussion, each loop takes $O((D_q)^m t)$. Therefore in the worst case, CkProp takes $O((D_p)^n (D_q)^m t)$ to finish.

We give more detailed estimate of D_p and D_q. Assume the input NDlog program $Prog$ has R rules, and each rule has at most W body tuples. Define the height H of a derivation tree as the number of rules on the longest path from the root predicate to any leaf predicate. Also, let $D_p(k)$ represent the maximum possible number of derivations for a predicate all of whose derivation trees have height at most k. Notice that $Prog$ is non-recursive, so we have $k <= R$, and $D_p = D_p(k)$. We have the following theorem:

Theorem 5. *Given $k \geqslant 1$ (k is a natural number), $D_p(k) \leqslant R^{\sum_{j=0}^{k-1} W^j}$.*

Proof. We prove Theorem 5 with mathematical induction. The base case is straightforward. When $k = 1$, we have $D_p(1) \leqslant R$. This is true because $k = 1$ means that the head predicate can only be derived using one rule with base tuples as bodies. Since there are at most R rules, the maximum number of derivations for the predicate is R. For the inductive case, assume Theorem 5 holds for $k = n - 1$, which means $D_p(n-1) \leqslant R^{\sum_{j=0}^{n-2} W^j}$. Now consider a predicate p' whose derivation trees' maximum height is n. Given a rule r that derives p', each body tuple of r has all its derivation trees' height bounded by $n - 1$. Remember that r has at most W body tuples. Thus the number of all possible derivations of p' using rule r is bounded by $D_p(n-1)^W$. Since there could be at most R rules that derives p', we have $D_p(n) \leqslant R(D_p(n-1))^W = R^{\sum_{j=0}^{n-1} W^j}$. □

Based on Theorem 5, $D_p = O(R^{W^R})$. Next, we calculate D_q. Given a q_i in Q, notice that q_i must appear in one of all derivations of $p_i's$ in P, which means the number of q_i's appearance is bounded by the number of nodes that could exist in all the derivations. Each such derivation has height at most R, with each node in the derivation having at most W children. Therefore, the maximum number of nodes in a derivation is W^R. Since there are n derivations corresponding to $p_i's$ in P, we have $D_q = nW^R$. Replace D_p and D_q in the

time complexity of CkProp, we have $O((D_p)^n(D_q)^m t) = O((R^{nW^R})n^m W^{Rn})$.

4.3. Extension to Recursive Programs

The dependency graph for a recursive program contains cycles. The derivation pool construction algorithm presented in Figure 24 does not work for recursive programs because it relies on the topological order of nodes in the dependency graph. In this section, we show how to augment our data structures and algorithms to handle recursive programs.

4.3.1. Derivation Pool for Recursive Predicates

When p is recursively defined, *dpool* maps p to a pair (c, Δ), where Δ has the same meaning as before. The additional constraint c is an invariant of p: c is satisfiable if and only if p is derivable.

Constraint pool	*dpool*	::=	$\cdots \mid dpool, (nID, p{:}\tau) \mapsto (c, \Delta)$
Derivation	$\mathcal{D}$	::=	$\cdots \mid (\mathsf{rec}, p(\vec{x}))$
Annotation	A	::=	$\cdot \mid A, (nID, p{:}\tau) \mapsto (\vec{x}, c)$

Derivation trees include a new leaf node $(\mathsf{rec}, p(\vec{x}))$, where p appears on a cycle in the dependency graph. This leaf node is a place holder for the derivation of p. We write A to denote annotations for recursive predicates, provided by the user. A maps a predicate p to a pair $(\vec{x}, c)$, where $\vec{x}$ is the arguments of p and c is the constraint which is satisfiable if and only if p is derivable.

The structure of the derivation pool construction remains the same. We highlight the changes in Figure 28. The main difference is that now when a cycle is reached, the annotations are used to break the cycle. The working set P, which contains the set of nodes that can be processed next, includes not only predicate nodes that do not have incoming edges, but also includes nodes that depend on only body tuples that have annotations. Consider the following scenario: Rule $r1$ derives p and has two body tuples q_1 and q_2. Let's assume that there is no edge from q_1 to $r1$, as q_1 has been processed and q_2 has an annotation in A. In this case, we will place p in the working set. The above mentioned change is encoded

in the new RemoveEdges function in Figure 28.

The second change is in constructing derivation pool entries for a predicate p. In the non-recursive case, each derivation tree of a predicate p corresponds to the application of a rule to the list of derivation trees for the body tuples of that rule. In the recursive case, if one of the body tuples, say q, is on a cycle, when we process p, q's entries in *dpool* have not been constructed. However, the constraint under which q can be derived is given in the annotation A. In this case, we use $(\mathsf{rec}, q(\vec{x}))$ as a place holder for derivations for q, and use the constraint in A as the constraint for this derivation. The change is reflected in the LookUp function for collecting possible derivations of the body predicates (lines 21-23).

Finally, annotations need to be verified. The GenDs function checks the correctness of the annotations after all the predicates have been processed (lines 5-15). For a recursive predicate, the derivation pool maps it to a summary constraint and a list of possible derivations (a pair (c, Δ)). The requirement of the summary constraint for p is that it has to be satisfiable if and only if there is at least one derivation for the recursive predicate p. That is, this summary constraint has to be logically equivalent to the disjunction of the constraints associated with all possible derivations of p in Δ. We consider two cases for a predicate on a cycle of the dependency graph: (1) there is an annotation for it in A and (2) there is no annotation. For both cases, we need to collect all the possible constraints for deriving p from Δ. Function EX_Disj computes the disjunction of constraints in Δ. Each constraint is existentially quantified over the arguments that do not appear in p. For case (1), we need to check that the annotation is logically equivalent to the disjunction of the constraints for all possible derivations of p (line 10). If this is the case, then the annotated constraint together with Δ is returned; otherwise, an error is returned, indicating that the invariant doesn't hold. For case (2), we return the disjunctive formula returned by EX_Disj (Lines 15). When p is not recursive, only Δ is returned (line 17).

4.3.2. Property Query

We use the same property query algorithm for non-recursive program. This obviously has limitations, because the derivations of recursive predicates are not expanded. The imprecision of the analysis comes from the following two sources. The first is that derivations represented as $(\mathsf{r}ec, p(\vec{x}))$ may contain predicates needed by the antecedent of the property (the q_is in φ). Without expanding these derivations, the algorithm may report that φ is violated because q_is cannot be found, even though this is not the case in reality. The second is that network constraints cannot be accurately checked. When we find a suitable derivation d that contains all the q_is such that c_q holds, checking the network constraints on d requires us to expand $(\mathsf{r}ec, p(\vec{x}))$s in d. The algorithm may report that the property holds, even though, the witness it finds does not satisfy the network constraints. Similarly, when the algorithm reports that the property does not hold, the counterexample may not satisfy the network constraints. For the analysis to be precise, we would need annotations for recursive predicates to provide invariants for recursive predicates. Our case studies do not require annotations. Expanding the algorithm to handle recursive predicates precisely remains our future work.

4.3.3. Analysis of the Algorithms

Correctness. Similar to the non-recursive case, we prove the correctness of derivation pool construction. We only prove the soundness of the query algorithm. Because derivations of recursive predicates are summarized as $(\mathsf{r}ec, p(\vec{x}))$, the correctness of the derivation pool construction needs to consider the unrolling of $(\mathsf{r}ec, p(\vec{x}))$.

First, we define a relation $dpool \vdash d_k, \sigma_k \rightsquigarrow_{k+1} d_{k+1}, \sigma_{k+1}$ (for $k \geqslant 0$) to mean that a derivation d_k with the substitution σ_k can be unrolled using derivations to *dpool* to another derivation d_{k+1} and a new substitution σ_{k+1}. The rules defining this relation allow the $(\mathsf{r}ec, \_)$ leafs in the derivation to be gradually expanded, starting from the root and moving up the tree.

We write d_k to denote the derivation after unrolling a derivation d for a sequence of k steps: $d, \sigma_0 \rightsquigarrow_0 d, \sigma_0 \rightsquigarrow_1 d_1, \sigma_1 \rightsquigarrow_2 \ldots \rightsquigarrow_i d_i, \sigma_i \rightsquigarrow_{i+1} d_{i+1}, \sigma_{i+1} \rightsquigarrow_{i+1} \ldots \rightsquigarrow_k d_k, \sigma_k$. The $\rightsquigarrow_{i+1}$ rules ensure that each d_i has no $(\mathit{rec}, \_)$ leafs from the root up to the $i-1^{th}$ level Step $d_i, \sigma_i \rightsquigarrow_{i+1} d_{i+1}, \sigma_{i+1}$ expands the $(\mathit{rec}, \_)$ leafs at the i^{th} level of d_i, and thus d_{i+1} has no $(\mathit{rec}, \_)$ leafs from the root up to the i^{th} level.

Figure 29 shows the inference rules for proving correctness of recursive cases. Rule Base does not extend the derivation. Rule WkInd weakens the index from k to a larger number n when derivation d does not contain any recursive subderivations of form $(\mathit{rec}, \_)$. Given a derivation $(\mathit{rID}, p(\vec{x}), d_1 :: \ldots :: d_n :: \mathsf{nil})$ with no $(\mathit{rec}, \_)$ leafs from the root up to the $k-1^{th}$ level, and whose subderivations $d_1, \ldots, d_n$ that can be expanded to subderivations $d'_1, \ldots, d'_n$ which have no $(\mathit{rec}, \_)$ leafs from the root to the $k-1^{th}$ level, then rule Rnrec expands $(\mathit{rID}, p(\vec{x}), d_1 :: \ldots :: d_n :: \mathsf{nil})$ to derivation $(\mathit{rID}, p(\vec{x}), d'_1 :: \ldots :: d'_n :: \mathsf{nil})$, which has no $(\mathit{rec}, \_)$ leafs from the root up to the the k^{th} level. The last rule, Rrec, is the key rule that expands the derivation of the recursive predicate p at the root in one step. Recursive derivation $(\mathit{rec}, p(\vec{x}))$ with substitution σ is expanded to a derivation $d{:}p(\vec{x}) \in \Delta_p$ whose constraint c is satisfiable for some substitution $\sigma \cup \sigma'$.

We write $d_0, \sigma_0 \longmapsto d_k, \sigma_k$ as shorthand notation for the above sequence of steps.

Lemma 6 shows that the derivation pool construction algorithm is correct with respect to an unrolling of the derivation. If a predicate p is derivable, then the derivation pool will have an entry for for p which can be unrolled into that concrete derivation. For every natural number ℓ, if there is a skeleton derivation d for p in the derivation pool and the corresponding constraint to derive d is satisfiable, then either we can unroll d to a concrete derivation d' for p within ℓ steps, where d' does not contain any subderivations of form $(\mathit{rec}, q(\vec{x_q}))$, or after unrolling d for ℓ steps, the resultant derivation d' contains some recursive subderivations of form $(\mathit{rec}, q(\vec{x_q}))$, and if every $(\mathit{rec}, q(\vec{x_q}))$ can be unrolled to a concrete derivation, then the derivation d for predicate p can be unrolled to a concrete derivation.

We state the key points of the lemma in this section.

Lemma 6 (Correctness of derivation pool construction (recursive))**.** D*Graph*(*prog*) = $\mathcal{G}$, *and* G*enDPool*($\mathcal{G}, A$) = *dpool*

1. *If* $prog, B \models d{:}p(\vec{t})$, *then either* p *is not on a cycle in* $\mathcal{G}$ *and* $\exists(c(\vec{x_c}), d'(\vec{x_d}'){:}p(\vec{x})) \in dpool(p)$, $\exists\sigma''_d$ *where* $\models c(\vec{x_c})\sigma''_d|_{\vec{x_c}}$, *s.t. using substitution* σ''_d, $d'(\vec{x_d}')$ *can be unrolled into* d, *or* p *is on a cycle in* $\mathcal{G}$ *and* $\exists(c_p(\vec{x}), \Delta_p) \in dpool(p)$, $\exists\sigma''_d$ *s.t.* $\models c_p(\vec{x})\sigma''_d|_{\vec{x}}$, *s.t. using substitution* σ''_d, $(rec, p(\vec{x}))$ *can be unrolled into* d.

2. $\forall \ell \in \mathbb{N}$,

   (a) *If* $(c(\vec{x_c}), d(\vec{x_d}){:}p(\vec{x})) \in dpool(p)$ *and* $\models c(\vec{x_c})\sigma$, *either* $\exists d'(\vec{x_d})$ *s.t.* $d'(\vec{x_d}')$ *does not contain any* $(rec, \_)$, $d(\vec{x_d})$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $d'(\vec{x_d}')$ *can be unrolled to* d *with an appropriate extension of substitution* σ, *or* $\exists d'(\vec{x_d}')$ *s.t.* $d'(\vec{x_d}')$ *contains some* $(rec, s(\vec{x_s}))$, $d(\vec{x_d})$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $\forall(rec, s(\vec{x_s})) \in d'(\vec{x_d}')$, $\exists d_s, \ell_s$ *s.t.* $(rec, s(\vec{x_s}))$ *can be unrolled to* d_s *with an appropriate extension of substitution* $\sigma|_{\vec{x_s}}$ *implies* $\exists d''$ *s.t.* $prog, B \models d''$ *and* $d(\vec{x_d})$ *and be unrolled to* d'' *with an appropriate extension of substitution* σ.

   (b) *If* $(c_{rec:p}(\vec{x}), \Delta_p) \in dpool(p)$ *and* $\models c_{rec:p}(\vec{x})\sigma$, *then either* $\exists d'(\vec{x_d})$ *s.t.* $d'(\vec{x_d}')$ *does not contain any* $(rec, \_)$, $(rec, p(\vec{x}))$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $d'(\vec{x_d}')$ *can be unrolled to* d *with an appropriate extension of substitution* σ, *or* $\exists d'(\vec{x_d}')$ *s.t.* $d'(\vec{x_d}')$ *contains some* $(rec, s(\vec{x_s}))$, $(rec, p(\vec{x}))$ *can be unrolled to* $d'(\vec{x_d}')$ *in* ℓ *steps, and* $\forall(rec, s(\vec{x_s})) \in d'(\vec{x_d}')$, $\exists d_s, \ell_s$ *s.t.* $(rec, s(\vec{x_s}))$ *can be unrolled to* d_s *with an appropriate extension of substitution* $\sigma|_{\vec{x_s}}$ *implies* $\exists d''$ *s.t.* $prog, B \models d''$ *and* $(rec, p(\vec{x}))$ *can be unrolled to* d'' *with an appropriate extension of substitution* σ.

As we discussed in Section 4.3.2, we cannot show a general correctness theorem without annotations for recursive predicates. We can only prove the soundness of the algorithm when there is no network constraint.

Lemma 7 (Soundness of property query)**.**

$\varphi = \forall \vec{x_1}.p_1(\vec{x_1}) \wedge \cdots \wedge \forall \vec{x_n}.p_n(\vec{x_n}) \wedge c_p(\vec{x_1} \cdots \vec{x_n}) \supset$

$\quad \exists \vec{y_1}.q_1(\vec{y_1}) \wedge \cdots \wedge \exists \vec{y_m}.q_m(\vec{y_m}) \wedge c_q(\vec{x_1} \cdots \vec{x_n}, \vec{y_1} \cdots \vec{y_m})$

$\mathrm{D}\mathit{Graph(prog)} = \mathcal{G}$ *and* $\mathrm{G}\mathit{enDPool}(\mathcal{G}, A) = \mathit{dpool}$ *and*

$\mathrm{C}\mathit{kProp}(\mathit{dpool}, \varphi) = \mathit{valid}$ *implies* $\forall B,\ \mathit{prog}, B \models \varphi$.

Time complexity. The time complexity of the property query algorithm on recursive programs is the same as that of non-recursive programs. More concretely, we also use CkProp to check properties on recursive programs, so the time complexity remains $O((D_p)^n(D_q)^m t)$. In addition, the estimate about D_p and D_q remains the same as in non-recursive case. Observe that the height of a derivation in the derivation pool is still bounded by R. This is because in the derivation pool construction algorithm (Figure 28), each rule node is processed at most once. Therefore a path in a derivation from the root predicate to any leaf predicate could have at most R rules. So we directly have $D_q = nW^R$. For D_p, Theorem 5 also holds true. The base case is unchanged. For the inductive step, the body tuple now could be a recursive tuple with user's annotation. However, since we do not expand the derivations for recursive tuples, but collect its annotation as a constraint, the number of derivations for the recursive tuple is effectively one, which satisfies the inductive hypothesis. The rest of the proof is the same. In conclusion, Theorem 5 remains true. Together with the bound on the height of derivations (i.e., R), this means that $D_p = O(R^{W^R})$. And the whole complexity remains unchanged.

4.4. Case Study

We apply our tool to the verification of software-defined networking (SDN) applications. SDN is an emerging networking technique that allows network administrators to program the network through well-defined interfaces (e.g., OpenFlow protocol [57]). SDNs intentionally separate the control plane and the data plane of the network. A centralized controller is introduced to monitor and manage the whole network. The controller provides an abstraction of the network to network administrators, and establishes connections with underlying

switches. Recently, declarative programming languages have been used to to write SDN controller applications [61]. Like any program, these applications are not guaranteed to be bug-free. We show the effectiveness of our tool in validating and debugging several SDN applications. We demonstrate that the tool can unveil problems in the process of SDN application development, ranging from software bugs, incomplete topological constraints and incorrect property specification. All verifications in our case study are completed within one second.

4.4.1. Verification Process

We first provide a high-level description of the verification process. When analyzing a property, the user is expected to provide three types of inputs: (1) formal specification of the property in the form discussed in Section 5.2; (2) formal specification of initial network constraints (e.g., topological constraints and switch default setup); and (3) formal specification of invariants on recursive tuples.

Our tool takes the above user specifications along with the NDlog program as inputs. It first checks the correctness of the invariants on recursive tuples. After invariants are validated, the tool runs the main algorithm for verification, and outputs either "True" if the property holds, or "False" if the property is not valid. For invalid properties, the tool also generates a concrete counterexample to help the programmer debug the program.

4.4.2. Ethernet Source Learning

The first case study we consider is Ethernet source learning, which allows switches in a network to remember the location of end hosts through incoming packets. More specifically, three kinds of entities are deployed in the network: (1) **end hosts** (servers or desktops) at the edge of the network that send packets to the network through connected switches, (2) **switches** that forward a packet if the packet matches a flow entry in the forwarding table, or relay the packet to the controller for further instruction if there is a table miss, and (3) **a controller** that connects to all switches in the network. The controller learns the

position of an end host through packets relayed from a switch, and installs a corresponding flow entry in the switch for future forwarding.

Encoding We encode the behaviors of each component in NDlog. Figure 30 presents the NDlog encoding of Ethernet Source Learning ($prog_{ESL}$). In a typical scenario, an end host initiates a packet and sends it to the switch that it connects to (rh1). The switch recursively looks up its forwarding table to match against the received packet (rs1, rs2). If a flow entry matches the packet, it is forwarded to the port indicated by the "Action" part of the entry (rs3). Otherwise, the switch wraps the packet in an OpenFlow message, and relays it to the controller for further instruction (rs5). On receiving the OpenFlow message, the controller first extracts the location information of the source address in the packet (the OpenFlow message registers incoming port for each packet), and installs a flow entry matching the source address in the switch (rc1). The controller then instructs the switch to broadcast the mismatched packet to all its neighbors other than the upstream neighbor who sent the packet (rc2). Rules rs5 and rs6 specify the reaction of the switch corresponding to Rules rc1 and rc2 respectively — the switch either inserts a flow entry into the forwarding table (rs5) or broadcasts the packet (rs6) as instructed.

Network constraints We use the basic network constraints in Figure 31 to limit the topology of the network that runs Ethernet source learning.

We demand that an end host always initiates packets using its own address as source, and the switch it connects to cannot be the source or the destination (constraints on initPacket). In addition, the controller cannot share addresses with switches (constraints on ofconn), and a switch cannot have a link to itself (constraints on single swToHst). Also, each switch should have only one link connecting the neighbor host, and no two hosts can connect to the same port of a switch (constraints on any two swToHsts).

Verification results We verify a number of safety properties that are expected to hold in a network running the Ethernet Source Learning program. Table 3 lists all the properties

of the program that we investigate. Here we discuss properties in detail.

Property φ_{ESL_2} specifies that whenever an end host receives a packet not destined to it, the switch that it connects to has no matching flow entry for the destination address in the packet. Though this property is seemingly true, our tool returns a negative answer, along with a counterexample shown in Figure 32. The counterexample reveals a scenario where an endhost (H4) receives a broadcast packet destined to another machine (H3) (Execution trace (1) in Figure 32), but the switch it connects to (S1) has a flowEntry that matches the destination MAC address in the packet (Execution trace (2) in Figure 32).

In the counterexample, switch S1 receives a packet $\langle Src : H6, Dst : H3\rangle$ through port 2 from the upstream switch S2 (①). Since S1 does not have a flow entry for the destination address H3, it relays the packet wrapped in an OpenFlow message (i.e. ofPacket) to the controller C1(②). The controller then instructs S1 to broadcast the packet to all neighbors except S2 (③). However, before Server H4 receives the broadcast packet, a new packet $\langle Src : H3, Dst : H4\rangle$ could reach switch S1(④), triggering an ofPacket message to the controller (⑤). The controller would then set up a new flow entry at switch S1, matching destination H3 (⑥,⑦). It is possible that due to network delay, server H4 receives its copy of the broadcast packet just now(⑧). Therefore, the execution trace generates packet (H4,S1,H6,H3), swToHst (S1,H4,1) (i.e. the link between S1 and H4), and flowEntry (S1,H3,2,1), with $Mac == DstMac$ ($H3 = H3$).

Our tool also generates a counterexample for another seemingly correct property—φ_{ESL_3}. This property specifies that whenever an end host receives a packet destined to it, the switch it connects to has a flowEntry matching the end host's MAC address. The generated counterexample (Figure 33) shows that a packet could reach the correct destination by means of broadcast — a corner case that can be easily missed with manual inspection. In the counterexample, switch S1 receives a packet destined to server H4(①). Since there is no flow entry in the forwarding table to match the destination address, switch S1 informs the controller of the received packet (②), and further broadcasts the packet under the

controller's instruction (③). In this way, server H4 does receive a packet destined to it (④), but switch S1 does not have a flow entry matching H4.

With further inspection, the above counterexamples, are attributed to incorrect specification of network properties, rather than bugs in the programs. In the first case, a stricter property would specify that a received broadcast message indicates an *earlier* table miss. While in the second one, the property fails to consider the possibility of specific broadcast messages in the execution.

4.4.3. Firewall

Our second case study is a stateful firewall, which is usually deployed at the edge of a corporate network to filter untrusted packets from the Internet. Compared to a stateless firewall, which makes decision purely based on specific fields of a packet, a stateful firewall allows richer access control depending on flow history. For example, the firewall can allow traffic from an outside end host to reach machines inside the local domain only if the communication was initiated by the internal machines.

