Distributed Provenance Compression

Chen Chen
University of Pennsylvania
chenche@seas.upenn.edu

Harshal Tushar Lehri
University of Pennsylvania
halehri@seas.upenn.edu

Lay Kuan Loh
Carnegie Mellon University
lkloh@cmu.edu

Anupam Alur
University of Pennsylvania
aalur@seas.upenn.edu

Limin Jia
Carnegie Mellon University
liminjia@cmu.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Wenchao Zhou
Georgetown University
wzhou@cs.georgetown.edu

ABSTRACT

Network provenance, which records the execution history of network events as meta-data, is becoming increasingly important for network accountability and failure diagnosis. For example, network provenance may be used to trace the path that a message traversed in a network, or to reveal how a particular routing entry was derived and the parties involved in its derivation. A challenge when storing the provenance of a live network is that the large number of arriving messages may incur substantial storage overhead. In this paper, we explore techniques to dynamically compress distributed provenance stored at scale. Logically, compression is achieved by grouping equivalent provenance trees and maintaining only one concrete copy for each equivalence class. To efficiently identify the equivalent provenance, we (1) introduce distributed event-based linear programs (DELPs) to specify distributed network applications, and (2) statically analyze DELPs to allow for quick detection of provenance equivalence at runtime. Our experimental results demonstrate that our approach leads to significant storage reduction and query latency improvement over alternative approaches.

Keywords

Provenance; distributed systems; storage; static analysis

1. INTRODUCTION

Network administrators require the capability to identify the root causes of device malfunction and performance slowdowns in data centers or across wide-area networks, and also to determine the sources of security attacks. Such capabilities often utilize *network provenance*, which allows the user to issue queries over network meta-data about the execution history. In recent years, network provenance has been successfully applied to various network settings, resulting in proposals for distributed provenance [28], secure network provenance [26], distributed time-aware provenance [27] and negative provenance [23]. These proposals demonstrate that database-style declarative queries can be used for maintaining and querying distributed provenance at scale.

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 provenance compression in the database literature [3], the work was not designed for distributed settings. Our paper's contributions are:

System Model. We propose a new network programming model, which specifies *distributed event-based linear programs* (DELPs), using a restricted variant of the Network Datalog language in declarative networking [12]. 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 first technique relies on materializing only the tuples that the administrators are interested in. We propose a distributed querying technique that can reconstruct the entire provenance tree from the reduced provenance information that is maintained. Our second technique combines multiple provenance trees together, based on a notion of *equivalence classes*. In each equivalence class, the provenance trees are identical except a few pre-defined nodes. We compress these equivalent trees by maintaining only one concrete copy for the shared provenance part, along with the delta information for each indi-

SIGMOD '17, May 14–19, 2017, Chicago, Illinois, USA.

DOI: http://dx.doi.org/10.1145/3035918.3035926

vidual 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 [14]. 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 – i.e., packet forwarding and DNS resolution – and the performance results show that the compression techniques achieve orders of magnitude reduction in storage and significant reduction in query latency, with only negligible network overhead added to each monitored network application at runtime.

2. BACKGROUND

We first provide an introduction to *Network Datalog* (NDlog) [12], a declarative networking programming language we use to model network applications in the distributed system, then we introduce the concept of distributed network provenance [28, 27].

2.1 Network Datalog

$$\begin{array}{lll} r1 & \mathsf{packet}(@N,S,D,DT) & :- \ \mathsf{packet}(@L,S,D,DT), \\ & & \quad\ \mathsf{route}(@L,D,N). \\ r2 & \mathsf{recv}(@L,S,D,DT) & :- \ \mathsf{packet}(@L,S,D,DT), D == L. \end{array}$$

Figure 1: An NDlog program for packet forwarding

To illustrate NDlog, we show an example query (Figure 1) that recursively forwards packets in a network. A typical NDlog program is composed of a set of rules. Each rule takes on the format $\mathsf{p} :- \mathsf{q}_1, \mathsf{q}_2, ...\mathsf{q}_n$, where p is a relation called the rule head, and q_is are rule bodies that are either relational atoms, arithmetic atoms or user-defined functions. Relations and rules of an NDlog program can be deployed in a distributed fashion. To logically specify the location of each relation, an "@" symbol – called the location specifier – is prepended to the first attribute of each relation.