Encoding We implement a SDN-based stateful firewall, which can set up filtering policies under the instruction of the controller. The detailed encoding of the program can be found in Figure 35. The firewall enforces the following policy: end-hosts in the corporate domain can send traffic to the outside world but, for security reasons, traffic from an end-host outside the domain can only enter the domain if that end-host has previously received traffic from some end-host within the domain. The network is configured as follows. Two types of hosts are connected to a switch: (i) trusted hosts (within the organization) via port 1; and (ii) untrusted hosts (outside the organization) via port 2. Packets from trusted hosts are always forwarded to untrusted hosts. Packets from untrusted hosts are forwarded to trusted hosts only if the source host has previously received a packet from a trusted host. Key tuples generated at each node executing the program are listed in Table 4. Table 5 summarizes all the rules in the firewall program.

Network constraints The network constraints for the base tuples in the firewall program are given in Figure 36.

Verification results We verify a number of properties about the stateful firewall, which are listed below. All the properties are valid.

Property φ_{FW_1} states that for every packet a trusted host receives from an untrusted host via Switch, in the past the switch has received a packet sent from some trusted host (via port 1) to the untrusted host (via port 2). Property φ_{FW_2} states that if the flow table on the switch contains an entry between Src (via untrusted port) and Dst (via trusted port), then in the past the switch has received a packet sent from some Host (via trusted port) to Src. Property φ_{FW_3} states that the trusted controller memory records a connection between Switch and a host, then in the past some trusted source had sent a packet to that host.

4.4.4. A Weak Firewall

To further evaluate our tool, we modify the above stateful firewall slightly to construct a "weaker" firewall. The new firewall differs by not requiring packets received from the trusted port to be forwarded to the Internet.

Encoding We present our implementation of the program ($prog_{WeakFW}$) in Figure 38. Key tuples generated at each node executing the program are listed in Table 6. The firewall forwards traffic from trusted hosts in the local domain without interference (r1), and also notifies the controller of the destination address in the packet (r2). When the firewall receives a packet from the Internet, it relays the packet to the controller for further decision (r4). If the source address was once registered at the controller, the controller would install a flow entry in the firewall (r5), allowing packets of the same flow to access the internal domain in the future (r3).

Network constraints The network constraints for this modified version of firewall are shown in Figure 39. They are similar to those given in the case study of firewall (Sec-

tion 4.4.3), but with an additional link tuple.

Verification results We check the property φ_{WeakFW} over the weak firewall:

$$\varphi_{WeakFW} =$$
$$\forall Host, Port, Src, SrcPort, Switch,$$
$$\mathsf{pktReceived}(Host, Port, Src, SrcPort, Switch) \supset$$
$$\exists Cntrl, \mathsf{trustedControllerMemory}(@Cntrl, Switch, Src)$$

The property specifies that source destinations of all packets reaching internal machines are trusted by the controller. Surprisingly, our tool gives a counterexample for this property (Figure 34), which depicts the scenario that an internal machine H3 sends a packet to another internal machine H4 in the same domain through the firewall F1. Because the controller C1 never registers local machines, the property is violated.

Despite its simplicity, we find the counterexample interesting, because it can be interpreted in different ways; each corresponds to a different approach to fixing the problem. The counterexample can be viewed as revelation of a program bug. The user can add a patch to the program and re-verify the property over the updated program. Alternatively, the counterexample could be linked to incomplete specification of network constraints that internal machines should never send internal traffic to the firewall. The fix would then be to insert extra constraints over base tuples of the program. In addition, the problem could also stem from the property specification, since users may only care about traffic from outside the domain. In this case, we can change the property specification, specifying that if a packet is from an *external* machine, the source address must be registered at the controller before. In real deployment, it is up to the user to decide which interpretation is most appropriate.

4.4.5. Load Balancing

The third case study is load balancing. When receiving packets to a specific network service (e.g., web page requests), a typical load balancer splits the packets on different network

paths to balance traffic load. There are a number of strategies for load balancing, e.g., static configuration or congestion-based adjustment. In our case study, we implement a load balancer which load balances traffic towards a specific destination address, and determines the path of a packet based on the hash value of its source address.

Encoding Figure 40 presents our implementation of load balancer implemented in NDlog ($prog_{LB}$). Key tuples generated at each node executing the program are listed in Table 7. We summarize the program in Table 8.

When the load balancer receives a packet, it first inspects its destination address. If the destination address matches the address that the load balancer is responsible for, the load balancer would generate a hash value of the source destination of the packet (r1). The hash value is used to select the server to which the packet should be routed. The load balancer replaces the original destination address in the packet with the address of the selected server, and forwards the packet to the server (r2). In addition, the load balancer has a default rule that forwards traffic not destined to the designated address without interference (r3).

Network constraints The network constraints are shown in Figure 41.

Verification result The property that we verify for load balancing is called flow affinity, that is, if two servers receives packets requesting the same service—which means the packets share the same initial destination address—the source addresses of the packets must be different. Formally:

$$
\begin{aligned}
&\forall \mathit{Server1}, \mathit{Server2}, \mathit{Src1}, \mathit{Src2}, \\
&\quad \mathsf{recvPacket}(\mathit{Server1}, \mathit{Src1}, \mathit{ServiceAddr}) \\
&\quad \wedge \mathsf{recvPacket}(\mathit{Server2}, \mathit{Src2}, \mathit{ServiceAddr}) \\
&\quad \wedge \mathit{Server1} \neq \mathit{Server2} \supset \\
&\qquad \mathit{Src1} \neq \mathit{Src2}
\end{aligned}
$$

The property does not hold in the given protocol specification, and a counterexample is given by our tool (Figure 42). In the counterexample, two load balancers responsible for different network service could co-exist in the network, and if a server sends packets to both load-balancers, requesting the same service, it is possible that the packets are routed to different servers.

Similar to the case of the firewall, the programmer can fix the counterexample of the load balancer by patching the program, adding network assumption (e.g., assuming no server is connected to two load-balancers), or changing property specification (e.g., "load-balanced packets that are forwarded out of different ports of the load balancer do not share the same source address").

4.4.6. Ethernet Address Resolution

The final case study we focus on is the Address Resolution Protocol (ARP) in an Ethernet network. End hosts use ARP to request the destination MAC address corresponding to an IP address that they want to communicate to. Traditionally, the ARP requests are broadcast through the domain. In our case study, we replace the broadcast with a centralized controller that answers ARP requests.

Encoding Figure 43 presents an implementation of our NDlog encoding of SDN-based ARP ($prog_{ARP}$). Key tuples generated at each node executing the program are listed in Table 9.

Network constraints The network constraints of ARP are defined in Figure 44.

Verification results We verified two safety properties on the ARP program (Figure 10). All properties are valid.

4.4.7. Discussion

We discuss our experience of using the tool and insights obtained from the case studies.

Overhead of Annotations Our tool demands that the user specify annotations for recursive predicates in recursive programs (Section 4.3.1). Since this is a manual process and requires domain-specific knowledge of the user, the whole verification time depends largely on the time spent in annotation specification. Based on our empirical experience when verifying recursive programs in the case study (e.g., Ethernet source learning), most annotations are straightforward – which involve simple equality/inequality between attributes, and in certain cases, could be empty at all (e.g., φ_{ESL_4} in Table 3). In most scenarios, the discovery of annotations takes only several minutes.

Cause of property violation The counterexamples we discuss above reveal a common pattern: when a predicate in the program has multiple derivations, proving properties over the predicate becomes harder. The situation is even worse when a property involves multiple predicates, each with multiple derivations. The increased complexity of predicate derivations makes it error-prone for human programmers to write correct programs or specify correct properties, and serves as the core cause of property violation. Naturally, the fixes we proposed for counterexamples generally fall into two categories: (1) enriching the property specification to include the missing derivations, or (2) changing the program to remove the uncovered derivations.

Iterative application development Another observation is that reasonable network assumptions (e.g., topological constraints) helps prune scenarios that would not appear in actual executions, and generate insightful counterexamples. For example, a counterexample may suggest a topology where a switch has a link to itself. A programmer may start with trivial network assumptions and let the tool guide the exploration of corner cases and gradually add (implicit) network assumptions that are not obvious to the programmer. In fact, our tool enables the programmer to *iteratively* develop applications. The generated counterexamples could help the programmer understand (1) applicable domain of the pro-

gram (feedback of missing network constraints); (2) implementation correctness (feedback of bugs in the program); and/or (3) expected behavior of the program (feedback of incorrect property specification). After the programmer fix the problem, she or he can redo the verification repeatedly until the specified property holds.

```
1: function GenDs(G, dpool, (nID, p:τ))
2:     Δ ← {}
3:     for each rule with ID rID where (rID,nID) in G do
4:         Δ ← Δ∪GenDRule(G, dpool, (nID, p:τ),rID)
5:     if (nID, p:τ) is on a cycle then
6:         (* gather all constraints *)
7:         (x⃗, c) ← Ex_DisJ(Δ)
8:         if A(p) = (y⃗, c_A) then
9:             (* check annotation *)
10:            if Check sat ¬(c_A[x⃗/y⃗]⇔c)=(sat, _) then
11:                return annotation_error
12:            else
13:                return (c_A, Δ)
14:        else
15:            return (c, Δ)
16:    else
17:        return Δ
18: end function
19:
20: function LookUp(dpool, q(x⃗))
21:    if q ∈ A then
22:        (y⃗, c_A) ← A(q)
23:        return (y⃗/x⃗, c_A, (rec, q(y⃗))::nil)
24:    else
25:        if dpool(q) = Δ then
26:            return List.Map (ExtractD x⃗) Δ
27:        else
28:            dpool(q) = (c_q(y⃗), Δ_q)
29:            return (y⃗/x⃗, c, (rec, q(y⃗))::nil)
30: end function
31:
32: function LookUpRec(dpool, q(x⃗))
33:    if dpool(q) = Δ then
34:        return List.Map (ExtractD x⃗) Δ
35:    else
36:        dpool(q) = (c_q(y⃗), Δ_q)
37:        return List.Map (ExtractD x⃗) Δ_q
38: end function
39:
40: function RemoveEdges(P, E, G)
41:    remove outgoing edges of nID from E
42:    for each rID with no edges of form (_, rID) in E do
43:        remove edges (rID, nID) from E
44:        if (nID, p : τ) has no incoming edges in E then
45:            add (nID, p : τ) to P
46:        if every (nID′, q : τ′) s.t. (nID′, rID), (rID, nID) ∈ E, (nID′, q : τ′) in A then
47:            add (nID, p : τ) to P
48: end function
```

Figure 28: Construct derivation pools for recursive programs

$$\textsc{Base}\ \frac{}{dpool \vdash d, \sigma \rightsquigarrow_0 d, \sigma}$$

$$\textsc{WkInd}\ \frac{dpool \vdash d, \sigma \rightsquigarrow_k d, \sigma \qquad k < n \qquad d \text{ does not contain } (\mathsf{rec}, \_) \text{ as subderivations}}{dpool \vdash d, \sigma \rightsquigarrow_n d, \sigma}$$

$$\textsc{Rnrec}\ \frac{\forall i \in [1, n], dpool \vdash d_i, \sigma \rightsquigarrow_k d'_i, \sigma_i \qquad \sigma' = \bigcup_{i=1}^{n} \sigma_i}{dpool \vdash (rID, p(\vec{x}), d_1 :: \ldots :: d_n :: \mathsf{nil}), \sigma \qquad \rightsquigarrow_{k+1} (rID, p(\vec{x}), d'_1 :: \ldots :: d'_n :: \mathsf{nil}), \sigma'}$$

$$\textsc{Rrec}\ \frac{\begin{array}{c} dpool(p) = (c_p, \Delta_p) \qquad (c(\vec{z_c}), d(\vec{z_d}) : p(\vec{z})) \in \Delta_p \qquad \vec{z_d}' = fresh(\vec{z_d} \backslash \vec{z}) \\ \vDash c(\vec{z_c})[\vec{z_d}'/(\vec{z_d} \backslash \vec{z})][\vec{x}/\vec{z}](\sigma \cup \sigma') \end{array}}{dpool \vdash (\mathsf{rec}, p(\vec{x})), \sigma \rightsquigarrow_1 d(\vec{z_d})[\vec{z_d}'/(\vec{z_d} \backslash \vec{z})][\vec{x}/\vec{z}], \sigma \cup \sigma'}$$

Figure 29: Inference rules for correctness proof of recursive NDlog programs

```
/* Controller program */
rc1 flowMod(@Switch, SrcMac, InPort) :-
  ofconn(@Controller, Switch),
  ofPacket(@Controller, Switch, InPort, SrcMac, DstMac).

rc2 broadcast(@Switch, InPort, SrcMac, DstMac) :-
  ofconn(@Controller, Switch),
  ofPacket(@Controller, Switch, InPort, SrcMac, DstMac).

/* Switch program */
rs1 matchingPacket(@Switch, SrcMac, DstMac, InPort, TopPriority) :-
  packet(@Switch, Nei, SrcMac, DstMac),
  swToHst(@Switch, Nei, InPort),
  maxPriority(@Switch, TopPriority).

rs2 matchingPacket(@Switch, SrcMac, DstMac, InPort, NextPriority) :-
  matchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
  flowEntry(@Switch, MacAdd, OutPort, Priority),
  Priority > 0, DstMac != MacAdd, NextPriority := Priority - 1.

rs3 packet(@OutNei, Switch, SrcMac, DstMac) :-
  matchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
  flowEntry(@Switch, MacAdd, OutPort, Priority),
  swToHst(@Switch, OutNei, OutPort),
  Priority > 0, DstMac == MacAdd.

rs4 ofPacket(@Controller, Switch, InPort, SrcMac, DstMac) :-
  ofconn(@Switch, Controller),
  matchingPacket(@Switch, SrcMac, DstMac, InPort, Priority),
  Priority == 0.

rs5 flowEntry(@Switch, DstMac, OutPort, Priority) :-
  flowMod(@Switch, DstMac, OutPort),
  ofconn(@Switch, Controller),
  maxPriority(@Switch, TopPriority), Priority := TopPriority + 1.

rs6 packet(@OutNei, Switch, SrcMac, DstMac) :-
  broadcast(@Switch, InPort, SrcMac, DstMac),
  swToHst(@Switch, OutNei, OutPort), OutPort != InPort.

/* Host program */
rh1 packet(@Switch, Host, SrcMac, DstMac) :-
  initPacket(@Host, Switch, SrcMac, DstMac),
  hstToSw(@Host, Switch, OutPort).

rh2 recvPacket(@Host, SrcMac, DstMac) :-
  packet(@Host, Switch, SrcMac, DstMac),
  hstToSw(@Host, Switch, InPort).
```

Figure 30: NDlog implementation of $prog_{ESL}$

$$
\begin{aligned}
\varphi_{net_1}\quad & \mathsf{initPacket}(Host, Switch, Src, Dst) \supset \\
& \quad Host \neq Switch \wedge Host = Src \wedge \\
& \quad Host \neq Dst \wedge Switch \neq Dst. \\
\varphi_{net_2}\quad & \mathsf{ofconn}(Controller, Switch) \supset \\
& \quad Controller \neq Switch. \\
\varphi_{net_3}\quad & \mathsf{swToHst}(Switch, Host, Port) \supset \\
& \quad Switch \neq Host \wedge Switch \neq Port \wedge Host \neq Port. \\
\varphi_{net_4}\quad & \mathsf{swToHst}(Switch1, Host1, Port1) \wedge \\
& \mathsf{swToHst}(Switch2, Host2, Port2) \supset \\
& \quad (Switch1 = Switch2 \wedge Host1 = Host2 \supset \\
& \quad\quad Port1 = Port2) \wedge \\
& \quad (Switch1 = Switch2 \wedge Port1 = Port2 \supset \\
& \quad\quad Host1 = Host2).
\end{aligned}
$$

Figure 31: Network constraints for Ethernet source learning

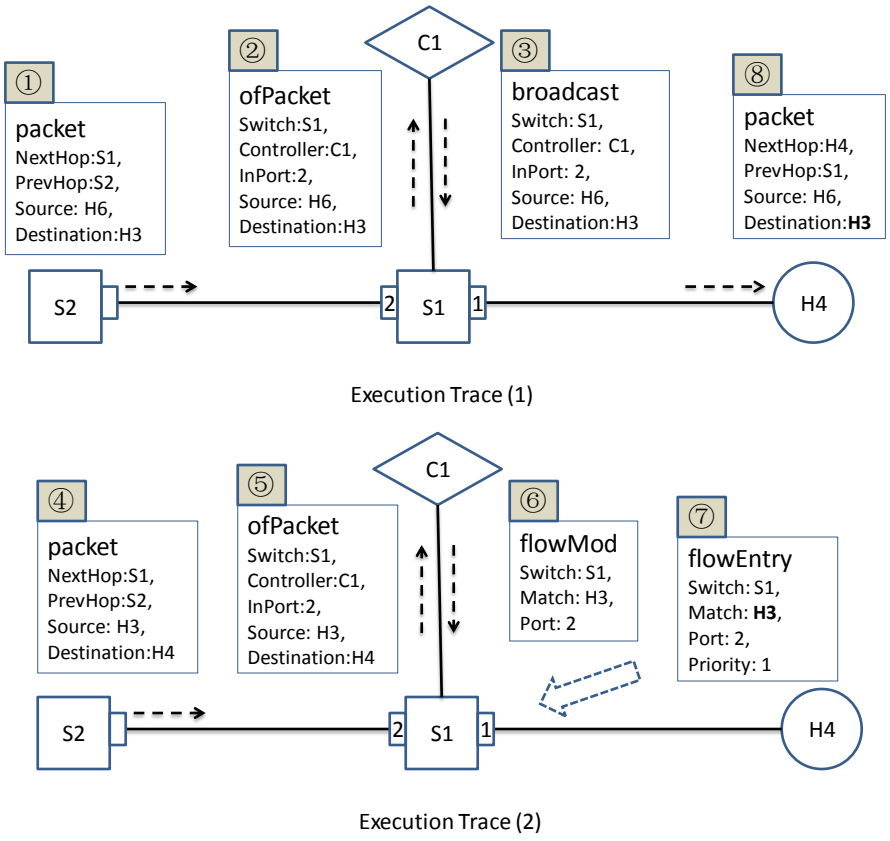


Figure 32: A counterexample for property φ_{ESL_2}

Prop	Description	Formal specification	Result
φ_{ESL_1}	If the switch has a routing entry for a host with MAC address A, it has received a packet sourced from that host in the past.	$\forall Switch, Mac, OutPort, Priority,$ $\mathsf{flowEntry}(Switch, Mac, OutPort,$ $Priority)$ $\wedge Mac = A \supset$ $\exists Nei, DstMac,$ $\mathsf{packet}(Switch, Nei, Mac, DstMac)$	true
φ_{ESL_2}	If an EndHost has received a packet that is not destined for its MAC address, the switch does not have a routing entry for that EndHost's MAC address.	$\forall EndHost, Switch, SrcMac, DstMac,$ $InPort, OPort, Outport, Mac, Priority,$ $\mathsf{packet}(EndHost, Switch, SrcMac,$ $DstMac)$ $\wedge \mathsf{swToHst}(Switch, EndHost, OPort)$ $\wedge \mathsf{flowEntry}(Switch, Mac, Outport,$ $Priority)$ $\wedge DstMac \neq EndHost \supset$ $Mac \neq DstMac$	false
φ_{ESL_3}	If EndHost has received a packet destined for it, then the switch has a flow entry for the EndHost.	$\forall EndHost, Switch, SrcMac,$ $DstMac, OPort,$ $\mathsf{packet}(EndHost, Switch, SrcMac,$ $DstMac)$ $\wedge \mathsf{swToHst}(Switch, EndHost, OPort)$ $\wedge DstMac = EndHost \supset$ $\exists Switch', Mac, Outport, Priority,$ $\mathsf{flowEntry}(Switch', Mac, Outport,$ $Priority)$ $\wedge Switch' = Switch \wedge Mac = DstMac$	false
φ_{ESL_4}	If the switch has a flowEntry for a host with mac address Mac, then there has been a flow table miss in the past for that particular host	$\forall Switch, Mac, Outport, Priority,$ $\mathsf{flowEntry}(Switch, Mac, Outport, Priority)$ $\supset$ $\exists Switch', SrcMac, DstMac, InPort,$ $Priority',$ $\mathsf{matchingPacket}(Switch', SrcMac,$ $DstMac, InPort,$ $Priority')$ $\wedge Switch' = Switch \wedge SrcMac = Mac$ $\wedge InPort = Outport \wedge Priority' = 0$	true

Table 3: Safety properties of $prog_{ESL}$ and verification results

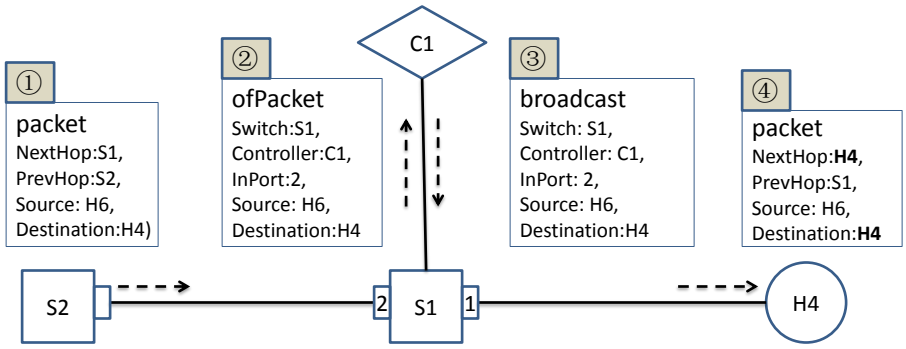


Figure 33: A counterexample for property φ_{ESL_3}

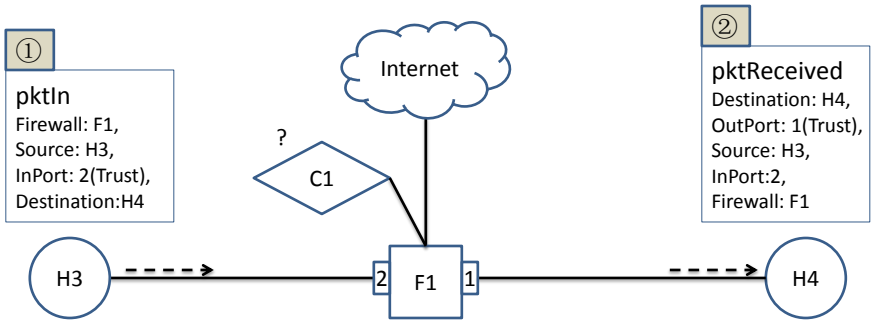


Figure 34: A counterexample for property φ_{WeakFW}

Predicate	**Description**
pktReceived(@*Dst*, *DstPort*, *Src*, *SrcPort*, *Switch*)	Host *Dst* has received a packet via the Switch through port *DstPort*, that was originally send by host *Src* through port *SrcPort*
pktIn(@*Switch*, *Src*, *SrcPort*, *Dst*)	A packet sent by host *Src* through port *SrcPort* with target host *Dst* appeared on the switch
trustedControllerMemory(@*Controller*, *Switch*, *Host*)	*Controller* stores a link between *Switch* an (untrusted) *Host*.
connection(@*Switch*, *Controller*)	There is a connection between *Switch* and *Controller*
perFlowRule(@*Switch*, *Src*, *SrcPort*, *Dst*, *DstPort*)	*Switch* stores in its memory that untrusted host *Src* is allowed to send packets to trusted host *Dst*
pktFromSwitch(@*Controller*, *Switch*, *Src*, *SrcPort*, *Dst*)	*Switch* asks *Controller* to check if untrusted host *Src* is allow to send a packet to host *Dst*

Table 4: Relations for $prog_{FW}$

```
#define TRUSTED_PORT 1
#define UNTRUSTED_PORT 2

/* (@Switch) Program
 * a packet from a trusted host via TRUSTED_PORT
 * appeared on switch without a forwarding rule
 * we know its from a trusted host since it came via
 * TRUSTED_PORT forward packet to untrusted hosts
 */
r1 pktReceived(@Dst, Uport, Src, Tport, Switch):-
  pktIn(@Switch, Src, Tport, Dst),
  Uport := UNTRUSTED_PORT,
  Tport == TRUSTED_PORT.

r2 trustedControllerMemory(@Controller,
                           Switch, Dst):-
  pktIn(@Switch, Src, Tport, Dst),
  connection(@Switch, Controller),
  Tport == TRUSTED_PORT.

/* (@Switch) Program
 * a packet from with a forwarding rule appears on
 * the switch Forward according to the rule
 * The packet may be from a trusted/untrusted source
 */
r3 pktReceived(@Dst, PortDst, Src, PortSrc, Switch):-
  pktIn(@Switch, Src, PortSrc, Dst),
  perFlowRule(@Switch, Src, PortSrc, Dst, PortDst).

/* (@Switch) Program
 * Packet from untrusted host appeared on
 * switch Send it to the controller to check
 * if it is trusted
 */
r4 pktFromSwitch(@Controller, Switch, Src,
                 Uport, Dst):-
  pktIn(@Switch, Src, Uport, Dst),
  connection(@Switch, Controller),
  Uport == UNTRUSTED_PORT.

r5 perFlowRule(@Switch, Src, Uport, Dst, Tport):-
  pktFromSwitch(@Controller, Switch, Src, Uport, Dst),
  trustedControllerMemory(@Controller, Switch, Src),
  Uport == UNTRUSTED_PORT,
  Tport := TRUSTED_PORT.
```

Figure 35: NDlog implementation of $prog_{FW}$

Rule	Summary
r1	a packet from a trusted host, with destination an untrusted host, appeared on switch without a forwarding rule. Forward the packet to the untrusted host.
r2	A packet from a trusted host appeared on switch without a forwarding rule. Insert the target host Dst of the packet into trusted controller memory.
r3	A packet from with a forwarding rule appears on the switch, which forwards it according to its flow table
r4	A packet from an untrusted host appeared on switch, which sends it to the controller to check if it can forward the packet to its intended destination
r5	Controller checks a packet originally sent by an untrusted host, found that there is a previous link between that untrusted host and the switch, and tells the switch that it can forward the packet by inserting a per flow rule into the switch for that untrusted host

Table 5: Summary of $prog_{FW}$ encoding

$$
\begin{aligned}
\varphi^{FW}_{net_1} \quad & \mathsf{connection}(Switch, Controller) \supset \\
& \quad Switch \neq Controller \\
\varphi^{FW}_{net_2} \quad & \mathsf{pktIn}(Switch, Src, SrcPort, Dst) \supset \\
& \quad Switch \neq Src \wedge Switch \neq SrcPort \\
& \quad \wedge Switch \neq Dst \wedge Src \neq SrcPort \\
& \quad \wedge Src \neq Dst \wedge SrcPort \neq Dst \\
\varphi^{FW}_{net_3} \quad & \mathsf{pktIn}(Switch1, Src1, SrcPort1, Dst1) \\
& \wedge \ \mathsf{pktIn}(Switch2, Src2, SrcPort2, Dst2) \\
& \wedge \ Switch1 \neq Switch2 \wedge Src1 = Src2 \supset \\
& \quad SrcPort1 = SrcPort2
\end{aligned}
$$

Figure 36: Network constraints for the firewall program

$$
\begin{aligned}
\varphi_{FW_1} \quad & \forall Switch, Src, SrcPort, Dst, \\
& \quad \mathsf{pktReceived}(Dst, PortDst, Src, PortSrc, Switch) \\
& \quad \wedge\ PortDst = 1 \wedge PortSrc = 2 \supset \\
& \quad \exists Controller, Host, HostPort, \\
& \quad\quad \mathsf{pktIn}(Switch, Host, HostPort, Src) \\
& \quad\quad \wedge HostPort = 1 \\
\varphi_{FW_2} \quad & \forall Switch, Src, SrcPort, Dst, DstPort, \\
& \quad \mathsf{perFlowRule}(Switch, Src, SrcPort, Dst, DstPort) \\
& \quad \wedge\ SrcPort = 2 \wedge DstPort = 1 \supset \\
& \quad \exists Host, HostPort, \\
& \quad\quad \mathsf{pktIn}(Switch, Host, HostPort, Src) \\
& \quad\quad \wedge\ HostPort = 1 \\
\varphi_{FW_3} \quad & \forall Controller, Switch, Host, \\
& \quad \mathsf{trustedControllerMemory}(Controller, Switch, Host) \supset \\
& \quad \exists Src, SrcPort, \\
& \quad\quad \mathsf{pktIn}(Switch, Src, SrcPort, Host) \\
& \quad\quad \wedge\ SrcPort = 1
\end{aligned}
$$

Figure 37: Properties for the stateful firewall

Predicate	**Description**
pktReceived(@*Dst*,*DstPort*,*Src*, *SrcPort*,*Switch*)	*Dst* has received a packet via the Switch through port *DstPort*, that was originally send by host *Src* through port *SrcPort*
pktIn(@*Switch*,*Src*,*SrcPort*,*Dst*)	A packet sent by host *Src* through port *SrcPort* with target host *Dst* appeared on the switch
trustedControllerMemory(@*Controller*, *Switch*,*Host*)	*Controller* stores a link between *Switch* and an (untrusted) *Host*.
connection(@*Switch*, *Controller*)	There is a connection between *Switch* and *Controller*
perFlowRule(@*Switch*,*Src*,*SrcPort*, *Dst*,*DstPort*)	*Switch* stores in its memory that untrusted host *Src* is allowed to send packets to trusted host *Dst*
pktFromSwitch(@*Controller*,*Switch*, *Src*,*SrcPort*,*Dst*)	*Switch* asks *Controller* to check if untrusted host *Src* is allow to send a packet to host *Dst*
link(@*Switch*,*Dst*,*PortDst*)	*Switch* is linked to *Dst* via *PortDst*

Table 6: Relations for $prog_{WeakFW}$

```
#define TRUSTED_PORT 1
#define UNTRUSTED_PORT 2

r1 pktReceived(@Dst, Uport, Src, Tport, Switch) :-
  pktIn(@Switch, Src, Tport, Dst),
  link(@Switch, Dst, Uport),
  Tport == TRUSTED_PORT.

r2 trustedControllerMemory(@Controller,
                           Switch, Dst) :-
  pktIn(@Switch, Src, Tport, Dst),
  connection(@Switch, Controller),
  Tport == TRUSTED_PORT.

r3 pktReceived(@Dst, PortDst, Src,
               PortSrc, Switch) :-
  pktIn(@Switch, Src, PortSrc, Dst),
  link(@Switch, Dst, PortDst),
  perFlowRule(@Switch, Src, PortSrc, Dst).

r4 pktFromSwitch(@Controller, Switch,
                 Src, Uport, Dst) :-
  pktIn(@Switch, Src, Uport, Dst),
  connection(@Switch, Controller),
  Uport == UNTRUSTED_PORT.

r5 perFlowRule(@Switch, Src, Uport, Dst) :-
  pktFromSwitch(@Controller, Switch, Src, Uport, Dst),
  trustedControllerMemory(@Controller, Switch, Src),
  Uport == UNTRUSTED_PORT,
  Tport := TRUSTED_PORT.
```

Figure 38: NDlog implementation of $prog_{WeakFW}$

$$
\begin{aligned}
\varphi_{net_1}^{WeakFW} \quad & \mathsf{connection}(Switch, Controller) \supset \\
& \quad Switch \neq Controller \\
\varphi_{net_2}^{WeakFW} \quad & \mathsf{pktIn}(Switch, Src, SrcPort, Dst) \supset \\
& \quad Switch \neq Src \wedge Switch \neq SrcPort \\
& \quad \wedge Switch \neq Dst \wedge Src \neq SrcPort \\
& \quad \wedge Src \neq Dst \wedge SrcPort \neq Dst \\
\varphi_{net_3}^{WeakFW} \quad & \mathsf{pktIn}(Switch1, Src1, SrcPort1, Dst1) \\
& \wedge \mathsf{pktIn}(Switch2, Src2, SrcPort2, Dst2) \\
& \wedge Switch1 \neq Switch2 \wedge Src1 = Src2 \supset \\
& \quad SrcPort1 = SrcPort2 \\
\varphi_{net_4}^{WeakFW} \quad & \mathsf{link}(Switch, Dst, PortDst) \supset \\
& \quad Switch \neq Dst \wedge Switch \neq PortDst \\
& \quad \wedge Dst \neq PortDst \\
\varphi_{net_5}^{WeakFW} \quad & \mathsf{link}(Switch1, Dst1, PortDst1) \\
& \wedge \mathsf{link}(Switch2, Dst2, PortDst2) \supset \\
& \quad (Switch1 = Switch2 \wedge Dst1 = Dst2 \\
& \quad \quad \supset PortDst1 = PortDst2) \\
& \quad \wedge (Switch1 = Switch2 \wedge PortDst1 = PortDst2 \\
& \quad \quad \supset Dst1 = Dst2)
\end{aligned}
$$

Figure 39: Network constraints for weak firewall

```
#define NUM_SERVERS 5

r1 packet(@LoadBalancer, Client, Server) :-
  initPacket(@Client, Server, LoadBalancer).

r2 hashed(@LoadBalancer, Client, ServerNum, Server) :-
    packet(@LoadBalancer, Client, Server),
  designated(@LoadBalancer, DesignatedDst),
  DesignatedDst == Server,
  Value := f_hashIp(Client),
  ServerNum := 1+f_modulo(Value, NumServers),
  NumServers := NUM_SERVERS.

r3 recvPacket(@Server, Client, ServiceAddr) :-
  hashed(@LoadBalancer, Client, ServerNum, ServiceAddr),
  serverMapping(@LoadBalancer, Server, ServerNum).

r4 recvPacket(@Server, Client, Server) :-
  packet(@LoadBalancer, Client, Server),
  designated(@LoadBalancer, DesignatedDst),
  Server != DesignatedDst,
  ServiceAddr := Server.
```

Figure 40: NDlog implementation of $prog_{LB}$

initPacket(@*Client*, *Server*, *LoadBalancer*)	*Client* sends out a packet to *LoadBalancer* with intended destination *Server*.
packet(@*LoadBalancer*, *Client*, *Server*)	*LoadBalancer* received a packet from *Client* that has destination *Server*
designated(@*LoadBalancer*, *DesignatedDst*)	For packets arriving on *LoadBalancer* with destination address *DesignatedDst*, *LoadBalancer* determines it path of a packet based on the hash value of its source address.
hashed(@*LoadBalancer*, *Client*, *ServerNum*, *Server*)	*LoadBalancer* had received a packet whose destination address matches the address that it is responsible for. *LoadBalancer* generates a hash value of the source address of *Client* to obtain an integer *ServerNum*. *ServerNum* is uniquely mapped to *Server*, to which the packet is to be routed.
serverMapping(@*LoadBalancer*, *Server*, *ServerNum*)	*LoadBalancer* stores the bijective mappings of each destination server to a unique number, *ServerNum*
recvPacket(@*Server*, *Client*, *ServiceAddr*)	*Server* has received a packet from source *Client* via *LoadBalancer*.

Table 7: Relations for $prog_{LB}$

Event	**Rule**	**Summary**
Initialize Packets	r1	A load balancer receives a packet that a client has sent out.
A packet appearing on a load balancer is destined to the load balancer's designated server	r2	A load balancer has received a packet to be sent to its designated destination. It hashes the source and uses that result modulo the number of servers to get a number corresponding to a server.
	r3	The load balancer matches the integer obtained by hashing to obtain a server to send the packet to.
Packet appearing on a load balancer is not to be sent to its designated server	r4	The load balancer forwards the packet directly to the destination as prescribed by the packet.

Table 8: Summary of $prog_{LB}$ encoding

$$
\begin{aligned}
\varphi^{LB}_{net_1} \quad & \mathsf{initPacket}(v1, v2, v3) \supset \\
& \quad v1 \neq v2 \wedge v2 \neq v3 \wedge v1 \neq v3 \\
\varphi^{LB}_{net_2} \quad & \mathsf{designated}(v4, v5) \supset \\
& \quad v4 \neq v5 \\
\varphi^{LB}_{net_3} \quad & \mathsf{designated}(v9, v10) \wedge \mathsf{designated}(v11, v12) \\
& \wedge\ v9 = v11 \supset \\
& \quad v10 = v12 \\
\varphi^{LB}_{net_4} \quad & \mathsf{serverMapping}(v6, v7, v8) \supset \\
& \quad v6 \neq v7 \wedge v7 \neq v8 \wedge v6 \neq v8 \\
\varphi^{LB}_{net_5} \quad & \mathsf{serverMapping}(v13, v14, v15) \\
& \wedge\ \mathsf{serverMapping}(v16, v17, v18) \\
& \wedge\ v13 = v16 \wedge v14 = v17 \supset \\
& \quad v15 = v18 \\
\varphi^{LB}_{net_6} \quad & \mathsf{serverMapping}(v13, v14, v15) \\
& \wedge\ \mathsf{serverMapping}(v16, v17, v18) \\
& \wedge\ v13 = v16 \wedge v15 = v18 \supset \\
& \quad v14 = v17
\end{aligned}
$$

Figure 41: Network constraints for load balancing

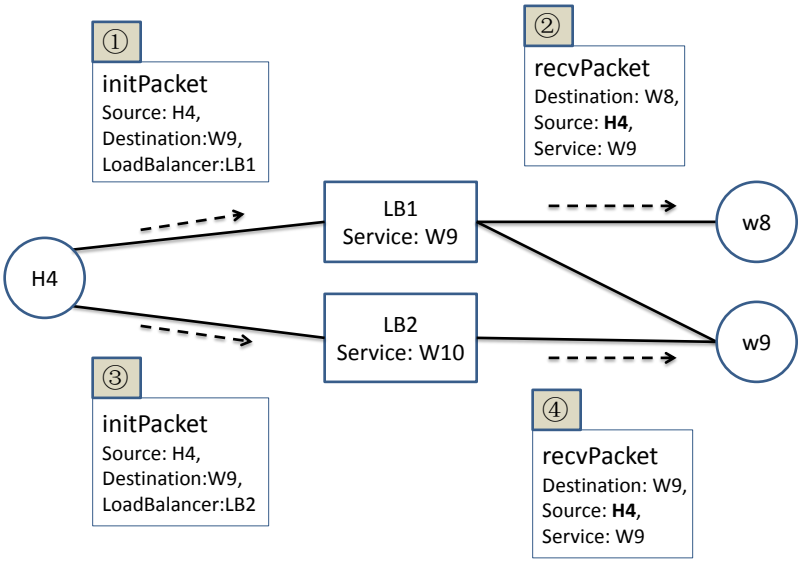


Figure 42: A counter example for property φ_{LB}

```
/* Constants */
#define BROADCAST="ff:ff:ff:ff:ff:ff", ALL_PORT=0, ARP_TYPE="ARP",
IPV4_TYPE="IPV4", CONTROLLER="controller ", ARP_REQUEST=1,
ARP_REPLY=2, ARP_PRIO=1

x/* Host program */
rh1 packet(@Switch, Host, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  linkHst(@Host, Switch, Port),
  arpRequest(@Host, SrcIp, SrcMac, DstIp, DstMac),
  Host == SrcIP, Arptype := ARP_REQUEST, DstMac == BROADCAST.

rh2 arpReply(@Host, SrcIp, SrcMac, DstIp, DstMac) :-
  linkHst(@Host, Switch, Port),
  packet(@Host, Switch, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  Arptype == ARP_REPLY, Type == ARP_TYPE, DstMac == Host.

/* Controller program */
rc1 hostPos(@Controller, SrcIp, Switch, InPort) :-
  ofconnCtl(@Controller, Switch),
  packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  Arptype == ARP_REQUEST, DstMac == BROADCAST.

rc2 arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac) :-
  packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  ofconnCtl(@Controller, Switch), Arptype == ARP_REQUEST.

rc3 arpMapping(@Controller, SrcIp, SrcMac) :-
  arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac).

rc4 arpReplyCtl(@Controller, DstIp, Mac, SrcIp, SrcMac) :-
  arpReqCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac),
  arpMapping(@Controller, DstIp, Mac).

rc5 packetOut(@Switch, Controller, Port, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  arpReplyCtl(@Controller, SrcIp, SrcMac, DstIp, DstMac),
  ofconnCtl(@Controller, Switch),
  hostPos(@Controller, DstIp, Switch, Port), Arptype := ARP_REPLY.

/*Switch program*/
rs1 packetIn(@Controller, Switch, InPort, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  ofconnSwc(@Switch, Controller),
  packet(@Switch, Host, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  linkSwc(@Switch, Host, InPort),
  flowEntry(@Switch, Arptype, Prio, Actions),
  Prio == ARP_PRIO, Actions == CONTROLLER, DstMac == BROADCAST.

rs2 packet(@Host, Switch, DstMac, DstIp, SrcMac, SrcIp, Arptype) :-
  packetOut(@Switch, Controller, OutPort, DstMac, DstIp, SrcMac, SrcIp, Arptype),
  linkSwc(@Switch, Host, OutPort), Arptype == ARP_REPLY.
```

Figure 43: NDlog implementation of $prog_{ARP}$

Predicate	Description
packet(@*Switch*, *Host*, *DstMac*, *DstIp*, *SrcMac*, *SrcIp*, *Arptype*)	*Switch* has received an ARP message of *Arptype* (Request/Reply) from *Host*. The message is from (*SrcMac*, *SrcIp*) to (*DstMac*, *DstIp*).
packetIn(@*Controller*, *Switch*, *InPort*, *DstMac*, *DstIp*, *SrcMac*, *SrcIp*, *Arptype*)	Initializes the packet above.
linkHst(@*Host*, *Switch*, *Port*)	*Host* is connected to *Switch* via *Port*
linkSwc(@*Switch*, *Host*, *InPort*)	*Switch* is connected to *Host* via *InPort*
arpRequest(@*Host*, *SrcIp*, *SrcMac*, *DstIp*, *DstMac*)	An ARP request message at *Host* of (*SrcMac*, *SrcIp*), querying the MAC address of *DstIp*.
hostPos(@*Controller*, *SrcIp*, *Switch*, *InPort*)	The controller registers the information that the host with Source IP SrcIp is connected the port InPort of Switch.
ofconnCtl(@*Controller*, *Switch*)	*Controller* has a connection to *Switch*
arpMapping(@*Controller*, *SrcIp*, *SrcMac*)	*Controller* remembers that the host of IP address *SrcIp* has the MAC address *SrcMac*.
arpReqCtl(@*Controller*, *SrcIp*, *SrcMac*, *DstIp*, *DstMac*)	An ARP request message sent from (*SrcMac*, *SrcIp*) to *Controller*, querying the MAC address of *DstIp*.
arpReplyCtl(@*Controller*, *DstIp*, *DstMac*, *SrcIp*, *SrcMac*)	An ARP reply message answering *SrcMac* of *SrcIp* to the host with IP address *DstIp* and MAC address *DstMac*,
packetOut(@*Switch*, *Controller*, *Port*, *DstMac*, *DstIp*, *SrcMac*, *SrcIp*, *Arptype*)	An OpenFlow message sent from *Controller* to *Switch*, to send an ARP packet of type Arptype from *SrcIp*, *SrcMac* to *DstIp*, *DstMac*
flowEntry(@*Switch*, *Arptype*, *Prio*, *Actions*)	A flow entry of priority Prio at *Switch* that applies Actions to packets of type *Arptype*.

Table 9: Relations for $prog_{ARP}$

$\varphi^{LB}_{net_1}$ $\mathsf{initPacket}(v1, v2, v3) \supset \quad v1 \neq v2 \wedge v2 \neq v3 \wedge v1 \neq v3$

$\varphi^{LB}_{net_2}$ $\mathsf{designated}(v4, v5) \supset v4 \neq v5$

$\varphi^{LB}_{net_3}$ $\mathsf{designated}(v9, v10) \wedge \mathsf{designated}(v11, v12) \wedge v9 = v11 \supset v10 = v12$

$\varphi^{LB}_{net_4}$ $\mathsf{serverMapping}(v6, v7, v8) \supset v6 \neq v7 \wedge v7 \neq v8 \wedge v6 \neq v8$

$\varphi^{LB}_{net_5}$ $\mathsf{serverMapping}(v13, v14, v15)$
$\wedge\ \mathsf{serverMapping}(v16, v17, v18)$
$\wedge\ v13 = v16 \wedge v14 = v17 \supset$
$v15 = v18$

$\varphi^{LB}_{net_6}$ $\mathsf{serverMapping}(v13, v14, v15)$
$\wedge\ \mathsf{serverMapping}(v16, v17, v18)$
$\wedge\ v13 = v16 \wedge v15 = v18 \supset$
$v14 = v17$

Figure 44: Network constraints for ARP

Prop	**Property description**	**Formal specification**	**Result**
φ_{ARP_1}	If any controller sends an ARP response for IP address IP_A, then some end host had sent a broadcast ARP request message for IP_A.	$\forall Ctl, IP_A, Mac_A, DstIP, DstMac,$ $\mathsf{arpReplyCtl}(Ctl, IP_A, Mac_A,$ $DstIP, DstMac) \supset$ $\exists Qmac,$ $\mathsf{arpRequest}(Host, DstIp, DstMac,$ $IP_A, Qmac)$ $\wedge\ Qmac = 255$	true
φ_{ARP_2}	If any controller has a map between IP address IP_A and MAC address Mac_A, then host A has sent a broadcast ARP request.	$\forall Ctl, IP_A, Mac_A,$ $\mathsf{arpMapping}(Ctl, IP_A, Mac_A) \supset$ $\exists Host, SrcIP, SrcMac,$ $DstIP, DstMac,$ $\mathsf{arpReply}(Host, IP_A, Mac_A,$ $DstIp, DstMac)$ $\wedge\ DstMac = 255$	true

Table 10: Results of checking safety properties of $prog_{ARP}$ on our tool

CHAPTER 5

Runtime Analysis with Compressed Provenance

The previous two chapters mainly focus on using formal methods to verify properties of distributed systems. In reality, formal verification cannot guarantee that a system would run without failure. One reason is that only the verified properties are expected to hold during execution, and due to the complexity of applying formal methods to system verification, such properties are only a fragment of the desirable property space. Another reason is attributed to the runtime failure – e.g., hardware errors and power outage – that could not be anticipated during verification stage. The above discussion motivates the need for supporting runtime diagnostics in distributed systems, by which the network administrator could identify the root cause of any failure that happens during system execution.