Each node in the network maintains a relational database storing base tuples (i.e., tuples that are input by the user) and/or derived tuples (i.e., tuples that are generated by the NDlog program). During program execution, when all the rule bodies of a rule r have corresponding tuples in the local database, r will be triggered, generating the head tuple. If the location specifier of the head tuple is different from that of the bodies (e.g., *r1* in Figure 1), the head tuple will be transmitted through the network to the remote node. In the example program of Figure 1, *r1* forwards a local packet (packet) at a node L to a node N by looking up the packet's destination D in the local routing table (route). *r2* receives a packet and stores it locally in the recv table, if the packet is destined to the local node ($D == L$).

2.2 Distributed Network Provenance

Data provenance [8] can be used to explain why and how a given tuple is derived. Based on data provenance, prior work [28] also proposes network provenance, which faithfully records the execution of (possibly erroneous) applications in a (possibly misconfigured) distributed system. This allows

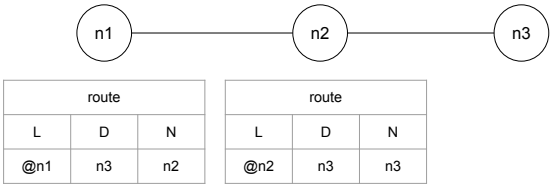

Figure 2: An example deployment of packet forwarding. Node *n1* and node *n2* has a local *route* table indicating routes towards node *n3*.

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 2. If the user prefers the routing with the shortest paths, the routing entry of *n1* in Figure 2 would have been erroneous – a correct entry should be $\mathsf{route}(@n1, n3, n3)$. 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 3 shows the provenance tree of a tuple $\mathsf{recv}(@n3, n1, n3, \text{"data"})$. This provenance tree records the traversal of $\mathsf{packet}(@n1, n1, n3, \text{"data"})$ from node *n1* to *n3* in Figure 2. 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 3) stand for the rules that are triggered in the program execution, while the tuple nodes (i.e., the square nodes in Figure 3) 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 [10] 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 [28], a representative distributed provenance engine, maintains the provenance information in a distributed relational database. There are two (distributed) tables in the 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 1 shows an example distributed database storing the provenance tree in Figure 3. 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 $\mathsf{recv}(@n3, n1, n3, \text{"}data\text{"})$ (Figure 3), ExSPAN first computes the hash value *vid6* of the tuple, and uses *vid6* to find the tuple $\mathsf{prov}(n3, vid6, rid3, n3)$ in the prov table. ExSPAN further uses the values *rid3* and *n3* to locate $\mathsf{ruleExec}(n3, rid3, r2, (vid5))$ 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			
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 1: Relational tables (ruleExec and prov) maintaining the provenance tree in Figure 3.

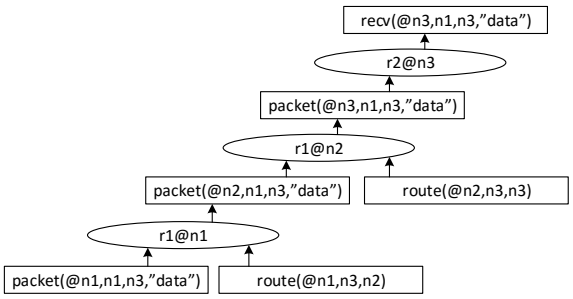

Figure 3: A distributed provenance tree for the execution of packet(@$n1, n1, n3,$ "$data$"), which traverses from node $n1$ to node $n3$ in Figure 2.

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.

2.3 Motivation for Provenance Compression

A key problem not addressed in prior work on network provenance [28][27] is that the provenance information 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 2, 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 3. 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.

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 2, 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.

The key challenge of provenance compression in a distributed system is to achieve significant storage saving while incurring low network overhead (e.g., extra bandwidth and computation), and still enabling the user to query the provenance information effectively as does uncompressed provenance. Hence, we avoid content-level compression techniques such as gzip, but opt for the conservative compression based on the structure of provenance trees.