In this part of the dissertation, STRANDS provides runtime analysis of distributed systems by introducing storage-optimized network provenance, which allows the user to issue queries over compressed network meta-data about the history execution of a distributed system.

In recent years, network provenance has been successfully applied to various network settings, resulting in proposals for distributed provenance [95], secure network provenance [92], distributed time-aware provenance [94], negative provenance [85], differential provenance [16]. These proposals demonstrate that database-style declarative queries can be used for maintaining and querying distributed provenance at scale. Moreover, they are useful for a wide range of forensic analysis for determining root causes of misconfigurations, errors, attacks, and have even been shown to allow automated repair of network configurations [84].

One of the main drawbacks of the existing techniques is their potentially significant storage overhead, when network provenance is incrementally maintained as network events occur continuously. This is particularly challenging for the *data plane* of networks that deals with frequent and high-volume incoming data packets. When there are streams of incoming

packet events, the provenance information can become prohibitively large. While there is prior work on the storage efficiency of provenance maintenance in the literature [15][46][94], there is no solution that achieves significant storage reduction in a distributed scenario with both low network overhead and low query latency. Our contributions are:

System Model. We propose a new network programming model, which specifies *distributed event-based linear programs* (DELPs), a restricted variant of NDLog programs. Each DELP is composed of a set of rules triggered by events, and executes until a fixpoint is reached. Unlike traditional event-condition-action rules, a DELP has the option of designating *slow changing tuples*, which do not change while a distributed fixpoint computation is happening, but are still amenable to update at intervals. An example of a slow changing tuple could be a routing entry in a router. We show, through two example applications (packet forwarding and DNS resolution), that this model is general enough to cover a wide range of network applications.

Distributed Provenance Compression. Based on the DELP model, we propose two techniques to store provenance information efficiently. Our second technique combines multiple provenance trees together, based on a notion of *equivalence classes*. In each equivalence class, provenance trees are identical except for a few pre-defined nodes. We compress these equivalent trees by maintaining only one concrete copy for the shared part, along with the delta information for each individual provenance tree. We also propose to efficiently identify equivalence between provenance trees by simply inspecting the values of input events' attributes, thus reducing the computation overhead in a distributed environment.

Implementation and Evaluation. We implement a prototype of our distributed compression scheme based on the RapidNet declarative networking engine [59]. We enhance RapidNet to include a rule rewrite engine that maintains provenance at runtime. Provenance queries are implemented as distributed recursive queries over the maintained provenance information. We deploy and evaluate our prototype using two popular network applications

$r1$ packet$(@N, S, D, DT)$:− packet$(@L, S, D, DT)$,
route$(@L, D, N)$.
$r2$ recv$(@L, S, D, DT)$:− packet$(@L, S, D, DT), D == L$.

Figure 45: An NDlog program for packet forwarding

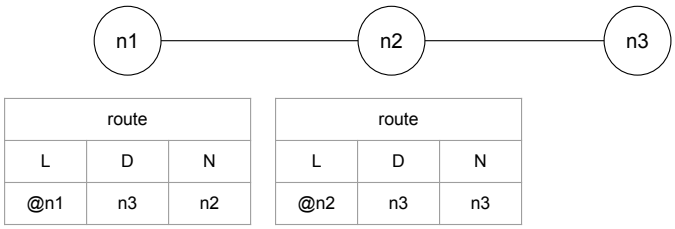


Figure 46: An example deployment of packet forwarding. Node $n1$ and node $n2$ has a local *route* table indicating routes towards node $n3$.

– i.e., packet forwarding and DNS resolution – and the performance results show that the compression techniques achieve comparably low storage demand as well as lower query latency compared to alternative solutions, with only negligible network overhead added to each monitored network application at runtime.

We use an example query of NDlog (Figure 45) and its deployment in a topology of three nodes (Figure 46) to help illustrate our design. In the example program of Figure 45, *r1* forwards a local packet (packet) to neighbor N by looking up the packet's destination D in the local routing table (route). *r2* receives a packet and stores it locally in recv table, if the packet is destined to the local node ($D == L$). In Figure 46, Nodes *n1* and *n2* both have a route table, storing the next hop – i.e., *n2* for *n1*, and *n3* for *n2* – to reach *n3*.

5.1. Background

We first introduce the concept of distributed network provenance, and shows how it is maintained in a typical Internet-scale network provenance engine ExSPAN [95].

5.1.1. Distributed Network Provenance

Data provenance [40] can be used to explain why and how a given tuple is derived. Based on data provenance, prior work [96] also proposes network provenance, which faithfully records the execution of (possibly erroneous) applications in a (possibly misconfigured) distributed system. This allows the network administrators to inspect the derivation history of system states. For example, suppose there is a direct link between $n1$ and $n3$ in Figure 46. If the user prefers the routing with the shortest paths, the routing entry of $n1$ in Figure 46 would have been erroneous – a correct entry should be route$(@n1, n3, n3)$. The provenance engine, agnostic of this error, would record the packet traversal on the path $n1 \rightarrow n2 \rightarrow n3$. The user can later use this recorded provenance as explanation on why the packet took a particular route, eventually leading to further investigation into the route table at $n1$.

Network provenance is typically represented as a directed tree rooted at the queried tuple. Figure 47 shows the provenance tree of a tuple recv$(@n3, n1, n3,$ "data"). This provenance tree records the traversal of packet$(@n1, n1, n3,$ "data") from node $n1$ to $n3$ in Figure 46. There are two types of nodes in a typical provenance tree: the rule nodes and the tuple nodes. The rule nodes (i.e., the oval nodes in Figure 47) stand for the rules that are triggered in the program execution, while the tuple nodes (i.e., the square nodes in Figure 47) represent tuples that trigger/are derived by the rule execution. Note that the root of a provenance tree is always a tuple node that represents the queried tuple.

To maintain the provenance, traditional database work [46] often stores data provenance along with the target tuple for efficient provenance querying. Such centralized provenance maintenance turns out to be very costly for network provenance – which is typically constructed in a distributed fashion – in terms of the extra bandwidth needed to ship the provenance information.

ExSPAN [96], a representative distributed provenance engine, maintains the provenance information in a distributed relational database. There are two (distributed) tables in the

prov			
Loc	**VID**	**RID**	**RLoc**
n3	**vid6** (sha1(recv(@$n3, n1, n3,$ "*data*")))	**rid3**	n3
n3	**vid5** (sha1(packet(@$n3, n1, n3,$ "*data*")))	**rid2**	n2
n2	**vid4** (sha1(packet(@$n2, n1, n3,$ "*data*")))	**rid1**	n1
n2	**vid3** (sha1(route(@$n2, n3, n3$)))	NULL	NULL
n1	**vid2** (sha1(packet(@$n1, n1, n3,$ "*data*")))	NULL	NULL
n1	**vid1** (sha1(route(@$n1, n3, n2$)))	NULL	NULL

ruleExec			
RLoc	**RID**	**R**	**VIDS**
n3	**rid3**(sha1($r2$+n3+**vid5**))	$r2$	(**vid5**)
n2	**rid2**(sha1($r1$+n2+**vid3**+**vid4**))	$r1$	(**vid3**,**vid4**)
n1	**rid1**(sha1($r1$+n1+**vid1**+**vid2**))	$r1$	(**vid1**,**vid2**)

Table 11: Relational tables (ruleExec and prov) maintaining the provenance tree in Figure 47.

database: a *prov* table and a *ruleExec* table. The *prov* table records the rule triggering of a derived tuple, while the *ruleExec* table maintains the body tuples triggering a specific rule. Table 11 shows an example distributed database storing the provenance tree in Figure 47. The **Loc** attribute in the *prov* table and the **RLoc** attribute in the *ruleExec* table indicate the location of each tuple.

ExSPAN uses a recursive query to retrieve the provenance tree of a queried tuple. For example, to query the provenance tree of recv(@$n3, n1, n3,$ "*data*") (Figure 47), ExSPAN first computes the hash value $vid6$ of the tuple, and uses $vid6$ to find the tuple prov($n3, vid6, rid3, n3$) in the prov table. ExSPAN further uses $rid3$ and $n3$ to locate ruleExec($n3, rid3, r2, (vid5)$) in the ruleExec table, which represents the provenance node of the rule execution (i.e., $r2$) that derives $vid6$. To further query the body tuples that triggered $r2$, the querier would then look up ($vid5$) in the prov table. This recursive querying continues until it reaches the base tuples (e.g., route(@$n1, n3, n2$)).

We adopt the same storage model as ExSPAN. However, our provenance compression scheme applies generally to any distributed provenance model.

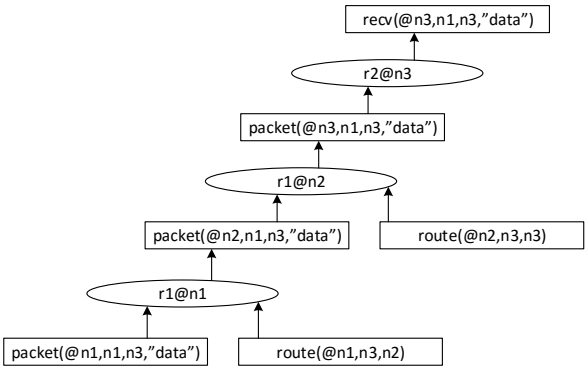


Figure 47: A (distributed) provenance tree for execution of packet($@n1, n1, n3,$ "*data*"), which traversed from node $n1$ to node $n3$ in Figure 46.

5.1.2. Motivation for Effective Provenance Compression

A key problem not addressed in prior work on network provenance [95][94] is to reduce storage effectively while retaining efficiency of the querying process. Provenance information in a typical network can become very large, especially for distributed applications (e.g., network protocols) where event tuples trigger rules in a streaming fashion. For example, in Figure 46, if *n1* initiates a large volume of traffic towards *n3*, each packet in the traffic would generate a provenance tree similar to the one in Figure 47. Given that today's routers forward over millions of packets per second, this would incur prohibitively high storage overhead for the maintenance of distributed provenance on each intermediate node.

On the other hand, querying efficiency of provenance is important as well. Failure in a company's cloud network, for example, could cost millions of dollars and quick provenance retrieval is essential to root-cause analysis of network anomaly. Therefore, though there are existing solutions that could achieve significant low storage overhead, such as general content-level compression (e.g., gzip) or replay-based reactive provenance maintenance [94], new approaches need to be developed to allow for more efficient querying process.

We observe however that the provenance of different packets share significant similarities

in their structures, presenting opportunities for provenance compression across different provenance trees. For example, in Figure 46, whenever a new packet is sent from $n1$ to $n3$, an entire provenance tree is created and maintained. However, it is not hard to observe that all the packets traversing through $n1$ and $n2$ take the same route – that is, they join with the same local route tuples. Therefore, storage of the provenance trees generated by these packets could be significantly reduced if we manage to remove the observed redundancy.

5.1.3. Challenges and Requirements

The key challenge of provenance compression in a distributed system is to achieve high storage saving while incurring low network overhead. More specifically, our compression strategy aims to achieve the following three goals:

- **Efficient querying of provenance.** A user should be able to query provenance information efficiently regardless of how provenance is stored. We hence avoid compression techniques that focus on content-level compression (e.g. gzip) – as such techniques would require the user to decompress the whole provenance information each time, even if the user is only interested in the provenance of one event – and opt for conservative compression that preserves the structure of the provenance trees.

- **Compression should be effective.** Our compression approach should ensure that significant reduction in provenance storage overhead.

- **Compression should have low network overhead.** Unlike centralized environment, a distributed system has limited network resources. Therefore, the compression technique is expected to have low impact on normal network operations – e.g., incur low bandwidth overhead.

5.2. Model

Following our background, we next introduce our system model, which includes modeling of distributed systems, network applications, and provenance information.

A distributed system DS is modeled as an undirected graph $G = (V, E)$. Each node N_i in V represents an entity in DS. Two nodes N_i and N_j can communicate with each other if and only if there is an edge (N_i, N_j) in E. In DS, each node N_i maintains a local state in the form of a relational database DB_i. Tables in DB_i can be divided into *base tables* and *derived tables*. Tuples in *base tables* are manually updated, while tuples in *derived tables* are derived by network applications. Figure 46 is an example distributed system with three nodes.

5.2.1. Network Applications

Each node in DS runs a number of network applications, which are specified in NDlog with syntactic restriction. The syntactic restriction enables efficient provenance compression (Section 5.4), while still being expressive enough to model most network applications. In particular, we have:

Definition 1. *An NDlog program $Prog$=$\{r_1, r_2, ..., r_n\}$ is a distributed event-driven linear program (DELP), if $Prog$ satisfies the following three conditions:*

- Each rule is event-driven. *Each rule r_i is in the form: $[head] :- [event], [conditions]$, where $[event]$ is a body relation designated by the programmer, and $[conditions]$ are all non-event body atoms.*

- Consecutive rules are dependent. *For each rule pair (r_i, r_{i+1}) in $Prog$, the head relation of r_i is identical to the event relation in r_{i+1}.*

- Head relations only appear as the event relations in rule bodies. *For each head relation hd in any rule r_i, there does not exist a rule r_j, such that hd is a non-event relation in r_j.*

In a typical network application, non-event relations often represent network states, which change slowly compared to the fast rate of incoming events. For example, in packet forwarding, the route relation is either updated manually or through a network routing protocol. In either case, it changes slowly compared to the large volume of incoming packets. We

call such non-event relations in a DELP as *slow-changing* relations, and assume they do not change during the fixpoint computation. This assumption is realistic and can be enforced easily in the networks where configuration is updated at runtime and packets see only either the old or new configuration version across routers, as shown in prior work [71] in the networking community.

A DELP $\{r_1, r_2, ..., r_n\}$ can be deployed in a distributed fashion over a network, and its execution follows the pipelined semi-naïve evaluation strategy introduced in prior work [54] – whenever a new event tuple is injected into a node N_i, it triggers r_1 by joining with the slow-changing tuples at N_i. The generated head tuple hd is then sent to a node N_j as identified by the location specifier of hd – triggering r_2 at N_j. This process continues until r_n is executed.

DELP can model a large number of network applications, due to their event-driven nature, such as packet forwarding (Figure 45), Domain Name System (DNS) resolution [58], Dynamic Host Configuration Protocol (DHCP) [28] and Address Resolution Protocol (ARP) [68].

5.2.2. Provenance of Interest

It is often the case that network administrators use a subset of network states more often than others as their starting point for debugging. In the packet forwarding example, an administrator is more likely to query the provenance of a recv tuple rather than a packet tuple upon packet misrouting, because the nodes that generate recv tuples are usually the first places where an administrator observes abnormality. Therefore, we allow a user to specify *relations of interest* – i.e., relations whose provenance information interests a user the most in a network application – and our runtime system maintains *concrete* provenance information only for those tuples of the relations of interest. However, the provenance of other tuples – i.e., those of the relations of less interest – is still accessible. We can adopt, for example, the reactive maintenance strategy proposed in DTaP [94], by only maintaining non-deterministic input tuples, and replaying the whole system execution to re-construct

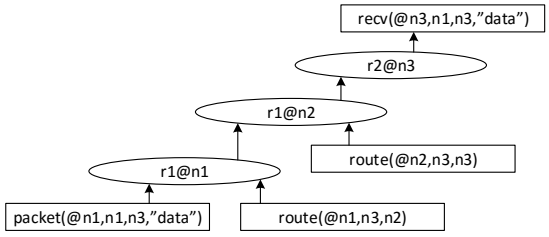


Figure 48: An optimized provenance tree for the tree in Figure 47.

the provenance information of the tuples of less interest during querying.

As with prior work [96], we represent the provenance information of the tuples of interest as provenance trees. The only difference is that, given the syntactic restriction of a DELP, our system treats slow-changing tuples as base tuples during provenance querying – i.e., the provenance tree of a slow-changing tuple, e.g., a route tuple, is not automatically presented, even if the tuple could be derived from another network application, e.g., a routing protocol. To obtain the provenance tree of a route tuple during provenance querying, a user could specify route as a relation of interest in the application that derives it, and explicitly query the provenance tree of the route tuple in a separate process.

5.3. Basic Storage Optimization

Based on the model introduced in the previous section, we propose our basic storage optimization for provenance trees, which lays the foundation for the compression scheme in Section 5.4. Simply put, for each provenance tree, we remove its provenance nodes representing the intermediate event tuples. Figure 48 shows an optimized provenance tree tr' of the tree in Figure 47. The (distributed) relational database maintaining tr' is shown in Table 12, where *vid* values and *rid* values are identical to those in Table 11.

Compared to Table 11, Table 12 differs at two parts:

- The prov table only maintains the provenance of the queried tuple, i.e., the *recv* tuple. Other entries in the prov table are omitted because they represent either the removed

prov			
Loc	**VID**	**RID**	**RLoc**
n3	**vid6**	**rid3**	n3

ruleExec					
RLoc	**RID**	**R**	**VIDS**	**NLoc**	**NRID**
n3	**rid3**	$r2$	NULL	n2	**rid2**
n2	**rid2**	$r1$	(**vid4**)	n1	**rid1**
n1	**rid1**	$r1$	(**vid1**,**vid2**)	NULL	NULL

Table 12: Optimized ruleExec and prov tables for the provenance tree in Figure 48.

intermediate tuples or the base tuples.

- Two extra columns **NLoc** and **NRID** are added to the *ruleExec* table. These two attributes help recursive queries find the child node for each provenance node.

The optimization of removing the intermediate nodes saves a fair amount of storage space, especially when the input events arrive at a high rate and generate a large number of intermediate tuples, as is common in typical networking scenarios. We use the querying of recv($@n3, n1, n3,$ "*data*")'s provenance in Table 12 to illustrate the two-step provenance querying process for optimized provenance trees:

Step 1: Construct the optimized provenance tree. The query first fetches the provenance tree in the optimized form through recursive querying over Table 12. Starting from the prov entry corresponding to recv($@n3, n1, n3,$ "*data*"), we fetch the provenance node for the last rule execution $rid3$ in the ruleExec table, then follow the values in **NLoc** and **NRID** to recursively fetch all the ruleExec tuples (i.e., $rid3$, $rid2$ and $rid1$) until no further provenance nodes can be fetched – i,e., both **NLoc** and **NRID** are $NULL$.

Step 2: Compute the intermediate provenance nodes. At the end of Step 1, we obtain the provenance tree tr' in Figure 48. To recover the intermediate provenance nodes, we start from the leaf nodes, i.e., packet($@n1, n1, n3,$ "*data*") and route($@n1, n3, n2$), and re-execute the rule $r1$ to derive packet($@n2, n1, n3,$ "*data*"). This process is repeated in a bottom-up fashion until the root is reached, resulting in the provenance tree in Figure 47 .

In summary, the basic optimization still allows the user to query the complete provenance trees, but incurs extra computational overhead during provenance querying to recover the intermediate nodes. The extra query latency is negligible, as is shown in Section 5.6.1.

5.4. Equivalence-based Compression

The storage optimization described in Section 5.3 focuses on reducing the storage overhead *within* a single provenance tree. Building upon this optimization, we further explore removing redundancy *across* provenance trees. We propose grouping provenance trees of DELP execution into equivalence classes, and only maintaining one copy of the shared sub-tree within each equivalence class. Our definition of the equivalence relation allows equivalent provenance trees to be quickly identified through inspection of equivalence keys – a subset of attributes of input event tuples – and compressed efficiently at runtime. The equivalence keys can be obtained through static analysis of a DELP.

5.4.1. Equivalence Relation

We first introduce the equivalence relation for provenance trees. We say that two provenance trees tr and tr' are equivalent, written ($tr \sim tr'$) if (1) they are structurally identical – i.e., they share the identical sequence of rules – and (2) the slow-changing tuples used in each rule are identical as well. In other words, two equivalent trees tr and tr' only differ at two nodes: (1) the root node that represents the output tuple and (2) the input event tuple.

More formally, We define tree equivalence using the following notations: an instance of the input event relation e is denoted $e(@\iota, \vec{c})$, or ev in shorthand; an instance of a slow-changing relation b is denoted as $b(@\iota, \vec{c})$ or B (thus we write $B_1::\cdots::B_n$ to denote the slow-changing tuples used to execute rule rID); and instances of fast-changing relations are denoted by P, $p(@\iota, \vec{c})$, Q, or $q(@\iota, \vec{c})$. A provenance tree tr is inductively defined as follows:

$$\begin{aligned} \textit{Provenance tree } tr \quad ::= \quad & \langle rID, P, ev, B_1::\cdots::B_n\rangle \\ & |\langle rID, P, tr, B_1::\cdots::B_n\rangle \end{aligned}$$

$tr \sim_K tr'$ is defined inductively as follows:

$$\frac{ev \sim_K ev'}{v\langle rID, P, ev, B_1::\cdots::B_n\rangle \sim_K \langle rID, P', ev', B_1::\cdots::B_n\rangle}$$

$$\frac{tr \sim_K tr'}{\langle rID, P, tr, B_1::\cdots::B_n\rangle \sim_K \langle rID, P', tr', B_1::\cdots::B_n\rangle}$$

In our packet forwarding example, the provenance tree generated by a new incoming event packet($@n1, n1, n3,$ "url") (with "url" as its payload) is equivalent to the tree in Figure 48.

For each equivalence class, we only need to maintain one copy for the sub-provenance tree shared by all the class members, while each individual tree in the equivalence class only needs to maintain a small amount of delta information – i.e., the root node, the event leaf node, and a reference to the shared sub-provenance tree. Additionally, this definition of equivalence enables more efficient equivalence detection than node-by-node comparison between trees. In fact, we show that equivalence of two provenance trees can be determined by checking equivalence of the input event tuples in both trees, based on the observation that the execution of a DELP is uniquely determined by the values of a subset of attributes in the input event tuple. For example, in the packet forwarding program (Figure 45), if the values of the attributes (loc, dst) in two input packet tuples are identical, these two tuples will generate equivalent provenance trees.

We denote the minimal set of attributes K in the input event relation whose values determine the provenance trees as *equivalence keys*. Two event tuples ev_1 and ev_2 of a relation e are said to be *equivalent w.r.t* K, written as $ev_1 \sim_K ev_2$, if their valuation of K is equal. Formally:

Definition 2 (Event equivalence)**.** *Let* $K = \{e{:}i_1, \cdots, e{:}i_m\}$, $e(t_1 \cdots t_n) \sim_K e(s_1 \cdots s_n)$ *iff* $\forall j \in \{i_1, \cdots, i_m\}$, $t_j = s_j$.

Here, $e{:}i$ denotes the i^{th} attribute of the relation e.

Based on the above discussion, our approach to compressing provenance trees, with regard to a program *DQ*, consists of the following two main algorithms. (1) an equivalence keys identification algorithm, which performs static analysis of *DQ* to compute the equivalence keys (Section 5.4.2); and (2) an online provenance compression algorithm, which maintains the shared provenance tree for each equivalence class in a distributed fashion (Section 5.4.3).

Correctness of using event equivalence for determining provenance tree equivalence is shown in Theorem 8. The proof will be discussed in Section 5.4.2.

Theorem 8 (Correctness of equivalence keys)**.** *Given a program DQ of DELP, and two input event tuples* ev_1 *and* ev_2*, if* $ev_1 \sim_K ev_2$*, where K is the equivalence keys for DQ, then for any provenance tree* tr_1 *(*tr_2'*) generated by* ev_1 *(*ev_2*), there exists a provenance tree* tr_2 *(*tr_1'*) generated by* ev_2 *(*ev_1*) s.t.* $tr_1 \sim tr_2$ *(*$tr_1' \sim tr_2'$*).*

5.4.2. Equivalence Keys Identification

Given a DELP, we define a static analysis algorithm to identify the equivalence keys of the input event relation. The algorithm consists of two steps: (1) building an attribute-level dependency graph reflecting the relationship between valuation of different attributes and (2) computing equivalence keys based on the constructed dependency graph. Details of each step are given below.

Build the attribute-level dependency graph. An attribute-level dependency graph $G{=}(V, E)$ is an undirected graph. Nodes of G represent attributes in the program. Specifically, the i-th attribute of a relation *rel* corresponds to a vertex labeled as $(rel{:}i)$ in G. Figure 49 shows an example attribute-level dependency graph for the packet forwarding program in Figure 45. Based on Section 5.4.2, the equivalence keys are (packet:0, packet:2).

Two vertices $v1$ and $v2$ are directly connected in G if and only if $v1$ represents an attribute $attr_1$ of the event relation in a rule r and $v2$ represents another attribute $attr_2$ in r, and

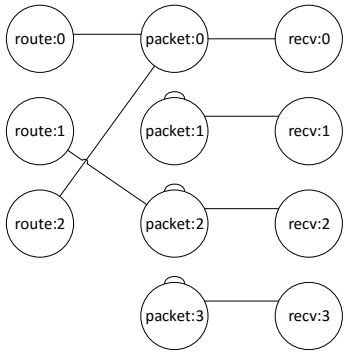


Figure 49: The attribute-level dependency graph for the packet forwarding program in Figure 45.

satisfies any of the following conditions: (1) $attr_2$ is an attribute of the same name as $attr_1$ in a slow-changing relation (e.g., $v1$ = (packet:1) and $v2$ = (route:1) in rule $r1$ of Figure 45); (2) $attr_2$ is a head attribute with the same name as $attr_1$ (e.g., $v1$ = (packet:1) and $v2$ = (recv:1) in $r2$ of Figure 45); (3) $attr_2$ and $attr_1$ appear in the same arithmetic atom (e.g., $v1$ = (packet:0) and $v2$ = (packet:2) in rule $r2$ of Figure 45); and (4) $v1$ is on the right hand side of an assignment asn and $attr_2$ is on the left hand side of asn. (e.g., if rule $r2$ of Figure 45 were to be redefined as $r2'$ $\mathsf{recv}(@L, S, N, DT) :- \mathsf{packet}(@L, S, D, DT), N := L + 2.$, and $v1$ = (packet:0) while $v2$ = (recv:2)).

Identify equivalence keys. Given the attribute-level dependency graph G, we identify the equivalence keys of the input event relation ev using the function GetEquiKeys (Figure 50). GetEquiKeys takes G and ev as input, and outputs a list of attributes $eqid$ representing the equivalence keys. In the algorithm, for each node (ev:i) in G, GetEquiKeys checks whether (ev:i) is reachable to any attribute in a slow-changing relation. If this is the case, (ev:i) would be identified as a member of the equivalence keys, and appended to $eqid$. We always include the attribute indicating the input location of ev (e.g., (packet:0)) in the equivalence keys, to ensure no two input event tuples at different locations have the same equivalence keys. When applied to the packet forwarding program, GetEquiKeys would identify (packet:0) and (packet:2) as equivalence keys.

Now we introduce a few denotations to help prove Theorem 8. We use predicate $\mathsf{joinSAttr}(p{:}n)$

```
1: function GetEquiKeys(G, ev)
2:     eqid ← {}
3:     eqid.append(ev:0)
4:     nodes ← event attribute nodes in G
5:     for each ev:i in nodes do
6:         for bnode in non-event nodes of G do
7:             if ev:i is reachable to bnode then
8:                 eqid.append(ev:i)
9:     return eqid
10: end function
```

Figure 50: Pseudocode to identify equivalence keys

to denote that a node $(p{:}n)$ in the dependency graph has an edge to an attribute in a slow changing relation. We denote each edge connecting two attributes $(p{:}n, q{:}m)$ not in any slow-changing relation as predicate $\mathsf{joinFAttr}(p{:}n, q{:}m)$. We further use predicate $\mathsf{joinFAttr}(p{:}n, q{:}m)$ to inductively define $\mathsf{connected}(e{:}i, p{:}n)$, denoting a path in the graph from $(e{:}i)$ to $(p{:}n)$. We then formally define below what it means, given a DELP, for K to be equivalence keys:

Definition 3. *K is equivalence keys for a program DQ of DELP, if $\forall(e{:}i) \in K$, either $DQ \vdash \mathsf{joinSAttr}(e{:}i)$ or $\exists p, n$ s.t. $DQ \vdash \mathsf{connected}(e{:}i, p{:}n)$ and $DQ \vdash \mathsf{joinSAttr}(p{:}n)$.*

We show the correctness of Theorem 8 by proving Lemma 9, a stronger lemma that gives us Theorem 8 as corollary. In Lemma 9, we write $tr : P$ to denote that tr is a provenance tree of the output tuple P, and write $prog, \mathcal{DB}, ev \vDash tr : P$ to mean that tr is generated by executing the program $prog$ over a database $\mathcal{DB}$, triggered by the event tuple ev.

Lemma 9 (Correctness of equivalence keys (Strong))**.**

If $\textsc{GetEquiKeys}(G, ev) = K$ *and* $ev_1 \sim_K ev_2$

and $prog, \mathcal{DB}, ev_1 \vDash tr_1 : p(t_1, ..., t_n)$,

then $\exists tr_2 : p(s_1, ..., s_n)$ *s.t.* $prog, \mathcal{DB}, ev_2 \vDash tr_2 : p(s_1, ..., s_n)$

and $tr_1 : p(t_1, ..., t_n) \sim tr_2 : p(s_1, ..., s_n)$

and $\forall i \in [1, n]$, $t_i \neq s_i$ *implies*

$\exists \ell$ *s.t.* $prog \vdash \mathsf{connected}(ev{:}\ell, p{:}i)$ *and* $(ev{:}\ell) \notin K$.

Intuitively, Lemma 9 states that given two equivalent input event tuples ev_1 and ev_2 w.r.t. K, and ev_1 generates a provenance tree tr_1, we can construct a tr_2 for ev_2 such that tr_1 and tr_2 are equivalent – i.e., they share the same structure and slow-changing tuples. Furthermore, if the two output tuples $p(t_1, ..., t_n)$ and $p(s_1, ..., s_n)$ have different values for a given attribute, this attribute must connect to an event attribute that is not in equivalence keys in the dependency graph. This last condition enables an inductive proof of Lemma 9 over the structure of the trees. More details could be found in the technical report [21].

Time complexity. Next, we analyze the time complexity of static analysis. Assume that a program DQ has m rules. Each rule r has k atoms, including the head relation and all body atoms. Each atom has at most t attributes. Hence, the attribute-level dependency graph G has at most $n{=}m * k * t$ nodes. The construction of G takes $O(n^2)$ time, and the identification of equivalence keys takes $O(t * n)$ time. Normally t is much smaller than n. Therefore, the total complexity of static analysis is $O(n^2)$.

5.4.3. Online Provenance Compression

We next present an online provenance compression scheme that compresses equivalent (distributed) provenance trees based on the identified equivalence keys. In our compression scheme, execution of a DELP, triggered by an event tuple ev, is composed of three stages:

- **Stage 1: Equivalence keys checking.** Extract the value v of ev's equivalence keys, and check whether v has ever been seen before. If so, set a Boolean flag $existFlag$ to $True$. Otherwise, set $existFlag$ to $False$. Tag $existFlag$ along with ev throughout the execution.

- **Stage 2: Online provenance maintenance.** If $existFlag$ is $True$, no provenance information is maintained during the execution. Otherwise, the complete provenance tree of the execution would be maintained.

- **Stage 3: Output tuple provenance maintenance.** When the execution finishes, associate the output tuple to the shared provenance tree to allow for future provenance

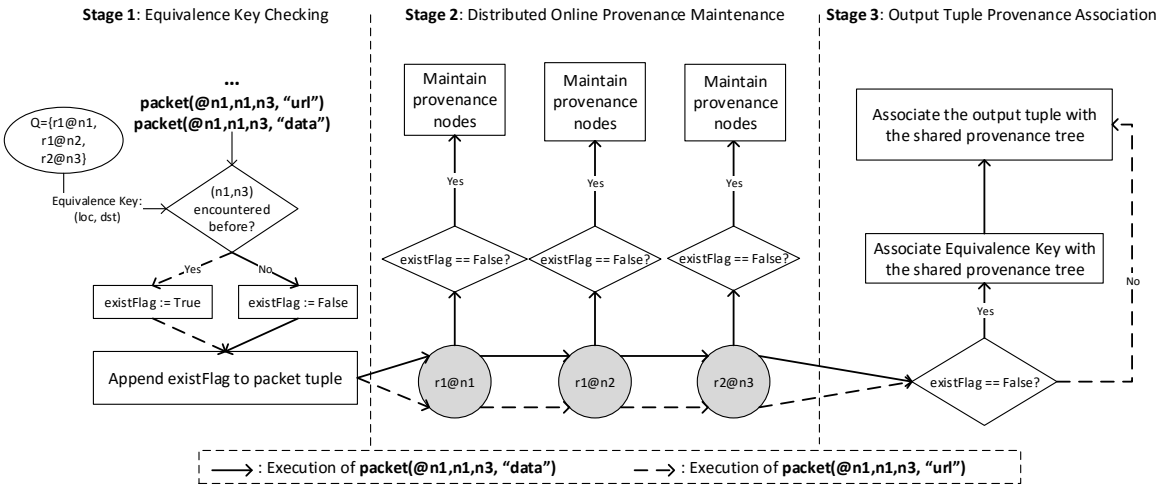


Figure 51: An example execution of the packet forwarding program. The program is first triggered by packet($@n1, n1, n3,$ "data"), followed by packet($@n1, n1, n3,$ "url").

querying.

To illustrate this, Figure 51 presents an example consisting of two packets traversing the network topology (from *n1* to *n3*) in Figure 46. packet($@n1, n1, n3,$ "data") is first inserted for execution (represented by the solid arrows), followed by the execution of packet($@n1, n1, n3,$ "url") (represented by the dashed arrows). The three stages of online compression are logically separated with vertical dashed lines. Table 13 presents the (distributed) relational tables (i.e., a ruleExec table and a prov table) that maintain the compressed provenance trees for the aforementioned execution. Next, we introduce each stage in detail.

Equivalence Keys Checking. Upon receiving an input event *ev*, our runtime system first checks whether the value of *ev*'s equivalence keys have been seen before. To do this, we use a hash table *htequi* to store all unique equivalence keys that have arrived. If *ev*'s equivalence keys *eqid* has a value that already exists in *htequi*, a Boolean flag $existFlag$ will be created and set to $True$. This $existFlag$ is supposed to accompany *ev* throughout the

ruleExec					
Loc	**RID**	**RULE**	**VIDS**	**NLoc**	**NRID**
n3	**rid1**(sha1(r2))	r2	NULL	n2	**rid2**
n2	**rid2**(sha1(r1,*vid*1))	r1	(*vid*1(sha1(route(@$n2$, $n3, n3$))))	n1	**rid3**
n1	**rid3**(sha1(r1,*vid*2))	r1	(*vid*2(sha1(route(@$n1$, $n3, n2$))))	NULL	NULL

prov				
Loc	**VID**	**RLoc**	**RID**	**EVID**
n3	**tid1**(sha1(recv(@$n3, n1$, $n3$, "data")))	n3	**rid1**	**evid1**(sha1(packet(@$n1$, $n1, n3$, "data")))
n3	**tid2**(sha1(recv(@$n3, n1$, $n3$, "url")))	n3	**rid1**	**evid2**(sha1(packet(@$n1$, $n1, n3$, "url")))

Table 13: a ruleExec table and a prov table for compressed provenance trees produced in Figure 51

execution, notifying all nodes involved in the execution to avoid maintaining the concrete provenance tree. Otherwise, $existFlag$ would be set to $False$, instructing the subsequent nodes to maintain the provenance tree. For example, in Figure 51, when the first packet tuple $packet(@n1, n1, n3,$ "data") arrives, it has values $(n1, n3)$ for its equivalence keys, which have never been encountered before, so its $existFlag$ is $False$. But when the second packet tuple $packet(@n1, n1, n3,$ "url") arrives, since it shares the same equivalence keys values with the first packet, the $existFlag$ for it is $True$.

Online Provenance Maintenance. For each rule r triggered in the execution, we selectively maintain the provenance information based on $existFlag$'s value. if $existFlag$ is $False$, the provenance nodes are maintained as tuples in the $ruleExec$ table locally. Otherwise, no provenance information is maintained at all. For example, in Figure 51, when packet(@$n2, n1, n3,$ "data") triggers rule $r1$ at node $n2$, the $existFlag$ is $False$. Therefore, we insert a tuple $ruleExec(n2, rid2, r1, vid1, n1, rid3)$ into the $ruleExec$ table at node $n2$ to record the provenance. The semantics of the inserted tuple are the same as introduced in Section 5.3. In comparison, when packet(@$n2, n1, n3,$ "url") triggers $r2$ at node $n2$, its $existFlag$ is $True$. In this case, we simply execute $r2$ without recording any provenance information.

Output Tuple Provenance Maintenance. For the execution whose $existFlag$ is $True$, we need to associate its output tuple to the shared provenance tree maintained by previous execution. To do this, we maintain a hash table *hmap* on each node to store the reference to the shared provenance tree, wherein the key is the hash value of the equivalence keys, and the value is the node closest to the root in the shared provenance tree. For example, in Figure 51, the shared provenance tree is stored in *hmap* as $\{hash(n1, n3)$: $(n3, rid1)\}$.

We then associate each output tuple *tp* to the shared provenance tree *st*, by looking up its equivalence keys' values in *hmap*. This association is stored as a tuple in the prov table. For example, in Figure 51, the first execution generates the output tuple recv($@n3, n1, n3,$ "*data*"), which is associated to $(n3, rid1)$. This is reflected by the tuple prov$(n3, tid1, n3, rid1, evid1)$ in the prov table (Table 13). $evid1$ is used to identify the event tuple peculiar to the execution, which is not included in the shared provenance tree.

Correctness of Online Compression. We prove the correctness of the online compression algorithm by showing that our compression scheme of provenance trees is lossless – that is, the distributed provenance nodes maintained in the ruleExec and prov tables contain the exact same set of provenance trees that would have been derived by semi-naïve evaluation [54] without compression (Theorem 10). To do this, we define the operational semantics of semi-naïve evaluation of a DELP with a set of transition rules of form: $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}'_{sn}$, where $\mathcal{C}_{sn}$ denotes a state in semi-naïve evaluation that records the complete execution as provenance [20]. We also define a set of transition rules of form: $\mathcal{C}_{cm} \rightarrow_{CM} \mathcal{C}'_{cm}$ for semi-naïve evaluation with our online compression algorithm. Here, $\mathcal{C}_{cm}$ denotes a state in semi-naïve evaluation with compression. The proof is to show that we can assemble entries in the ruleExec and prov tables to reconstruct an original provenance tree $\mathcal{D}$. Likewise, given a provenance tree $\mathcal{D}$, we can also find an identical tree $\mathcal{P}$ encoded as entries in the ruleExec and prov tables. This correspondence is denoted as $\mathcal{D} \sim_d \mathcal{P}$ and can be defined by induction over the structure of provenance trees.

Theorem 10 (Correctness of Compression)**.** $\forall n \in \mathbb{N}$ *and an initial state* $\mathcal{C}_{init}$*, if* $\mathcal{C}_{init} \rightarrow^n_{SN}$

$\mathcal{C}_{sn}$, then $\exists \mathcal{C}_{cm}$ s.t. $\mathcal{C}_{init} \rightarrow^{n}_{CM} \mathcal{C}_{cm}$ and for any provenance tree $\mathcal{D} \in \mathcal{C}_{sn}$, there exists a provenance tree $\mathcal{P} \in \mathcal{C}_{cm}$ s.t. $\mathcal{D} \sim_d \mathcal{P}$ and for any provenance tree $\mathcal{P} \in \mathcal{C}_{cm}$, there exists a provenance tree $\mathcal{D} \in \mathcal{C}_{sn}$ s.t. $\mathcal{D} \sim_d \mathcal{P}$. And the same is true for semi-naïve evaluation when $\mathcal{C}_{cm}$ is given.

The above theorem states that if we initiate a DELP DQ from an initial state $\mathcal{C}_{init}$, and execute DQ for n steps to reach a state $\mathcal{C}_{sn}$, then we can also execute DQ for n steps with the online compression scheme, starting from $\mathcal{C}_{init}$ and ending in $\mathcal{C}_{cm}$. In the end, the sets of provenance trees respectively maintained by these two processes are identical. An implication of Theorem 10 is that compressed provenance trees, like traditional network provenance, would faithfully record the system execution, even if the execution is erroneous due to misconfiguration (e.g., wrong routing tables).

To prove Theorem 10, we show Lemma 11 which implies Theorem 10 as corollary. Lemma 11 shows that semi-naïve evaluation with the online compression scheme is bisimilar to the one that stores provenance trees without compression. This bisimilarity relation shows that both evaluation strategies have identical semantics.

Lemma 11 (Compression Simulates Semi-naïve Evaluation)**.** *$\forall n \in \mathbb{N}$ and an initial state $\mathcal{C}_{init}$, if $\mathcal{C}_{init} \rightarrow^{n}_{SN} \mathcal{C}_{sn}$, then $\exists \mathcal{C}_{cm}$ s.t. $\mathcal{C}_{init} \rightarrow^{n}_{CM} \mathcal{C}_{cm}$ and $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$, and vice versa.*

We define a bisimulation relation $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ between $\mathcal{C}_{sn}$ and $\mathcal{C}_{cm}$ – i.e., $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$ means that when $\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}'_{sn}$, there exists a state $\mathcal{C}'_{cm}$ s.t. $\mathcal{C}_{cm} \rightarrow_{CM} \mathcal{C}'_{cm}$ and $\mathcal{C}'_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}'_{cm}$, and vice versa. Intuitively, $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ relates two states of the two evaluation strategies – i.e., semi-naïve evaluation with and without compression – that execute identical programs to the point of identical program execution states, and, most importantly, for any provenance tree $\mathcal{P} \in \mathcal{C}_{cm}$, there exists a provenance tree $\mathcal{D} \in \mathcal{C}_{sn}$ s.t. $\mathcal{D} \sim_d \mathcal{P}$, and vice versa. Proof details of Lemma 11, along with the formal definition of the bisimulation relation $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$, are presented in the technical report [21].

Generality of equivalence-based compression. The idea of equivalence-based compres-

sion is not just applicable to distributed scenarios, but can be generally used to compress arbitrary provenance tree sets maintained in a centralized manner as well. We adopt the definition of the equivalence relation in Section 5.4.1 because it allows us to use equivalence keys to efficiently identify equivalent provenance trees, thus more suitable for the distributed environment where networking resources (e.g., bandwidth) are scarce.

5.4.4. Inter-Equivalence Class Compression

The online compression scheme introduced in Section 5.4.3 focuses on intra-equivalence class compression of provenance trees – i.e., only trees of the same equivalence class are compressed. In fact, provenance trees of different equivalence classes can be compressed as well. For example, assume a tuple packet($@n2, n2, n3,$ "ack") is inserted into $n2$ in Figure 51 for execution. The produced provenance tree *prov* shares the provenance nodes $rid1$ and $rid2$ in the ruleExec table of Table 13. To avoid the storage of such redundant rule execution nodes, we separate the ruleExec table into two sub-tables: a ruleExecNode table and a ruleExecLink table (Table 14). The ruleExecNode table maintains the concrete rule execution nodes, while the ruleExecLink table, maintained for each provenance tree tr individually, records the parent-child relationship of the rule execution nodes in tr. If two provenance trees, whether in the same equivalence class or not, share the same rule execution node nd, only one copy of the concrete nd will be maintained in the ruleExecNode table. Each tree maintains a reference pointer pointing to nd in their respective ruleExecLink tables.

ruleExecNode			
Loc	**RID**	**RULE**	**VIDS**
n3	**rid1**	r2	NULL
n2	**rid2**	r1	(**vid1**)
n1	**rid3**	r1	(**vid2**)

ruleExecLink			
Loc	**RID**	**NLoc**	**NRID**
n3	**rid1**	n2	**rid2**
n2	**rid2**	n1	**rid3**
n1	**rid3**	NULL	NULL

Table 14: The ruleExecNode table and the ruleExecLink table replacing the ruleExec table in Table 13 to allow for compression of the shared rule execution nodes.

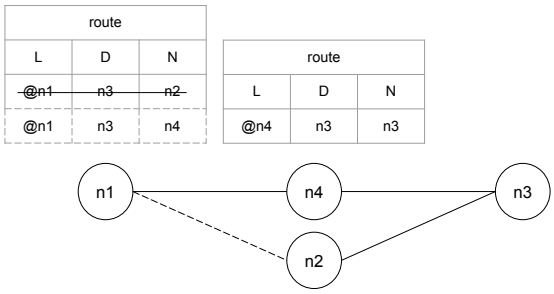


Figure 52: An updated topology of Figure 46. A new node $n4$ is deployed to reach $n3$. The route table of $n1$ is updated to forward packets to $n4$ now.

5.4.5. Updates to Slow-changing Tables

Though we assume that slow-changing tables do not change during a fixpoint computation, our system is designed to handle these updates at runtime. Figure 52 presents an example scenario based on Figure 46, where a network administrator decides to use $n4$, instead of $n2$, as the intermediate hop for packets sent from $n1$ to $n3$. To redirect the traffic, the administrator (1) deletes the route entry route$(@n1, n3, n2)$, and (2) inserts a new route entry route$(@n1, n3, n4)$.

Deletion of a tuple from a slow-changing table – e.g., route$(@n1, n3, n2)$ in Figure 52 – does not affect the stored provenance, as provenance information is monotone – that is, it represents the execution history which is immutable [94].

However, if a tuple tp is *inserted* into a slow-changing table – e.g., route$(@n1, n4, n3)$ – the provenance tree generated by tp could be incorrect or missing. For example, in Figure 52, after route$(@n1, n4, n3)$ is inserted, the provenance trees for all subsequent packets need to be recalculated. However, since these packets are not the first in their equivalence classes, their $existFlag$s are set to $true$. As a result, the provenance tree for the packet traversal on the path $n1 \rightarrow n4 \rightarrow n3$ would not be maintained.