3. MODEL

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 2 is an example distributed system with three nodes.

3.1 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), 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 *can be specified in the form:* $[head] :- [event], [conditions]$*, where* $[event]$ *is a body relation designated by the programmer, and* $[conditions]$ *are all non-event body atoms.*
- 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 the network states, which change slowly compared to the fast rate of incoming events. For example, in the packet forwarding program, 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 configurations are updated at runtime and packets see only either the old or new configuration version across routers, as shown in prior work [19] 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 [11] – 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 1), Domain Name System (DNS) resolution [13], Dynamic Host Configuration Protocol (DHCP) [7] and Address Resolution Protocol (ARP) [18].

3.2 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 [27], by only maintaining non-deterministic input tuples, and replaying the whole system execution to re-construct the provenance information of the tuples of less interest during querying.

As with prior work [28], 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.

4. 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. Simply put, for each provenance tree, we remove its provenance nodes representing the intermediate event tuples. Figure 4 shows an optimized provenance tree tr' of the tree in Figure 3. The (distributed) relational database maintaining tr' is shown in Table 2, where *vid* values and *rid* values are identical to those in Table 1.

Compared to Table 1, Table 2 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 intermediate tuples or the base tuples.
- Two extra columns **NLoc** and **NRID** are added to the *ruleExec* table. They help the recursive query find the child node for each provenance node in the tree.

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

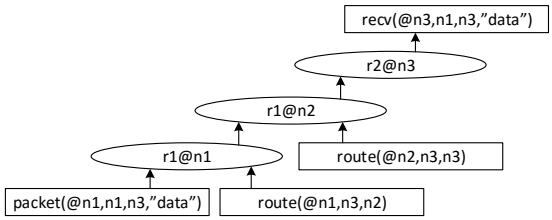

Figure 4: An optimized provenance tree for the tree in Figure 3.

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 2: Optimized ruleExec and prov tables for the provenance tree in Figure 4.

of intermediate tuples, as is common in typical networking scenarios. We use the querying of recv($@n3, n1, n3,$ "*data*")'s provenance in Table 2 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 2. 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 4. 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 3 .

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 6.1.3.

5. EQUIVALENCE-BASED COMPRESSION

The storage optimization described in Section 4 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.

5.1 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. The formal definition of $tr \sim tr'$ can be found in Appendix A. In our packet forwarding example, the provenance tree generated by a new event packet($@n1, n1, n3$, "url") (with "url" as its payload) is equivalent to the tree in Figure 4.

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 1), 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 5.2); and (2) an online provenance compression algorithm, which maintains the shared provenance tree for each equivalence class in a distributed fashion (Section 5.3).

Correctness of using event equivalence for determining provenance tree equivalence is shown in Theorem 1. The proof will be discussed in Section 5.2.

Theorem 1 (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.* $tr_1 \sim tr_2$ *(*$tr_1' \sim tr_2'$*).*