To handle such scenarios, we require that, once a new tuple tp is inserted into a node n's slow-changing table, n should broadcast a control message sig to all the nodes in the system. Any node receiving sig would empty the hash table used for equivalence keys checking

(Section 5.4.3). Therefore, provenance trees will be maintained again for all equivalence classes. In Figure 52, after the insertion of route($@n1, n3, n4$), $n1$ would broadcast a *sig* to all the nodes, including itself. When a new packet *pkt* destined to $n3$ arrives at $n1$, the packet would have its $existFlag$ set as $false$. When this packet traverses the path $n1 \rightarrow n4 \rightarrow n3$, the nodes on the path are expected to maintain the corresponding provenance nodes. In all our network applications, the extra network overhead incurred by the broadcast and the impact on the effectiveness of compression due to reset of the hash table is negligible, as slow-changing tables are updated infrequently in practice (relative to the rate of event arrival). We experimentally validated this, as is shown in Section 5.6.1.

5.4.6. Provenance Querying

To query the provenance tree of an output tuple *tp*, we take the following steps:

- Compute the hash value *htp* of *tp*, and find the tuple *prvtp* in the prov table that has *htp* as its **VID**.
- Initiate a recursive query for the (shared) provenance nodes in the ruleExec table, starting with the values of (**Loc**, **RID**) in *prvtp*. Also, tag the event ID *evid* stored in the attribute **EVID** along with the query.
- When the query reaches a ruleExec tuple at node n with values $(NULL, NULL)$ for (**NLoc**, **NRID**), the tagged *evid* is used to retrieve the event tuple materialized at n.

For example, in Table 13, to query the provenance tree of recv($@n3, n1, n3$, "data"), we first find prov($n3$, $tid1$, $n3$, $rid1$, $evid1$), and use the values $(n3, rid1)$ to initiate the recursive query in the ruleExec table to fetch the provenance nodes $rid1, rid2$ and $rid3$. *evid* is carried throughout the query, and is used to retrieve the event packet($@n1, n1, n3$, "data") when the query stops at ruleExec($n1$, $rid3$, $r1$, $vid2$, $NULL$, $NULL$). The above steps return to the initial querying location a collection of entries from the ruleExec and prov tables. We define a top-level algorithm Query that reconstructs the complete provenance tree $\mathcal{D}$ based on these entries. The pseudocode of Query can be found in Figure 53. Query takes as input

the network state $\mathcal{C}_{cm}$ of the online compression scheme, an output tuple P, an event ID *evid*, and returns a set of provenance trees, each of which corresponds to one derivation of P using the input event tuple of ID *evid*. The example based on Table 13 has only one derivation for the output tuple, so we return a singleton set.

Correctness of Querying. To show that our query algorithm (Query in Figure 53) is able to recover the correct derivation tree of a given tuple from our compressed provenance storage, we state and prove the correctness of the query algorithm (Theorem 12). This theorem states that given initial network state $\mathcal{C}_{init}$ that transitions to $\mathcal{C}_{cm}$ in n steps using the rules for online compression, there exists a network state $\mathcal{C}_{sn}$ for semi-naïve evaluation s.t. for any derivation tree $\mathcal{D}$ that is a proof of output tuple P and derived using an input event tuple *ev* with ID *evid*, Query takes as inputs $\mathcal{C}_{cm}$, P and *evid* and returns a set $\mathcal{M}$ consisting of all derivation trees (including $\mathcal{D}$) that are proofs of P and that were derived using *ev*. Theorem 12 tells us that Query always returns all the derivations in $\mathcal{C}_{sn}$ for P and *evid*.

Theorem 12 (Correctness of the Query Algorithm)**.**

$\forall n \in \mathbb{N}$, *given an initial state* $\mathcal{C}_{init}$ *s.t.* $\mathcal{C}_{init} \rightarrow^{n}_{CM} \mathcal{C}_{cm}$

and there are no more updates to be processed,

then $\exists \mathcal{C}_{sn}$ *s.t.* $\mathcal{C}_{init} \rightarrow^{n}_{SN} \mathcal{C}_{sn}$

and $\forall \mathcal{D}$:P *in the output provenance storage of* $\mathcal{C}_{sn}$

s.t. $\#\textsc{EventOf}(\mathcal{D}) = evid$,

$\exists \mathcal{M}$ *s.t.* $\textsc{Query}(\mathcal{C}_{cm}, P, evid) = \mathcal{M}$ *and* $\mathcal{D} \in \mathcal{M}$

and $\forall \mathcal{D}' \in \mathcal{M} \backslash \mathcal{D}$, $\mathcal{D}'$ *is a proof of* P *stored in* $\mathcal{C}_{sn}$

and $\#\textsc{EventOf}(\mathcal{D}') = evid$.

The proof relies on Lemma 10 to determine that there exists a network state $\mathcal{C}_{sn}$ for semi-naïve evaluation s.t. $\mathcal{C}_{cm}$ and $\mathcal{C}_{sn}$ are bisimilar ($\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$). Given such an $\mathcal{C}_{sn}$, we pick an arbitrary derivation tree $\mathcal{D}$ for tuple P in $\mathcal{C}_{sn}$ that was generated by event tuple *ev* with event ID *evid*. Because $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$, there exists a tuple *prov* for P in a specific prov table

```
1: function QUERY(C_cm, P, evid)
2:     htp ← #P
3:     if ⟨_, htp, _, _, evid⟩ ∈ C_cm then
4:         [prov_1 ··· prov_n] ← GET_PROV(C_cm, htp, evid)
5:         M ← {}
6:         for i ∈ [1, n] do
7:             ⟨loc, htp, rloc_i, rid_i, evid⟩ ← prov_i
8:             P_i ← QR(C_cm, (rloc_i, rid_i))
9:             D_i ← TRANSFORM_TO_D(P_i, evid)
10:            M ← M ∪ D_i
11:        return M
12:    else
13:        return ∅
14: end function
15:
16: function QR(C_cm, (loc, rid))
17:    if loc == NULL and rid == NULL then
18:        return []
19:    else
20:        ruleExec ← GET_RULEEXEC(C_cm, (loc, rid))
21:        ⟨loc, rid, r, vids, nloc, nrid⟩ ← ruleExec
22:        return QR(C_cm, (nloc, nrid)) :: ruleExec
23: end function
```

Figure 53: Pseudocode for querying a provenance tree.

in $\mathcal{C}_{cm}$ storing an association to a specific provenance $\mathcal{P}$, and furthermore $\mathcal{D} \sim_d \mathcal{P}$. By the above reasoning and the semantics of Query, the "If" branch of the If-Else statement on lines 3-13 of Query is taken. On line 4, Get_Prov takes as input $\mathcal{C}_{cm}$, *htp* (the hash of P), and *evid*, then returns every $prov_i$ in the prov tables of $\mathcal{C}_{cm}$ containing an association $(rloc_i, rid_i)$ to a provenance tree $\mathcal{P}_i$ for P that was derived using ev. By the relation in $\mathcal{E}_5$, each $\mathcal{P}_i$ is recorded in $\mathcal{C}_{cm}$. We use Qr to retrieve $\mathcal{P}_i$. If we can show that Qr can correctly retrieve $\mathcal{P}_i$, it is straightforward to show that Transform_To_D recovers $\mathcal{D}$ when given $\mathcal{P}_i$ and *evid* as inputs. Hence, the conclusion holds.

We still need to show that recursive algorithm Qr will return $\mathcal{P}_i$ when given $\mathcal{C}_{cm}$ and the association $(rloc_i, rid_i)$ to provenance $\mathcal{P}_i$. We prove Lemma 13 below. The proof uses a *uniqueness* property on elements in the ruleExec table–the first two arguments of *ruleExec*

(i.e. *loc* and *rid*) are *primary keys* that uniquely determine it. Thus, given any $ruleExec$ and $ruleExec'$, that agree on the first two arguments *loc* and *rid*, then $ruleExec = ruleExec'$.

Lemma 13 (Correctness of Qr)**.** *Given that* $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$ $\mathcal{C}_{cm}$ *and* $\mathcal{D}{:}P \in \mathcal{C}_{sn}$ *and* $ruleExec = \langle loc, rid, r, vids, nloc, nrid\rangle$ *and* $\mathcal{P} :: ruleExec$ *is stored in the* ruleExec *tables of the local states of* $\mathcal{C}_{cm}$ *and* $\mathcal{D} \sim_d \mathcal{P} :: ruleExec$*, then* $\textsc{Qr}(\mathcal{C}_{cm}, (nloc, nrid)) = \mathcal{P} :: ruleExec$.

We prove Lemma 13 by induction over ℓ, the length of $\mathcal{P} :: ruleExec$.

Base Case A: $\ell = 0$. By the assumption we have $\mathcal{P} :: ruleExec = []$. By the definition of $\sim_d$ that relates derivation trees to compressed provenance trees, $\nexists \mathcal{D} \in \mathcal{C}_{sn}$ s.t. $\mathcal{D} \sim_d []$. Thus the antecedent of the lemma is false.

Base Case B: $\ell = 1$. By the assumption we have $\mathcal{P} = []$ and thus $\mathcal{P} :: ruleExec = ruleExec$. Because $\mathcal{D}{:}P \sim_d ruleExec$, $\mathcal{D}$ has only one rule. Thus (*nloc*, *nrid*) are null by the correspondence relation as only rule r was used to derive P. Because (*loc*, *rid*) are not null, the "Else" branch of the If-Else statement on Lines 17-22 of Qr is taken. Therefore on Line 20 of Qr, the algorithm finds $ruleExec$ (where $ruleExec = \langle loc, rid, r, vids, nloc, nrid\rangle$) by the uniqueness property. The query $\textsc{Qr}(\mathcal{C}_{cm}, (nloc, nrid))$ initiates a query for an empty rule provenance list, that by *Base Case A* returns an empty list. By Line 22 of Qr, we have $\textsc{Qr}(\mathcal{C}_{cm}, (loc, rid)) = [] :: ruleExec$ as desired.

Inductive Case: $\ell = k + 1 \geqslant 2$. By assumption, *nloc* and *nrid* are not null, thus the "Else" branch of the If-Else statement on Lines 17-22 of Qr is taken. Therefore on Line 20 of Qr the algorithm finds $ruleExec$ (where $ruleExec = \langle loc, rid, r, vids, nloc, nrid\rangle$) by the uniqueness property. By assumption, there exists $\mathcal{P}'$ and $ruleExec'$ s.t. $\mathcal{P} = \mathcal{P}' :: ruleExec'$ and $ruleExec'$ is not null. By the above and the correspondence $\mathcal{D} \sim_d \mathcal{P} :: ruleExec'$, exists $\mathcal{D}'$ in $\mathcal{C}_{sn}$ that is a subderivation of $\mathcal{D}$ s.t. $\mathcal{D}' \sim_d \mathcal{P}$ and $ruleExec' = \langle nloc, nrid, r', vids', nloc', nrid'\rangle$. Using I.H. we can obtain $\textsc{Qr}(\mathcal{C}_{cm}, (nloc, nrid)) = \mathcal{P}' :: ruleExec'$. By Line 22 of Qr, we have $\textsc{Qr}(\mathcal{C}_{cm}, (loc, rid)) = \mathcal{P}' :: ruleExec' :: ruleExec = \mathcal{P} :: ruleExec$ as desired.

5.5. Implementation

We have implemented a prototype based on enhancements to the RapidNet [59] declarative networking engine – RapidNet compiles NDlog programs to generate efficient distributed network protocol implementation – First, we provide compiler extensions to add support for DELP. We further provide a static analysis module that takes as input a DELP, and generates equivalence keys for event relations.

$$
\begin{array}{ll}
eqc\_r1 & \mathsf{equiHash}(@loc, src, dst, data, eqid, id) :- \\
& \quad \mathsf{packet}(@loc, src, dst, data), \\
& \quad \mathsf{progID}(@loc, dst, id), \\
& \quad eqid := f\_sha1(loc + dst). \\
eqc\_r2 & \mathsf{hashCount}(@loc, src, dst, data, eqid, id, a\_COUNT\langle * \rangle) :- \\
& \quad \mathsf{equiHash}(@loc, src, dst, data, eqid, id), \\
& \quad \mathsf{hashSet}(@loc, eqid). \\
eqc\_r3 & \mathsf{hashSet}(@loc, eqid) :- \\
& \quad \mathsf{hashCount}(@loc, src, dst, data, eqid, id, hcount), \\
& \quad hcount == 0. \\
eqc\_r4 & \mathsf{packet}(@loc, src, dst, data, existFlag) :- \\
& \quad \mathsf{hashCount}(@loc, src, dst, data, eqid, id, hcount), \\
& \quad hcount == 0, \\
& \quad existFlag := false. \\
eqc\_r5 & \mathsf{packet}(@loc, src, dst, data, existFlag) :- \\
& \quad \mathsf{hashCount}(@loc, src, dst, data, eqid, id, hcount), \\
& \quad hcount! = 0, \\
& \quad existFlag := true.
\end{array}
$$

Figure 54: Rewritten program implementing equivalence key checking for packet forwarding in Figure 45.

At compile time, we further add a program rewrite step that rewrites each DELP into a new *NDlog* program $prog_{PROV}$ that supports online provenance maintenance and compression at runtime. Interestingly, because all the provenance tables are maintained and queried as relational tables, no additional runtime enhancements are required beyond rule writes. For example, Figure 54 presents the rewritten DELP rules performing equivalence key checking for the packet forwarding program in Figure 45. In this program, rule eqc_r1 computes the hash value $eqid$ of the equivalence keys attributes loc and dst, and stores $eqid$ in a equiHash tuple. Rule eqc_r2 is an aggregation rule that checks whether $eqid$ exists in the hashSet

table. If the aggregation result *hcount* is *0*, *eqc_r*3 would update hashSet table with the new *eqid*, and *eqc_r*4 would generate *existFlag* with value $false$. Otherwise, if *hcount* is not *0*, *eqc_r*5 generates *existFlag* with value *true*. We omit the detailed nineteen rules of $prog_{PROV}$, but present all the relations of $prog_{PROV}$ in Table 15.

In $prog_{PROV}$, $prog_{PROV}$ is triggered when a tuple initPacket($Node, SrcAdd, DstAdd, Data$) is inserted. The program first checks whether the hash values of the packet's equivalence keys – i.e., ($Node, DstAdd$) in the packet forwarding example – exist in a hash table equiHashTable. If the hash value does not exist, $prog_{PROV}$ needs to maintain the provenance for the execution of the inserted packet. This is done by first joining the initPacket with the local flowEntry and conn to perform the forwarding. But instead of generating a new packet that is to be sent to the next hop, $prog_{PROV}$ first generates a temporary packet epacketTemp, which contains the contents of the original initPacket as well as necessary information for provenance maintenance, such as the triggered rule's name and used rule bodies. Based on epacketTemp, the program will generate a ruleExec tuple that records the provenance, and a provLink tuple that connects the local provenance node to the previous one to facilitate future querying. Also, a packet will be generated to be sent to the next hop. On the other hand, if the hash value of the equivalence keys exist, only the new packet will be produced, and no provenance-related tuples would be generated.

When the packet is forwarded to intermediate nodes in the network, each node would produce a temporary epacketTemp to help record the provenance information. When the packet is received at the destination, additional processing is needed. The program would first generate a temporary tuple erecvPacketTemp, which helps record the provenance information at the last hop. In addition, the erecvPacketTemp will derive a provHashTable tuple and a provRef tuple. The provHashTable associates the values of equivalence keys to the last provenance node of the shared provenance in the equivalence class, while the provRef tuple associates the received packet to the corresponding equivalence keys' values and the input initPacket. With the above two tuples, the user could construct the complete provenance information

during querying.

Predicate	Description
initPacket(@*Node*, *SrcAdd*, *DstAdd*, *Data*, *PIDequi*, *ProgID*)	Initiate a packet at node *Node* to be sent from *SrcAdd* to *DstAdd*, with payload *Data*. The packet also contains the hash value of equivalence key *PIDequi*, and the program ID *ProgID*.
equiHashTable(@*Node*, *DstAdd*, *PIDequi*)	A hash table at node *Node*, with the destination address *DstAdd* as key and the hash value of the equivalence keys *PIDequi* as value.
packet(@*Node*, *SrcAdd*, *DstAdd*, *Data*, *PIDHash*)	A packet transmitted to *Node* with the source address *SrcAdd*, the destination address *DstAdd* and the payload *Data*. *PIDHash* contains the hash values related to provenance information (e.g., equivalence keys)
flowEntry(@*Node*, *DstAdd*, *Next*)	A routing entry at node *Node* that indicates the next hop *Next* towards the destination *DstAdd*.
conn(@*Node*, *Next*)	A link connecting node *Node* and *Next*
epacketTemp(@*Node*, *Next*, *SrcAdd*, *DstAdd*, *Data*, *RID*, *R*, *List*, *Tag*)	A temporary packet at *Node* containing the provenance, including the executed rule's *RID*, its name *R*, and its bodies *List*. *Tag* contains the global provenance information (e.g., equivalence keys).
packetProv(@*Node*, *SrcAdd*, *DstAdd*, *Data*, *Tag*)	A transmitted packet whose provenance information needs to be recorded. The meaning of attributes is the same as that of epacketTemp.
ruleExec(@*RLOC*, *RID*, *R*, *List*)	A ruleExec records the provenance information of a triggered rule *R*. The meaning of the attributes is the same as that of *epacketTemp*.
provLink(@*RLOC*, *RID*, *CurCount*, *Preloc*, *PreRID*, *PreCount*, *PIDequi*)	A provLink tuple links two consecutive provenance nodes of *RID* and *PreRID* to enable querying. *CurCount* and *PreCount* are program step counters that differentiate two provenance nodes of the same contents.
recvPacket(@*Node*, *SrcAdd*, *DstAdd*, *Data*)	A received packet at *Node*, from the source *SrcAdd* to the destination *DstAdd* with payload *Data*.
provHashTable(@*Node*, *PIDequi*, *ProgID*, *Loc*, *RID*, *Count*)	A hash table at node *Node* that stores the reference to the last provenance node *RID* at node *Loc*, corresponding to the equivalence keys *PIDequi*.
provRef(@*Node*, *PID*, *PIDequi*, *PIDev*)	A hash table at node *Node* that stores the reference to the input tuple *PIDev* corresponding to the equivalence keys *PIDequi*.
recvPacketNP(@*Node*, *SrcAdd*, *DstAdd*, *PIDHash*)	A received packet whose traversal of the network does not trigger the provenance maintenance. *PIDHash* contains the hash values of the equivalence keys and the input tuple.

Table 15: Relations for maintaining compressed provenance

5.5.1. Properties of Provenance Maintenance Programs

We further show that static analysis introduced in Chapter 4 could facilitate runtime analysis by verifying properties of provenance maintenance programs. We verify three properties that are supposed to hold when provenance is maintained and compressed (Table 16).

Prop	**Property description**	**Formal specification**	**Result**
φ_{PROV_1}	When a packet was sent out, the provenance node for the packet has been recorded.	$\forall Next, SrcAdd, DstAdd, Data, PIDequi,$ $PIDev, ProgID,$ $\mathsf{packet}(Next, SrcAdd, DstAdd, Data,$ $PIDequi, PIDev, ProgID)$ $\supset$ $\exists Node, RID, CurCount, Preloc, PreRID,$ $PreCount,$ $\mathsf{provLink}(Node, RID, CurCount, Preloc,$ $PreRID, PreCount, PIDequi)$	false
φ_{PROV_2}	When a packet which is the first of its equivalence class was sent out, the provenance node for the packet has been recorded.	$\forall Next, SrcAdd, DstAdd, Data, NewCount,$ $RLOC, RID, PIDequi, PIDev, ProgID,$ $\mathsf{packetProv'}(Next, SrcAdd, DstAdd, Data,$ $NewCount, RLOC, RID, PIDequi, PIDev,$ $ProgID)$ $\supset$ $\exists CurCount, Preloc, PreRID, PreCount,$ $\mathsf{provLink}(RLOC, RID, CurCount, Preloc,$ $PreRID, PreCount, PIDequi)$ $\wedge NewCount = CurCount + 1.$	true
φ_{PROV_3}	The provenance node of each received tuple which is the first of the equivalence class is linked to the shared subtree.	$\forall Node, SrcAdd, DstAdd, Data,$ $\mathsf{recvPacket}(Node, SrcAdd, DstAdd, Data),$ $\supset$ $\exists PIDequi, PIDev, PID, ProgID,$ $Loc, RID, Count,$ $\mathsf{provHashTable}(Node, PIDequi, ProgID,$ $Loc, RID, Count)$ $\wedge \mathsf{provRef}(Node, PID, PIDequi, PIDev)$	true

Table 16: Safety properties of maintenance of compressed provenance and verification results

In Table 16, property φ_{PROV_1} intends to show that each transmitted packet has its provenance recorded. The fact that this property is false actually shows that our solution correctly avoids maintaining concrete provenance for all incoming packets– i.e., only packets that are the first of their equivalence classes need to have their provenance recorded (φ_{PROV_2}). Prop-

erty φ_{PROV_3} shows that the representative provenance tree is correctly associated to each equivalence class when the first packet of the equivalence class is received. This ensures that all subsequent provenance trees in the equivalence class could share the maintained provenance tree.

The proof strategy follows the symbolic execution-based approach introduced in Chapter 4. We first construct a dependency graph G_{prov} for $prog_{PROV}$, and use G_{prov} to generate a derivation pool DP_{prov} that contains the derivations, along with corresponding constraints, for all relations in $prog_{PROV}$. A property φ is verified only if no derivation could violate φ.

5.6. Evaluation

We have implemented a prototype based on enhancement to the RapidNet [59] declarative networking engine. At compile time, we add a program rewrite step that rewrites each DELP program into a new program that supports online provenance maintenance and compression at runtime. We evaluate our prototype to understand the effectiveness of the online compression scheme. In all the experiments, we focus on comparing four techniques for maintaining distributed provenance. The first is ExSPAN [95], a typical network provenance engine. We maintain uncompressed provenance trees in the same way as ExSPAN. The second is the distributed provenance maintenance with basic storage optimization (Section 5.3). The third is the provenance maintenance using equivalence-based compression (Section 5.4). We also compared our solution with the reactive provenance maintenance in DTaP [94] using the packet forwarding application. Reactive maintenance, during system execution, stores only the non-deterministic input tuples (e.g., packet and route tuples). When a user queries the provenance tree of an output tuple, he/she needs to replay the execution of the stored input tuples on the relevant nodes – i.e., the nodes that forward the input tuple so as to generate the queried output tuple – and queries the provenance re-generated during replay.

In the evaluation section, we refer to the above four techniques as *ExSPAN*, *Basic*, *Advanced*

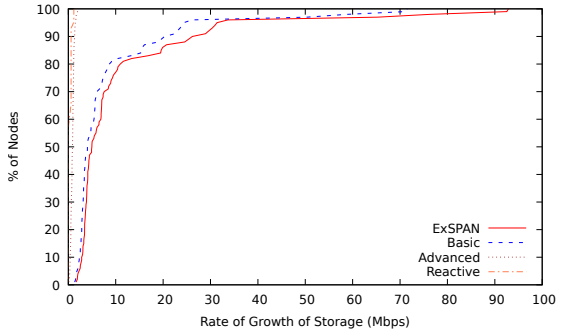


Figure 55: Cumulative growth rate of provenance with 100 pairs of communicating nodes, at input rate of 100 packets/second.