5.2 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.

```
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 5: Pseudocode to identify equivalence keys

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. We refer interested readers to Appendix C for an example dependency graph of the packet forwarding program.

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 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 1); (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 1); (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 1); 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 1 were to be redefined as $r2'$ recv($@L, S, N, DT$) :− 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 5). 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 1. We use predicate joinSAttr($p{:}n$) 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 joinFAttr($p{:}n, q{:}m$). We further use predicate joinFAttr($p{:}n, q{:}m$) to inductively define 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 1 by proving Lemma 2, a stronger lemma that gives us Theorem 1 as corollary. In Lemma 2, we write $tr : P$ to denote that tr is a provenance tree of the output tuple P, and write $DQ, \mathcal{DB}, ev \vDash tr : P$ to mean that tr is generated by executing the program DQ over a database $\mathcal{DB}$, triggered by the event tuple ev.

Lemma 2 (Correctness of equivalence keys (Strong))**.**
If GetEquiKeys$(G, ev) = K$ *and* $ev_1 \sim_K ev_2$
and $DQ, \mathcal{DB}, ev_1 \vDash tr_1 : p(t_1, ..., t_n)$,
then $\exists tr_2 : p(s_1, ..., s_n)$ *s.t.* $DQ, \mathcal{DB}, ev_2 \vDash tr_2 : p(s_1, ..., s_n)$
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.* $DQ \vdash \mathsf{connected}(ev{:}\ell, p{:}i)$ *and* $(ev{:}\ell) \notin K$.

Intuitively, Lemma 2 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 (Appendix B) of Lemma 2 over the structure of the trees.

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 $O(n^2)$.

5.3 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, the 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 querying.

To illustrate this, Figure 6 presents an example consisting of two packets traversing the network topology (from *n1* to *n3*) in Figure 2. $\mathsf{packet}(@n1, n1, n3, \text{“data”})$ is first inserted for execution (represented by the solid arrows), followed by the execution of $\mathsf{packet}(@n1, n1, n3, \text{“url”})$ (represented by the dashed arrows). The three stages of online compression are logically separated with vertical dashed lines. Table 3 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 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 6, when the first packet tuple $packet(@n1, n1, n3, \text{“data”})$ 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 $packet(@n1, n1, n3, \text{“url”})$ 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 6, when $\mathsf{packet}(@n2, n1, n3, \text{“data”})$ 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 4. In comparison, when $\mathsf{packet}(@n2, n1, n3, \text{“url”})$ 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 use a hash table *hmap* 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 6, 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 6, the first execution generates the output tuple $\mathsf{recv}(@n3, n1, n3, \text{“}data\text{”})$, which is associated to the reference $(n3, rid1)$. This is reflected by the tuple $\mathsf{prov}(n3, tid1, n3, rid1, evid1)$ in the prov table (Table 3). *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

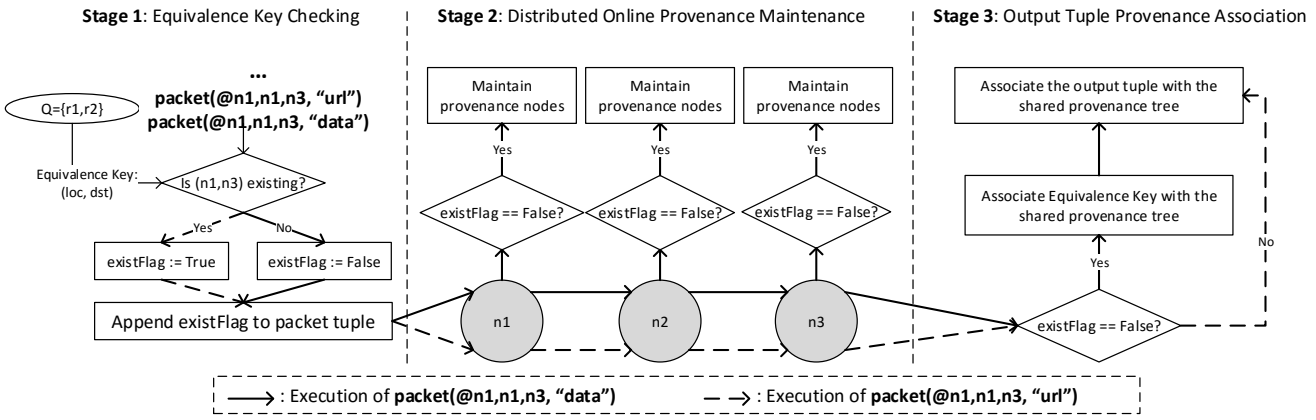

Figure 6: An example execution of the packet forwarding program in Figure 1. The program is first triggered by packet($@n1, n1, n3,$ "data"), followed by packet($@n1, n1, n3,$ "url").

ruleExec					
Loc	**RID**	**RULE**	**VIDS**	**NLoc**	**NRID**
n3	**rid1**(sha1(r2))	r2	NULL	n2	**rid2**
n2	**rid2**(sha1(r1,*vid1*))	r1	(*vid1*(sha1(route($@n2, n3, n3$))))	n1	**rid3**
n1	**rid3**(sha1(r1,*vid2*))	r1	(*vid2*(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 3: a ruleExec table and a prov table for compressed provenance trees produced in Figure 6

in the ruleExec and prov tables contain the exact same set of provenance trees that would have been derived by semi-naïve evaluation [11] without compression (Theorem 3). 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 [5]. We also define a set of transition rules of form: $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$ 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 tr. Likewise, given a provenance tree tr, we can also find an identical tree $\mathcal{P}$ encoded as entries in the ruleExec and prov tables. This correspondence is denoted as $tr \sim_d \mathcal{P}$ and can be defined by induction over the structure of provenance trees.

Theorem 3 (Correctness of Compression)**.** *$\forall n \in \mathbb{N}$ and an initial state $\mathcal{C}_{init}$, if $\mathcal{C}_{init} \rightarrow_{SN}^{n} \mathcal{C}_{sn}$, then $\exists \mathcal{C}_{cm}$ s.t. $\mathcal{C}_{init} \nearrow_{CM}^{n} \mathcal{C}_{cm}$ and for any provenance tree $tr \in \mathcal{C}_{sn}$, there exists a provenance tree $\mathcal{P} \in \mathcal{C}_{cm}$ s.t. $tr \sim_d \mathcal{P}$ and for any provenance tree $\mathcal{P} \in \mathcal{C}_{cm}$, there exists a provenance tree $tr \in \mathcal{C}_{sn}$ s.t. $tr \sim_d \mathcal{P}$. 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 3 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 3, we show Lemma 4 which implies Theorem 3 as corollary. Lemma 4 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 4 (Compression Simulates Semi-naïve Evaluation)**.** *$\forall n \in \mathbb{N}$ and an initial state $\mathcal{C}_{init}$, if $\mathcal{C}_{init} \rightarrow_{SN}^{n} \mathcal{C}_{sn}$, then $\exists \mathcal{C}_{cm}$ s.t. $\mathcal{C}_{init} \nearrow_{CM}^{n} \mathcal{C}_{cm}$ 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. $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$ 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 $tr \in \mathcal{C}_{sn}$ s.t. $tr \sim_d \mathcal{P}$, and vice versa.

Proof details of Lemma 4, along with the formal definition of the bisimulation relation $\boldsymbol{\mathcal{R}_{\mathcal{C}}}$, are presented in Appendix D. Briefly, we apply induction over n, the number of steps taken by the execution. The key is to show that if two bismiluar states both take one step, the resulting states are still bisimular.

Generality of equivalence-based compression. The idea of equivalence-based compression 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 5.1 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 Inter-Equivalence Class Compression

The online compression scheme introduced in Section 5.3 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 6 for execution. The produced provenance tree *prov* shares the provenance nodes *rid1* and *rid2* in the ruleExec table of Table 3. 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 4). 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 4: The ruleExecNode table and the ruleExecLink table replacing the ruleExec table in Table 3 to allow for compression of the shared rule execution nodes.

5.5 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 7 presents an example scenario based on Figure 2, 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, such as route($@n1, n3, n2$) in Figure 7, does not affect the stored provenance, as provenance information is monotone – that is, it represents the execution history which is immutable [27].

However, when a tuple *tp* is inserted into a slow-changing table, such as route($@n1, n4, n3$) in Figure 7, the provenance tree generated by *tp* could be incorrect or missing. For example, in Figure 7, after route($@n1, n4, n3$) is inserted, the

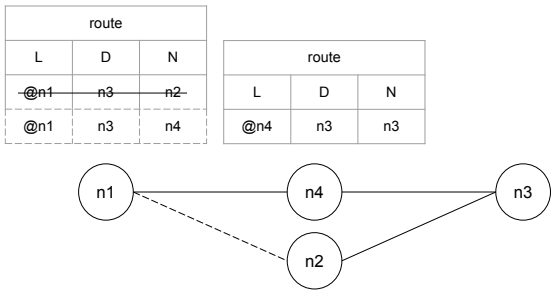

Figure 7: An updated topology of Figure 2. A new node *n4* is deployed to reach *n3*. The route table of *n1* is updated to forward packets to *n4* now.