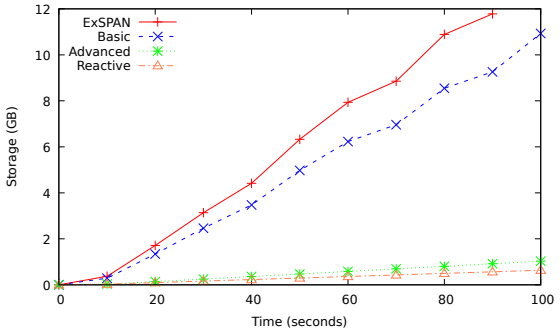


Figure 56: Provenance storage growth of all nodes, with input rate of 100 packets/second for 100 pairs of communicating nodes.

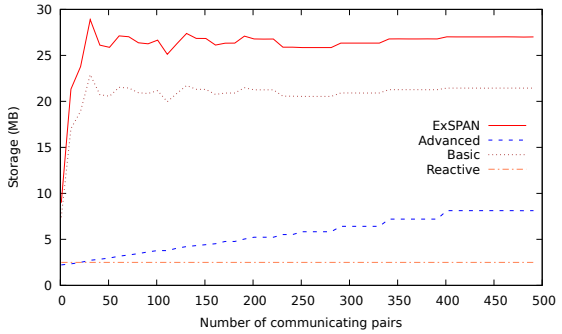


Figure 57: Provenance storage usage with 2000 input packets evenly distributed among given number of pairs.

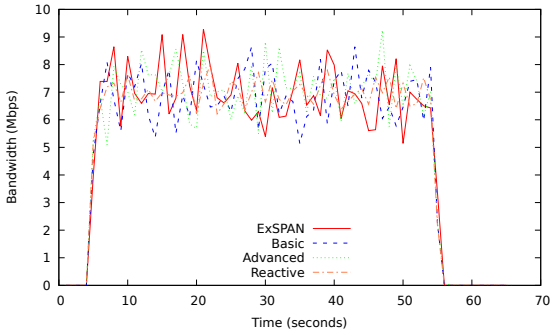


Figure 58: Bandwidth consumption during packet forwarding, with 500 pairs of nodes, each transmitting 100 packets.

and *Reactive* respectively.

Workloads. Our experiments are carried out on two classic network applications: packet forwarding (Section 5.1) and Domain Name System (DNS) resolution. DNS resolution is an Internet service which translates human-readable domain names into IP addresses. Both applications are event-driven, and typically involve large volume of traffic during execution. The high-volume traffic incurs large storage overhead if we maintain provenance information for each packet/DNS request, which leaves potential opportunity for compression. The workloads are also sufficiently different to evaluate the generality of our approach. Packet forwarding involve larger messages along different paths in a graph, while DNS lookups involve smaller messages on a tree-like topology.

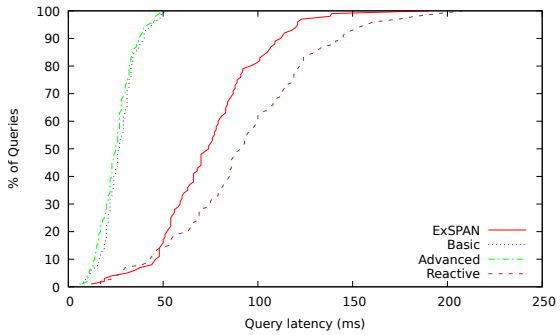


Figure 59: Cumulative distribution of provenance querying latency for 100 random queries with 100 node pairs.

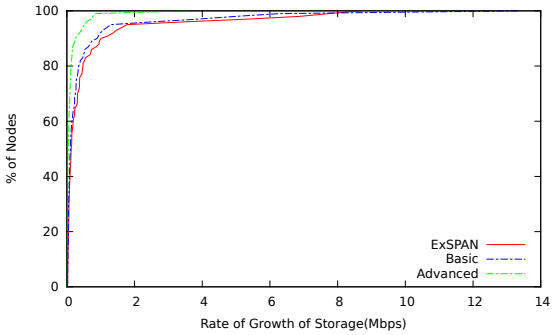


Figure 60: Cumulative provenance storage growth rate of nameservers with input request at a rate of 1000 requests/second.

Testbed. In our experiment setup, we write the packet forwarding and DNS resolution applications in DELP, and use our enhanced RapidNet [59] engine to compile them into low-level (i.e., C++) execution codes.

The experiments for measuring storage and bandwidth are run on the ns-3 [64] network simulator, which is a discrete-event simulator that allows a user to evaluate network applications on a variety of network topologies. The simulation is run on a 32-core server with Intel Xeon 2.40 GHz CPUs. The server has 24G RAM, 400G disk space, and runs Ubuntu 12.04 as the operating system. We run multiple node instances on the same machine communicating over the ns-3 simulated network.

Performance Metrics. The performance metrics that we use in our experiments are: (1) the storage overhead, and (2) the network overhead (i.e., bandwidth consumption) for provenance maintenance, and (3) the query latency when different provenance maintenance techniques are adopted.

In our experiments, the relational provenance tables are maintained in memory. To measure the storage occupation, we use the boost library [74] to serialize C++ data structures into binary data. At the end of each experiment run, we serialize the per-node provenance tables (i.e., ruleExec table and prov table) into binary files, and measure the size of files to estimate the storage overhead.

5.6.1. Application #1: Packet Forwarding

Our first set of results is based on the packet forwarding program in Figure 45. The topology we used for packet forwarding is a 100-node transit-stub graph, randomly generated by the GT-ITM [89] topology generator. In particular, there are four transit nodes – i.e., nodes through which traffic can only traverse, but not initiated – in the topology, each connecting to three stub domains, and each stub domain has eight stub nodes – i.e., nodes where traffic only originates or terminates. Transit-transit links have 50ms latency and 1Gbps bandwidth; transit-stub links have 10ms latency and 100Mbps bandwidth; stub-stub links have 2ms latency and 50Mbps bandwidth. The diameter of the topology is 12, and the average distance for all node pairs is 5.3. Each node in the topology runs one instance of the packet forwarding program.

In the experiment, we randomly selected a number of node pairs (s, d) – where s is the source and d is the destination– and sent packets from s to d while the provenance of each packet is maintained. To allow the packets to be correctly forwarded in the network, we pre-computed the shortest path p between s and d using a distributed routing protocol written as a declarative networking program[55]. The routes are stored in the route tables at each node in p.

Storage of Provenance Trees Figure 55 shows the CDF (Cumulative Distribution Function) graph of storage growth for all the nodes in the 100-node topology. In the experiment, we randomly selected 100 pairs of nodes, and continuously sent packets within each pair at the rate of 100 packets/second. As packets are transmitted, their provenance information is incrementally created and stored at each node (and optionally compressed for Basic and Advanced). We calculated the average storage growth rate of each node, and plotted a CDF graph based on the results. We observe that ExSPAN has the highest storage growth rate among the three: 20% of the nodes have storage growth greater than 5 Mbps; 4% of nodes (i.e., transit nodes) have storage growth greater than 30 Mbps. This is

because a number of node pairs share the same transit node in their paths. As expected, Basic has less storage growth rate compared to ExSPAN, as it removes intermediate packet tuples from the provenance tables of each node. Reactive shows significantly lower storage growth rates, where 57% of the nodes have their growth rates close to 0 Mbps – i.e., most of the nodes which have no incoming packets stop maintaining provenance information after the routing tables have been set up – and the rest of the nodes have lower than 1.2 Mbps storage growth. Advanced significantly outperforms ExSPAN and Basic: all the nodes in the topology have less than 2 Mbps storage growth rate. The gap between Advanced and ExSPAN results from the fact that Advanced only maintains one representative provenance tree for each pair of nodes, while ExSPAN has to maintain provenance trees for all the traversing packets. However, compared to Reactive, Advanced has higher storage growth rate. The reason is that, apart from the storage of the incoming packets' provenance information – this information is maintained by both Advanced and Reactive – Advanced also needs to maintain the provenance of each output tuple for query purpose.

Figure 56 shows the total storage usage with continuous packet insertion. We ran the experiment for 100 seconds and took a snapshot of the storage every 10 seconds. The figure shows that ExSPAN has the highest storage overhead. For example, it reaches the storage of 11.8 GB at 90 seconds, and keeps growing in a linear fashion. Basic has a similar pattern, with 9.2 GB at 90 seconds. However, Advanced presents lower storage growth, where at 90 seconds it only consumes storage space of 0.92 GB. Reactive, however, has even lower storage growth with storage occupation of 0.57 GB at 90 seconds. This is about half of the storage space needed by Advanced. The extra storage of Advanced comes from two sources: (1) the materialized shared sub-provenance trees, and (2) the materialized output tuples. Reactive adopts a replay-based approach, thus only maintaining the input tuples, which, without multicast, are of the same number as output tuples. Advanced tradeoffs the extra storage with efficiency in provenance querying, as shown in Section 5.6.1. We further calculate the average growth rate for all three lines. ExSPAN's storage grows at 131 MB/second, Basic at 109 MB/second, Advanced at 10.3 MB/second, and Reactive at 6.3

MB/second. This means that ExSPAN could fill a 1TB disk within 2 hours, Basic within 2.5 hours, whereas Advanced more than one day, and Reactive almost two days.

Figure 57 shows the storage usage when we increase the number of communicating pairs, but keep the total number of packets the same (i.e., 2000 packets). All the packets are evenly distributed among all the communicating pairs. We observe that the storage usage of ExSPAN and Basic remains almost constant: ExSPAN's total storage usage is around 27 MB and Basic's total storage usage is around 21 MB. This is because in both cases, each packet has a provenance tree maintained in the network, irrelevant of its source and destination. The burst of storage at the beginning of the experiments for ExSPAN and Basic is due to the fact that sizes of provenance trees also depend on the length of the path that each packet traverses. In our experiment, the initial node pairs happen to have path length shorter than the average path length in the topology, thus incurring less storage overhead. Reactive's storage usage is completely constant in this experiment and remains as low as 2.5MB. The provenance information maintained by Reactive is always the 2000 injected packets, thus keeping constant regardless of the increase of node pairs.

For the case of Advanced, its storage usage increases with the number of communicating pairs. This is because each communicating pair forms an equivalence class, and maintains one copy of the shared provenance tree in the equivalence class. Therefore, whenever a new communicating pair is added to the experiment, we need to maintain one more provenance tree for that pair, which increases the total storage. The storage gap between Advanced and Reactive increases with more node pairs. However, the gap is bounded, as in the most extreme case, each pair of nodes sends one packet between each other, creating one provenance tree for each of the 2000 packets. Despite the storage increase, Advanced still consumes much less storage space than ExSPAN and Basic.

In summary, we observe that Basic is able to reduce storage growth, and in combination with the equivalence-based compression (Advanced), the storage reduction is significant compared to the naïve provenance maintenance strategy – i.e., a 92% reduction over ExS-

PAN. However, Reactive could achieve even lower storage consumption against Advanced – i.e., a further 38% by maintaining only the non-deterministic input tuples. Such reduction is obtained at the cost of longer query latency, as is shown in Section 5.6.1. Therefore, both solutions have tradeoff between storage overhead and query latency, and an selection strategy will be an interesting topic for future exploration (Section 7.1.3).

Network Overhead. Figure 58 presents the bandwidth utilization when we randomly selected 500 pairs of nodes and each pair communicated 100 packets. As expected, the bandwidth consumption of Advanced is close to other solutions. This is because the extra information carried with each packets is merely *existFlag* and some auxiliary data (e.g., hash value of the event tuple), which is negligible compared to the large payload of the packets. We repeated the experiment for Advanced, but updated a route every 10 seconds, in order to study the effects of updates to slow-changing tuples. We observe a negligible bandwidth increase of 0.6%.

Query Latency To evaluate latency of queries, we used an actual distributed implementation that can account for both network delays and computation time. We ran the packet forwarding application on a testbed consisting of 25 machines. Each machine is equipped with eight Intel Xeon 2.67 GHz CPUs, 4G RAM and 500G disk space, running CentOS 6.8 as the operating system.

On each machine, we ran up to four instances of the same packet forwarding application with provenance enabled. Instead of communicating via the ns-3 network, actual sockets were used over a physical network. In total, there were 100 nodes, connected together using the same transit-stub topology we used for simulation.

In our experiment, we executed 100 queries, selected on random nodes, where each query returned the provenance tree for a recv tuple corresponding to a random source and destination pair, where the destination node is the starting point of the query. The query is executed in a distributed fashion as described in Section 5.4.6. Based on our physical

network topology, each query takes 5.3 hops on average in the network. We repeated the experiment for Basic, Advanced, Reactive and ExSPAN for 100 queries each.

Figure 59 shows our experimental results in the form of a CDF of query latency. We observe that both Basic and Advanced have significantly lower latency compared to ExSPAN and Reactive. For example, the mean/median for ExSPAN is 75ms and 74ms respectively, and 93ms and 91ms for Reactive, as compared to only 25.5ms and 25ms for Basic. This is approximately 3X reduction for ExSPAN in latency times, and 4X for Reactive. The overhead saving of Advanced and Basic is attributed to the fact that the querying algorithm is designed based on DELP, which allows for more optimization compared to ExSPAN. For example, a query of Advanced knows which tuples are the slow-changing tuples during the querying process, thus it could directly return these slow-changing tuples without further probing their provenance information, while ExSPAN needs to probe one level further to determine that these slow-changing tuples are treated as base input themselves. Reactive requires even longer latency, as it involves the replay time of the original network events. Also, note that in our experiments, we assume the replay of Reactive starts right with the input packet corresponding to the query – i.e., Reactive makes a checkpoint right before the packet is inserted – but this may not be the case for all real scenarios. If the checkpoint is taken earlier, the querying time for Reactive could be even longer.

In the original work of DTaP [94], similar results were shown for query latency. In the experiments, DTaP compared the query latency of proactive and reactive maintenance respectively. Proactive provenance maintenance stores the provenance information in the form of "deltas" between adjacent provenance versions. The average query latency for proactive maintenance is within 0.34 seconds, compared to 37.7 seconds for reactive maintenance. Our online compression solution achieves both low storage overhead and low query latency, with the help of static analysis of programs.

5.6.2. Application #2: DNS Resolution

DNS resolution [58] is an Internet service that translates the requested domain name, such as "www.hello.com", into its corresponding IP address in the Internet. In practice, DNS resolution is performed by DNS nameservers, which are organized into a tree-like structure, where each nameserver is responsible for a domain name (e.g., "hello.com" or ".com"). We used the *recursive name resolution* protocol in DNS, and implemented the protocol as a DELP program (Figure 61. During the execution of each DELP DNS program, provenance support is enabled so that the history DNS requests can be queried.

The program in Figure 61 is composed of four rules. Rule *r1* forwards a DNS request of ID *RQID* to the root nameserver *RT* for resolution. The request is generated by the host *HST* for the URL *URL*. Rule *r2* is triggered when a nameserver X receives a DNS request for *URL*, but has delegated the resolution of sub-domain *DM* corresponding to *URL* to another nameserver SV. Rule *r2* then forwards the DNS request to SV for further DNS resolution. Rule *r3* generates a DNS resolution result containing the IP address *IPADDR* corresponding to the requested *URL*, when *URL* matches an address record on the nameserver X. Finally, Rule *r4* is responsible for returning the DNS result to the requesting host *HST*.

$r1$ request$(@RT, URL, HST, RQID)$:–
url$(@HST, URL, RQID)$.
rootServer$(@HST, RT)$.
$r2$ request$(@SV, URL, HST, RQID)$:–
request$(@X, URL, HST, RQID)$,
nameServer$(@X, DM, SV)$.
f_isSubDomain$(DM, URL) == true$.
$r3$ dnsResult$(@X, URL, IPADDR, HST, RQID)$:–
request$(@X, URL, HST, RQID)$,
addressRecord$(@X, URL, IPADDR)$.
$r4$ reply$(@HST, URL, IPADDR, RQID)$:–
dnsResult$(@X, URL, IPADDR, HST, RQID)$.

Figure 61: DELP for DNS resolution.

We synthetically generated the hierarchical network of DNS name servers. In total, there

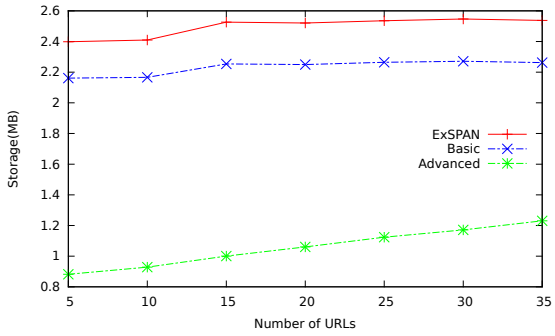


Figure 62: Provenance storage growth with increasing URLs, with 200 requests sent in total.

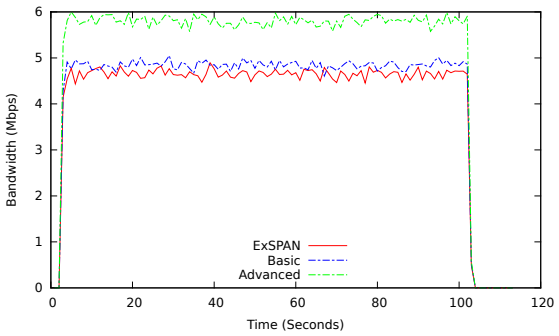


Figure 63: Bandwidth consumption for DNS resolution with 100,000 DNS requests.

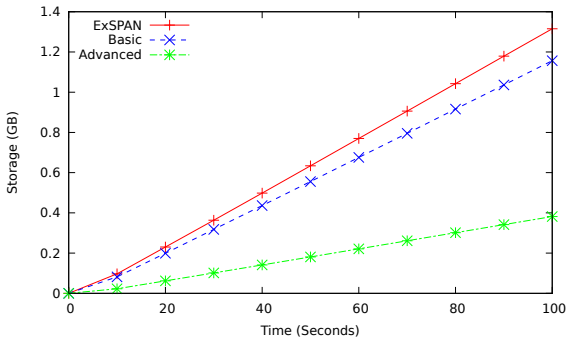


Figure 64: Provenance storage growth with continuous input requests at 1000 requests/sec.

were 100 name servers, and the maximum tree depth is 27. Our workload consists of clients issuing requests to 38 distinct URLs. In total, DNS requests were issued at a rate of 1000 requests/second. Our topology resembles real-world DNS deployments. Prior work [44] has shown that in reality, the requested domain names satisfy Zipfian distribution. In our experiments, we adopted the same distribution.

Storage of Provenance Trees

Figure 60 shows the provenance storage growth rate for all nameservers in the Domain Name System over a 100 seconds duration. We measure the storage growth of each nameserver by first measuring the growth rate of each 10-second interval, and calculating the average growth rates over all 10 intervals. We observe that ExSPAN has the largest storage growth

rate for each node among the three experiments, while Advanced has the lowest storage growth rate. Note that the reduction of storage growth rate in Figure 60 is not as significant as that in the packet forwarding experiments (Figure 55). For example, 80% of nameservers in ExSPAN have storage growth rate less than 476 Kbps. while the rate is 121 Kbps for Advanced. Advanced is four times better than ExSPAN, compared to 11 times in packet forwarding. The reason is that, compared to packet forwarding, we rate the total throughput of incoming events – i.e., packet tuples in packet forwarding and request tuple in DNS resolution – and this causes the storage growth rate at each node using either ExSPAN and Basic to drop as well.

Figure 64 shows provenance storage growth for all nameservers. We record storage growth rates at 10-second intervals. In Figure 64, the storage of ExSPAN and Basic grows much faster than that of Advanced. Specifically, the growth rate of ExSPAN, Basic and Advanced are 13.15 Mbps, 11.57 Mbps and 3.81 Mbps respectively, and their storage space at 100 seconds reaches 1.32 GB, 1.16 GB, and 0.38 GB. With the given rates, ExSPAN would fill up a 1TB disk within 21 hours, Basic within 24 hours, and Advanced up to 3 *days*.

Figure 62 shows the storage growth when we increased the number of requested URLs. In this experiment, we fixed the total number of requests at 200, so that when more URLs were added, there would be fewer duplicate requests. In Figure 62, the storage overhead for ExSPAN and Basic remains stable at around 2.5 MB and 2.26 MB respectively. This is because the storage overhead is mostly determined by the number of provenance trees, which is equal to the number of incoming requests (i.e., 200 in this case). For Advanced, the storage grows at a rate of 11.6 Kb per URL. This is expected as we need to maintain one provenance tree for each equivalence class, and the number of equivalence classes grows in proportion to the number of URLs. Similar to our packet forwarding results, despite the storage growth, Advanced still requires significantly less storage compared ExSPAN and Basic. Unless a URL is only requested once (highly unlikely in reality), which represents the worst case for Advanced, Advanced always performs better than ExSPAN and Basic.

Network Overhead

Figure 63 shows the bandwidth usage with elapsed time when 100,000 requests are continuously sent to the root nameserver. All three experiments finish within 102 seconds. Throughout the execution, ExSPAN and Basic have similar bandwidth usage at around 4.5 MBps. On the other hand, Advanced's bandwidth usage is about 6 MBps, which is about 25% higher than the other two techniques. This is because unlike in the packet forwarding where each packet carries a payload of 500 characters, each DNS request does not have any extra payload. Therefore, the meta-data tagged with each request (e.g., $existFlag$) accounts for a large part of the size of each request, resulting in higher bandwidth overhead.

5.6.3. Summary

In all the experiments, the storage overhead of Advanced is significantly lower than ExSPAN and Basic. Moreover, the storage gap varies – when a same equivalence class contains more events, the ExSPAN would incur much higher provenance storage compared to Advanced. Reactive, however, outperforms Advanced in storage occupation, by maintaining only the non-deterministic input and sacrificing the query latency. Therefore, tradeoff exists as to which solution is more desirable.

Regarding the network overhead, when each packet carries large payload, such as the case in packet forwarding, the extra provenance maintenance information (e.g., $existFlag$) carried with each intermediate tuple is negligible. But when the execution incurs light traffic, like DNS requests, the overhead of extra information accounts for a large portion of the intermediate tuples, causing extra network overhead. Reactive is supposed to perform the best in this case, as it incurs no provenance overhead at all during the execution. We refer interested readers to ExSPAN [95] for comparison of network overhead against the normal execution without provenance maintenance. The network overhead for light-traffic execution could be reduced by several optimizations, such as removing the ID of the event tuple when there is no projection throughout the execution.

CHAPTER 6

Related Work

We summarize the related works that analyze distributed systems using static analysis and dynamic analysis.

6.1. Static Analysis of Distributed Systems

We first introduce important related works that apply static analysis to distributed systems for verification purpose.

6.1.1. Verification of System Trace Properties.

There are a number of works that focus on developing logic for verifying trace properties of programs (protocols) that run concurrently with adversaries [27, 35]. The first piece of our work (Chapter 3) are inspired by their program logic that requires the asserted properties of a program to hold even when that program runs concurrently with adversarial programs.

6.1.2. Networking Protocol Verification.

There has been a large body of work on verifying the correctness of various network protocol design and implementations using proof-based and model-checking techniques [11, 39, 30]. Also, several papers have investigated the verification of route authenticity properties on specific wireless routing protocols for mobile networks [6, 7, 26]. They have showed that identifying attacks on route authenticity can be reduced to constraint solving, and that the security analysis of a specific route authenticity that depends on the topologies of network instances can be reduced to checking these properties on several four-node topologies. In our own prior work [17], we have verified route authenticity on variants of S-BGP using a combination of manual proofs and an automated tool, Proverif [12]. The modeling and analysis in these works are specific to the protocols and the route authenticity properties.

Some of the properties that we verify in our case study are similar. We propose a general framework for leveraging a declarative programming language for verification and empirical evaluation of routing protocols. The program logic proposed in Chapter 3 can be used to verify generic safety properties of SANDlog programs.

6.1.3. Control-plane and Data-plane Verification.

In recent years, formal verification has been applied to both the control-plane (e.g., routing protocols) and data-plane (e.g., packet forwarding) of networks. There has been a cloud of prior work on network verification focusing on several different aspects. One aspect is the verification of network configurations, where the proposed solutions detect network configuration errors either 1) through static analysis of the configuration files [33, 4, 32, 62, 87], or 2) by analyzing snapshots of the data plane—reflecting the aggregate impact of all configurations—during system execution [48, 47, 56, 90]. These solutions rely heavily on application-specific network models and property specifications, which limits its adoption in more general scenarios.

One special case of network verification is SDN verification [9, 13, 50, 3, 42, 69, 78]. For example, VeriCon [9] defines its own special language for modeling SDN controller and switches. A Hoare-logic is developed on this language to prove properties of SDN controllers. The proof obligations are translated to constraints and solved by the SMT solver. NICE is a testing tool for SDN controllers written in Python [13]. NICE combines symbolic execution of the controller programs with state-exploration-based model checking.

An alternative approach is to verify network configurations generated by SDN controllers in realtime, instead of verifying the protocols directly [50, 56]. For instance, Anteater reduced SDN data plane verification into SAT problems so that SAT solvers can solve them effectively in practice [56]. NetKAT is a high-level language designed specifically for programming SDN. Its semantics are based on Kleene algebra. The correctness properties of networks programming using NetKAT are tightly connected to the semantics of Kleene

algebra, for instance, reachability, way points and traffic separation. All of the above tools are specially designed to analyze SDN controllers or data planes.

6.2. Runtime Analysis of Distributed Systems

Next, we summarize the works that allow for runtime analysis of distributed systems.

6.2.1. System Monitoring

There are a number of research works that monitors the execution of the distributed system. The monitoring allows the system developer to understand the key behavior of the distributed system, and to diagnose the problem in a more effective way. For example, Gunter et al. [41] proposes to dynamically monitor the distributed system by instrumenting network devices in a light-weighted manner. Chen et al. [24] records runtime paths that requests follow in a distributed system to manage failure and evolution of the system. P2 Monitor [77] develops an application logging, monitoring and debugging facility built on top of P2 system. D3S [52] allows the developers to specify prediate over the distributed system, and would output a sequence of state changes when failure occurs.

6.2.2. Message Logging

Message logging is a well-studied area for analyzing distributed systems. Aguilera et al. [2] uses message-level traces to debug performance in distributed systems with black boxes. Netlogger [80] records the event logs in a distributed system to provide detailed end-to-end application and system-level monitoring. Pip [72] proposes infrastructure that allows the user to compare his/her expected behavior of the distributed system with the actual behavior that is recorded through instrumentation of the system. Liblog [37] records log execution for C/C++ applications in a distributed system, and deterministically replay the logs to help debug the system. Google has also proposed Dapper [76], which is a tracing infrastructure that help debug the distributed system. X-trace [34] designs a tracing framework that provides a comprehensive view of the distributed system. Friday [36] combines

the deterministic replay with symbolic debugging. WiDS Checker [53] uses simulation to replay the execution of the distributed system, and allows the user to specify high-level properties which are fed to the checker for verification.

6.2.3. Network Provenance

Network provenance is an area which draws the attention of researchers recently. Compared to message logging, network provenance explicitly captures the causality relationship among events in the distributed system, providing a more convenient tool for the user to debug the system.

Network provenance has been proposed and developed by ExSPAN [95] and DTaP [94]. These two proposals apply the concept of data provenance to the networking field to support network diagnosis and forensics. In Chapter 5, we adopted ExSPAN's provenance tree model as a starting point, and developed compression schemes based on the model. Secure Network Provenance [92] enables network provenance in an adversarial environment, where misbehaving nodes can be detected even if they lie. Negative provenance [85] extends the network provenance to explain the missing event in the network. There are a number of works that have already used network provenance to systematically debug the system. Differential provenance [16] debugs the distributed system by identifying the difference between a faulty provenance tree and a reference provenance tree. Meta Provenance [84] proposes to debug the software of network devices with the help of provenance. Our compression scheme is expected to work on all the systems that support network provenance.

6.2.4. Provenance Compression

In database literature, a number of works have considered optimization of provenance storage. However, we differ significantly in our design due to the distributed nature of our target environment. We briefly list a few representative bodies of work, and explain our differences.

Woodruff *et al.* [83] reduce storage usage for maintaining fine-grained lineage (i.e., provenance) by computing provenance information dynamically during query time through invertible functions. Their approach makes tradeoff between storage efficiency and accuracy of provenance. On the other hand, our approach requires no such tradeoff, achieving the same level of accuracy as queries on uncompressed provenance trees.

Chapman *et al.* [15] develop a set of factorization algorithms to compress workflow provenance. Their proposal does not consider a distributed setting. For example, node-level factorization (combining identical nodes) requires additional states to be maintained and propagated from node to node during provenance maintenance to resolve potential ambiguities. Maintaining and propagating these states can lead to significant communication overhead in a distributed environment. In contrast, our solution uses equivalence keys to avoid comparing provenance trees on a node-by-node basis, and hence minimizes communication overhead during provenance maintenance.

Our compression technique implicitly factorizes provenance trees at runtime before removing redundant factors among trees in the same equivalence class. Olteanu *et al.* [65][66] propose factorization of provenance polynomials for conjunctive queries with a new data structure called factorization tree. Polynomial factorization in [66] can be viewed as a more general form of the factorization used in the equivalence-based compression proposed in Chapter 5. If we encode the provenance trees of each packet as polynomials, the general factorization algorithm in [66], with specialized factorization tree, would produce the same factorization result in our setting. Our approach is slightly more efficient, as we can skip the factorization step by directly using equivalence keys at runtime to group provenance trees for compression. Exploring the more general form of factorization in [66] for provenance of distributed queries is an interesting avenue of future work.

ProQL [46] proposes to save the storage of *single* provenance tree by (1) using primary keys to represent tuples in the provenance, and (2) maintaining one copy for attributes of the same values in a mapping (rule). These techniques could also be applied alongside our

online compression algorithm to further reduce storage. ProQL does not consider storage sharing *across* provenance trees. Amsterdamer *et al.* [5] theoretically defines the concept of *core provenance*, which represents derivation shared by multiple equivalent queries. In our scenario, the shared provenance tree of each equivalence class can be viewed as *core provenance*.

Xie *et al.* [86] propose to compress provenance graphs with a hybrid approach combining Web graph compression and dictionary encoding. Zhifeng *et al.* [10] proposes rule-based provenance compression scheme. Their approaches on a high level compresses provenance trees to reduce redundant storage. However, these approaches require knowledge of the *entire* trees prior to compression, which is not practical, if not impossible, for distributed provenance.

CHAPTER 7

Future Work

In addition to the work that has been done, we propose future work that aims at providing a more complete framework that allows for unified static analysis and runtime analysis of declarative distributed systems. The future work is mainly composed of two parts:

- **A more complete framework for provenance compression.** To make our framework more complete, we intend to add more features into our current framework, along with more evaluation on additional applications.
- **Optimization of static analysis.** We plan to optimize static analysis by designing algorithms that could incrementally maintain derivation instances for evolving NDlog programs.

7.1. A More Complete Framework of Provenance Compression

Though our provenance compression scheme has already achieved significant storage reduction for packet forwarding and DNS resolution applications, we try to make the framework more complete by (1) allowing the user to dynamically specify the equivalence relation of provenance trees, and (2) performing more evaluation over the prototype we have now. We introduce these two tasks in more detail below.

7.1.1. Dynamic Definition of Equivalence Relation

When we perform the provenance compression now, the equivalence relation for provenance trees is pre-defined. This definition, though effective in reducing storage space, could still incur large storage overhead under certain circumstances. For example, consider a slow-changing table T (e.g., a routing table). Two provenance trees that use different tuples in T, but share the rest of the provenance nodes would not be considered equivalent, given our

current definition of the equivalence relation. However, if we exclude T from the definition of the equivalence relation, the above two provenance trees can be viewed as equivalent and got compressed.

Therefore, we intend to support such dynamic definition of the equivalence relation, so that users could specify their own equivalence relations that mostly meet their needs. We expect the new design to further compress provenance trees that differ in *a portion of* provenance nodes, rather than only one *single* provenance node.

The change of design could possibly make provenance maintenance and query more complicated, as it now requires nodes in the distributed system to selectively maintain provenance information – i.e., these nodes need to compress provenance nodes involved in the definition of the equivalence relation, but not those not involved. We expect to explore the new design more carefully in the future.

We also plan to evaluate the storage overhead using the dynamic definition of the equivalence relation, and compare it with the static definition, over a few popular networking applications, such as packet forwarding.

7.1.2. Provenance Maintenance for SDN Applications

The two applications we are evaluating now – i.e., packet forwarding and DNS resolution – are relatively simple, in that their specification involves no more than five NDlog rules. To show the generality of our solution, we plan to apply the compressed provenance to applications in software-defined networking, such as Ethernet MAC learning or Ethernet address resolution. A typical SDN-enabled distributed system involves multiple types of network devices, including a controller, a number of bare-metal switches (e.g., Openflow switches) and end hosts. Each network device will be modeled in NDlog rules and the provenance information will be collected and compressed for packets traversing the network, as well as control-plane messages (e.g., ARP messages).

We will not only apply our solution to the selected SDN application, but also redo all the evaluation for the new application. This includes measurement of the provenance storage (growth), the bandwidth utilization and the query latency.

7.1.3. A Cost Model for Selection between Alternative Solutions

The experimental results In Section 5.6.1 show that the reactive maintenance strategy of DTaP [94] could save more storage than our online compression scheme, sacrificing a certain amount of time during querying. As a result, the user needs to make tradeoff in reality when selecting either solution. A rigorous cost model could ease the pain of making a decision as to which strategy to take. The cost model is expected to take into account several factors: (1) frequency of querying, where the online compression scheme is favored when the query frequency is high; (2) storage availability, where DTaP is preferred when storage is scarce. (3) bandwidth availability, where DTaP consumes less bandwidth, and (4) cost of replay, where the online compression scheme is a winner if replay is costly – e.g., no duplication system exists.

7.2. Optimization of Static Analysis

We intend to optimize computation of the derivation pool in Chapter 4. Specifically, we plan to develop a new algorithm that incrementally maintains the derivation pool when new rules are added to a NDlog program.

7.2.1. Incremental Maintenance of the Derivation Pool

The symbolic execution of NDlog programs in Chapter 4 would generate a derivation pool which contains all possible derivations of each relation in the program. Since the derivation pool is large, it is slow and unnecessary to generate the pool from scratch for every program, especially when one program is an evolved version of another program. For example, in the application of Ethernet Address Resolution (Section 4.4). the user might want to add a rule for access control, so that only an authorized machine is allowed to query the MAC address

of another one. Though the new program only differs in one rule from the old program, the derivation pool has to be re-generated all over again with our current solution, which is inefficient.

We would like to explore the possibility of maintaining the derivation pool incrementally, when new rules are added into the program. This potentially requires a new algorithm that takes as input an existing NDlog program *prog*, the delta rules rs and the derivation pool of *prog*, and outputs a new derivation pool corresponding to $prog \cup rs$.

7.2.2. Evaluation of the Incremental Maintenance Algorithm

We intend to prove correctness of the incremental maintenance algorithm and analyze its time complexity. We would also empirically evaluate the time that is needed for the algorithm to re-compute the derivation pool, and compare the results with the old algorithm where the derivation pool is re-computed from scratch.

CHAPTER 8

Conclusion

In this dissertation, we have shown that STRANDS could not only ensure correctness of distributed systems, but, in the case of failure, help network administrators debug the system. We also demonstrate that static analysis and runtime analysis could aid each other to achieve better performance during system analysis.

We have designed a program logic for verifying secure routing protocols specified in the declarative language SANDlog. We have integrated verification into a unified framework for formal analysis and empirical evaluation of secure routing protocols.

To better automate the verification process, we presented a symbolic execution-based approach to analyze and debug network protocols using declarative networking. By focusing on a specific class of safety properties, we are able to analyze NDlog programs with few annotations. Our algorithm reduces property checking to constraint solving that can be automatically checked by SMT solvers (e.g., Z3). We analyzed formal properties of our algorithms and implemented a prototype tool on top of RapidNet, a compilation and execution framework for NDlog. Using our tool, we analyzed a number of real-world SDN network protocols. Our tool can unveil problems ranging from software bugs, incomplete topological constraints, and incorrect property specification. When a given safety property is violated, our tool can provide meaningful counterexamples to help debug the protocol specification.

Finally, we enable the users of distributed systems to deploy storage-efficient distributed provenance, faciliting root cause analysis during system failure. We propose an online compression scheme that compresses distributed network provenance during the execution of network applications. Our work is motivated by network settings, where the large volume of events necessitate compression techniques, and existing centralized approaches do not

work. We define an equivalence relation between provenance trees, and come up with a compile time static analysis phase for determining equivalence keys attributes that can be used for grouping provenance tress together. At runtime, through a rule rewrite process, our network implementation maintains and stores only one concrete copy for provenance trees that are shared by all the members in the equivalence class. Our evaluation results show that the compression scheme saves storage significantly, incurs little network overhead, and allows for efficient provenance query.

Bibliography

[1] Wireshark. https://www.wireshark.org/.

[2] Marcos Kawazoe Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 74–89, 2003.

[3] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *SafeConfig*, 2010.

[4] Ehab Al-Shaer and Hazem Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM*, 2004.

[5] Yael Amsterdamer, Daniel Deutch, Tova Milo, and Val Tannen. On provenance minimization. *ACM Trans. Database Syst.*, 37(4):30, 2012.

[6] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. Modeling and verifying ad hoc routing protocols. In *Proceedings of CSF*, 2010.

[7] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. Deciding security for protocols with recursive tests. In *Proceedings of CADE*, 2011.

[8] Christel Baier and Joost-Pieter Katoen, editors. *Principles of Model Checking*. The MIT press, 2008.

[9] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 31, 2014.

[10] Zhifeng Bao, Henning Köhler, Liwei Wang, Xiaofang Zhou, and Shazia Wasim Sadiq. Efficient provenance storage for relational queries. In *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1352–1361, 2012.

[11] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4), 2002.

[12] Bruno Blanchet and Ben Smyth. Proverif 1.86: Automatic cryptographic protocol verifier, user manual and tutorial. http://www.proverif.ens.fr/manual.pdf.

[13] Marco Canini, Daniele Venzano, Peter Peresíni, Dejan Kostic, and Jennifer Rexford. A NICE way to test openflow applications. In *Proceedings of the 9th USENIX Symposium*

on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012, pages 127–140, 2012.

[14] Chin-Liang Chang and Richard C. T. Lee. *Symbolic logic and mechanical theorem proving*. Computer science classics. Academic Press, 1973.

[15] Adriane Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 993–1006, 2008.

[16] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 115–128, 2016.

[17] Chen Chen, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Reduction-based security analysis of internet routing protocols. In *WRiPE*, 2012.

[18] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. June 2014.

[19] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. Technical report, CIS Dept. University of Pennsylvania, February 2014.

[20] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 117–132. Springer, 2014.

[21] Chen Chen, Harshal Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Provably correct distributed provenance compression (cmu-cylab-17-001). Technical report, CyLab, Carnegie Mellon University, Jan. 2017.

[22] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Distributed provenance compression. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 203–218, 2017.

[23] Chen Chen, Lay Kuan Loh, Limin Jia, Wenchao Zhou, and Boon Thau Loo. Automated verification of safety properties of declarative networking programs. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 79–90, 2015.

[24] Mike Y. Chen, Anthony Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. Path-based failure and evolution management. In *1st Symposium*

on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings, pages 309–322, 2004.

[25] CNET. How pakistan knocked youtube offline.

[26] Véronique Cortier, Jan Degrieck, and Stéphanie Delaune. Analysing routing protocols: four nodes topologies are sufficient. In *Proceedings of POST*, 2012.

[27] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol Composition Logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007.

[28] Ralph Droms. Dynamic host configuration protocol. 1997. RFC 2131.

[29] E. N. Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *Digest of Papers: FTCS/24, The Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing, Austin, Texas, USA, June 15-17, 1994*, pages 298–307, 1994.

[30] Dawson Engler and Madanlal Musuvathi. Model-checking large network protocol implementations. In *Proceedings of NSDI*, 2004.

[31] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL protocol analyzer: grammar generation. In *Proceedings of FMSE*, 2005.

[32] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *NDSI*, 2005.

[33] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[34] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.*, 2007.

[35] Deepak Garg, Jason Franklin, Dilsun Kaynar, and Anupam Datta. Compositional system security with interface-confined adversaries. *ENTCS*, 265:49–71, September 2010.

[36] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.*, 2007.

[37] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications (awarded best paper!). In *Proceedings of the 2006 USENIX*

Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006, pages 289–300, 2006.

[38] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003.

[39] Alwyn Goodloe, Carl A. Gunter, and Mark-Oliver Stehr. Formal prototyping in early stages of protocol design. In *Proceedings of ACM WITS*, 2005.

[40] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40, 2007.

[41] Dan Gunter, Brian Tierney, Keith R. Jackson, Jason Lee, and Martin Stoufer. Dynamic monitoring of high-performance distributed applications. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002), 23-26 July 2002, Edinburgh, Scotland, UK*, pages 163–170, 2002.

[42] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *HotSDN*, 2012.

[43] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 71–85, 2014.

[44] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Trans. Netw.*, 10(5):589–603, 2002.

[45] Hans W. Kamp. *Tense Logic and the Theory of Linear Order*. Phd thesis, Computer Science Department, University of California at Los Angeles, USA, 1968.

[46] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 951–962, 2010.

[47] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 99–111, 2013.

[48] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Symposium on Net-*

worked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012, pages 113–126, 2012.

[49] Stephen Kent, Charles Lynn, Joanne Mikkelson, and Karen Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 18:103–116, 2000.

[50] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 15–27, 2013.

[51] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[52] 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 *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 423–437, 2008.

[53] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. Wids checker: Combating bugs in distributed systems. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.*, 2007.

[54] 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 *SIGMOD*, 2006.

[55] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. In *Communications of the ACM*, 2009.

[56] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 290–301, 2011.

[57] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[58] P. V. Mockapetris. *Domain names - implementation and specification*, November 1987. RFC 1035.

[59] Shivkumar C. Muthukumar, Xiaozhou Li, Changbin Liu, Joseph B. Kopena, Mihai Oprea, and Boon Thau Loo. Declarative toolkit for rapid network protocol simulation and experimentation. In *SIGCOMM (demo)*, 2009.

[60] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazieres, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with ICING. In *Proceedings of CoNEXT*, 2011.

[61] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

[62] Timothy Nelson, Christopher Barratt, Daniel Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[63] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining distributed logic programs incrementally. In *Proceedings of PPDP*, 2011.

[64] ns 3 project. Network Simulator 3. http://www.nsnam.org/.

[65] Dan Olteanu and Jakub Závodný. On factorisation of provenance polynomials. In *Proceedings of TaPP*, 2011.

[66] Dan Olteanu and Jakub Závodný. Factorised representations of query results: size bounds and readability. In *Proceedings of ICDT*, pages 285–298, 2012.

[67] One Hundred Eleventh Congress. 2010 report to congress of the u.s.-china economic and security review commission, 2010.

[68] David C. Plummer. An ethernet address resolution protocol. 1982. RFC 826.

[69] P Porras, S Shin, V Yegneswaran, M Fong, M Tyson, and G Gu. A security enforcement kernel for openflow networks. In *HotSDN*, 2012.

[70] RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation. http://netdb.cis.upenn.edu/rapidnet/.

[71] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of ACM SIGCOMM*, pages 323–334, 2012.

[72] 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 *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings.*, 2006.

[73] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems (reprint). *Commun. ACM*, 26(1):96–99, 1983.

[74] Robert Ramey. http://www.boost.org/doc/libs/1_61_0/libs/serialization/doc/index.html.

[75] Secure BGP. http://www.ir.bbn.com/sbgp/.

[76] Benjamin H. Sigelman, Luiz AndrÃĺ Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[77] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using queries for distributed monitoring and forensics. In *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, pages 389–402, 2006.

[78] R. W Skowyra, A Lapets, A Bestavros, and A Kfoury. Verifiably-safe software-defined networks for cps. In *HiCoNS*, 2013.

[79] The Coq project. The Coq Proof Assistant. https://coq.inria.fr/.

[80] Brian Tierney, William E. Johnston, Brian Crowley, Gary Hoo, Christopher X. Brooks, and Dan Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, HPDC '98, Chicago, Illinois, USA, July 28-31, 1998.*, pages 260–267, 1998.

[81] Tao Wan, Evangelos Kranakis, and P. C. Oorschot. Pretty secure BGP (psBGP). In *Proceedings of 12th NDSS*, 2005.

[82] Russ White. Securing bgp through secure origin BGP (soBGP). *The Internet Protocol Journal*, 6(3):15–22, 2003.

[83] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, pages 91–102, 1997.

[84] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks, Philadelphia, PA, USA, November 16 - 17, 2015*, pages 26:1–26:7, 2015.

[85] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 383–394, 2014.

[86] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell D. E. Long. Evaluation of a hybrid approach for efficient provenance storage. *TOS*, 9(4):14, 2013.

[87] L Yuan, H Chen, J Mai, C. N. Chuah, Z Su, and P Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *SRSP*, 2006.

[88] Z3. http://z3.codeplex.com/.

[89] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proceedings IEEE INFOCOM '96, The Conference on Computer Communications, Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Networking the Next Generation, San Francisco, CA, USA, March 24-28, 1996*, pages 594–602, 1996.

[90] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, , and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *NSDI*, 2014.

[91] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. Scion: Scalability, control, and isolation on next-generation networks. In *Proceedings of Oakland S&P*, 2011.

[92] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 295–310, 2011.

[93] Wenchao Zhou, Yun Mao, Boon Thau Loo, and Martín Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *ICDE*, 2009.

[94] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary G. Ives, Boon Thau Loo, and Micah Sherr. Distributed time-aware provenance. *PVLDB*, 6(2):49–60, 2012.

[95] 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 the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 615–626, 2010.

[96] 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*, 2010.