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 5.3). Therefore, provenance trees will be maintained again for all equivalence classes. In Figure 7, 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 6.1.2.

5.6 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 recursive query reaches a ruleExec tuple at node *n* that has $(NULL, NULL)$ for (**NLoc**, **NRID**), the tagged *evid* is used to retrieve the event tuple materialized at *n*.

For example, in Table 3, 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 *tr* based on these entries. The

pseudocode of QUERY can be found in Figure 18 of Appendix E. 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 3 has only one derivation for the output tuple, so we return a singleton set.

Correctness of Querying. From the correctness of the online compression algorithm (Theorem 3), we can prove that all the provenance trees generated by semi-naïve evaluation can be queried and the query algorithm will return the correct provenance tree. One subtlety is that the compression algorithm may propagate updates out of order, causing ruleExec entries to be referred to in a provenance tree before being stored. We handle this subtlety by assuming all updates are processed before querying.

Theorem 5 (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 $\forall tr$:P in the output provenance storage of $\mathcal{C}_{sn}$
s.t. $\mathsf{hash}(\text{EVENTOF}(tr)) = evid$*,*
$\exists \mathcal{M}$ s.t. $\text{QUERY}(\mathcal{C}_{cm}, P, evid) = \mathcal{M}$ and $tr \in \mathcal{M}$
and $\forall tr' \in \mathcal{M} \backslash tr$, tr' is a proof of P stored in $\mathcal{C}_{sn}$
and $\mathsf{hash}(\text{EVENTOF}(tr')) = evid$*.*

Details of the proof are in Appendix E. Briefly, by Theorem 3, there exists $\mathcal{C}_{sn}$ s.t. $\mathcal{C}_{init} \rightarrow^{n}_{SN} \mathcal{C}_{sn}$ and $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$. By $\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{C}}}\ \mathcal{C}_{cm}$, we know that for any provenance tree tr of tuple P in $\mathcal{C}_{sn}$, there exists a prov tuple in $\mathcal{C}_{cm}$ that stores the reference to a provenance tree $\mathcal{P}$ for P, such that $tr \sim_d \mathcal{P}$. We induct over the depth of $\mathcal{P}$ to show that given the root of $\mathcal{P}$, the recursive lookup will return $\mathcal{P}$. Now, it is straightforward to reconstruct tr from $\mathcal{P}$ as the return value of QUERY.

6. EVALUATION

We have implemented a prototype based on enhancement to the RapidNet [14] declarative networking engine. At compile time, we add a program rewrite step that rewrites each DELP 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 compare three techniques for maintaining distributed provenance. The first is ExSPAN [28], 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 4). The third is the provenance maintenance using equivalence-based compression (Section 5). In the evaluation, we refer to the three techniques as *ExSPAN*, *Basic*, and *Advanced* respectively.

Workloads. Our experiments are carried out on two classic network applications: packet forwarding (Section 2) 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 a 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 involves larger messages along different paths in a graph, while DNS resolution involves smaller messages on a tree-like topology.

Testbed. In our experiment setup, we write the packet forwarding and DNS resolution applications in DELP, and use our enhanced RapidNet [14] 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 [15] 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, (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 [20] to serialize C++ data structures into binary data. At the end of each experiment run, we serialize the per-node provenance tables (i.e., the ruleExec table and the prov table) into binary files, and measure the size of files to estimate the storage overhead.

6.1 Application #1: Packet Forwarding

Our first set of results is based on the packet forwarding program in Figure 1. The topology we used for packet forwarding is a 100-node transit-stub graph, randomly generated by the GT-ITM [25] topology generator. In particular, there are four transit nodes – i.e., nodes through which traffic can traverse – 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 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[12]. The routes are stored in the route tables at each node in p.

6.1.1 Storage of Provenance Trees

Figure 8 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

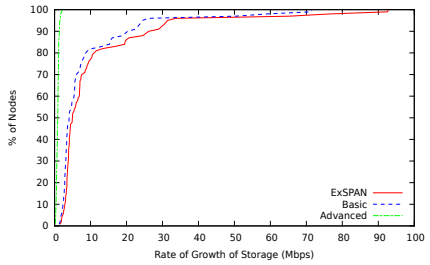

Figure 8: Cumulative growth rate of provenance with 100 pairs of communicating nodes, at input rate of 100 packets/second.

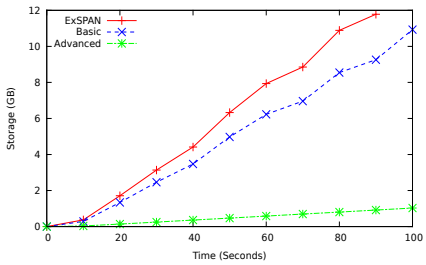

Figure 9: Provenance storage growth of all nodes, with input rate of 100 packets/second for 100 pairs of communicating nodes.

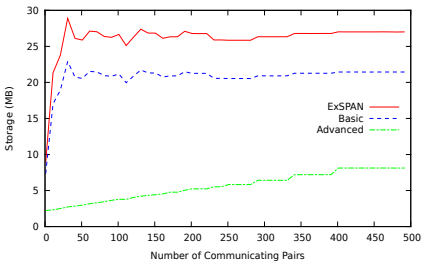

Figure 10: Provenance storage usage with 2000 input packets evenly distributed among increasing number of communicating pairs.

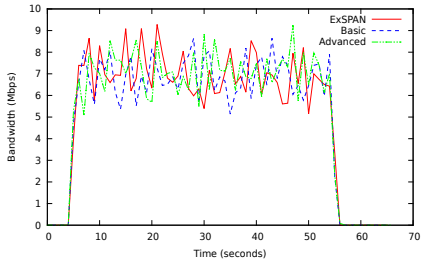

Figure 11: Bandwidth consumption during packet forwarding, with 500 pairs of nodes, each transmitting 100 packets.

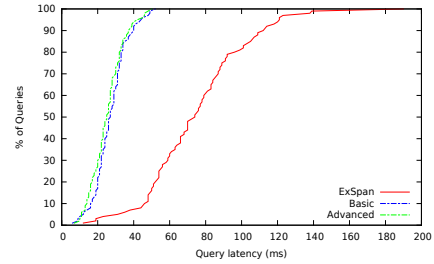

Figure 12: Cumulative distribution of provenance querying latency for 100 random queries with 100 pairs of communicating nodes.

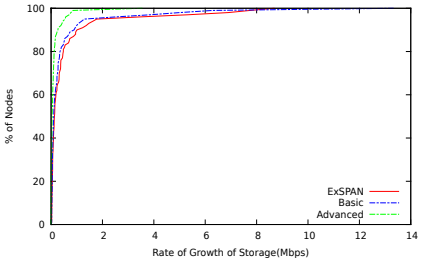

Figure 13: Cumulative provenance storage growth rate of nameservers with input request at a rate of 1000 requests/second.

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 nodes in their paths. As expected, Basic has smaller storage growth rate compared to ExSPAN, as it removes intermediate packet tuples from the provenance tables of each node. Advanced significantly outperforms the other two: all the nodes in the topology has 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 of all the traversing packets.

Figure 9 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. We further calculate the average growth rate for all three lines. ExSPAN's storage grows at 131 MB/second, Basic at 109 MB/second, and Advanced at 10.3 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.

Figure 10 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 a path length shorter than the average path length in the topology, thus incurring less storage overhead.

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. Despite the storage increase, Advanced still consumes much less storage space than the other two schemes.

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 – i.e., a 92% reduction over ExSPAN.

6.1.2 *Network Overhead.*

Figure 11 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 the ones of ExSPAN and Basic. 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 or-

der to study the effects of updates to slow-changing tuples. We observe a negligible bandwidth increase of 0.6%.

6.1.3 Query Latency

To evaluate the latency of queries, we ran emulation that could 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 of a recv tuple corresponding to a random source and destination pair – the destination node is the starting point of the query. The query is executed in a distributed fashion as described in Section 5.6. Based on our physical network topology, each query takes 5.3 hops on average in the network. We repeated the experiment for Basic, Advanced, and ExSPAN for 100 queries each.

Figure 12 shows our experimental results in the form of a CDF. We observe that both Basic and Advanced have latency numbers that are significantly lower than that of ExSPAN. For example, the mean/median for ExSPAN is 75ms and 74ms respectively, as compared to only 25.5ms and 25ms for Basic. This is approximately a 3X reduction in latency time. The extra overhead is due to ExSPAN's need in processing larger intermediate tuples. Basic and Advanced avoid this overhead by symbolically rederiving intermediate results during query execution.

6.2 Application #2: DNS Resolution

DNS resolution [13] 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 (see Appendix F). During the execution of each DNS program, provenance support is enabled so that the history DNS requests can be queried.

We synthetically generated the hierarchical network of DNS name servers. In total, there 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 [9] has shown that in reality, the requested domain names satisfy Zipfian distribution. In our experiments, we adopted the same distribution.

6.2.1 Storage of Provenance Trees

Figure 13 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 13 is not as significant as that in the packet forwarding experiments (Figure 8). 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 16 shows the provenance storage growth for all name servers. We record the current storage growth rate at 10-second intervals. In Figure 16, the storage of ExSPAN and Basic grows much faster than that of Advanced. Specifically, the growth rates of ExSPAN, Basic and Advanced are 13.15 Mbps, 11.57 Mbps and 3.81 Mbps respectively, and the storage space at 100 seconds reaches 1.32 GB, 1.16 GB, and 0.38 GB respectively. 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 14 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 14, 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 to ExSPAN and Basic. Unless a URL is only requested once (highly unlikely in reality), which is the worst possible case for Advanced, Advanced always performs better than ExSPAN and Basic.

6.2.2 Network Overhead

Figure 15 shows the bandwidth usage with elapsed time when we continuously sent 100,000 requests to the root nameserver. All three experiments finish within 102 seconds. Throughout the execution, ExSPAN and Basic have similar bandwidth usage, which is stable 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 experiments 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 additional bandwidth overhead.

7. RELATED WORK

Network provenance has been proposed and developed by ExSPAN [28] and DTaP [27]. These two proposals store un-

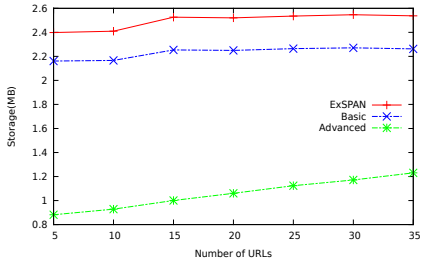

Figure 14: Provenance storage growth with increasing URLs, with 200 requests sent in total.

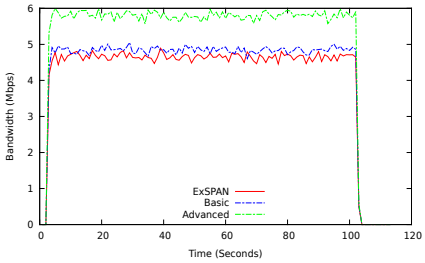

Figure 15: Bandwidth consumption for DNS resolution with 100,000 DNS requests.

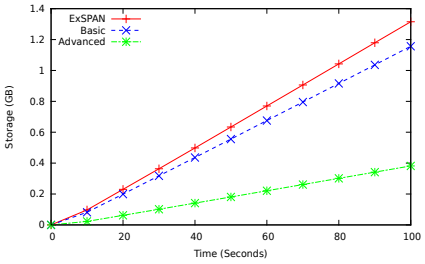

Figure 16: Provenance storage growth with continuous input requests at 1000 requests/second.

compressed provenance information, laying the foundation for our work. 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.* [21] reduce storage usage for maintaining fine-grained lineage (i.e., provenance) by computing provenance information dynamically during query time through invertible functions. Their approach tradeoffs storage with 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.* [3] 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 the 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.* [16][17] propose factorization of provenance polynomials for conjunctive queries with a new data structure called factorization tree. Polynomial factorization in [17] can be viewed as a more general form of the factorization used in the equivalence-based compression proposed in this paper. If we encode the provenance trees of each packet as polynomials, the general factorization algorithm in [17], 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 the equivalence keys at runtime to group provenance trees for compression. Exploring the more general form of factorization in [17] for provenance of distributed queries is an interesting avenue of future work.

ProQL [10] 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.* [1] 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.* [24] propose to compress provenance graphs with a hybrid approach combining Web graph compression and dictionary encoding. Zhifeng *et al.* [2] 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.

Provenance has been applied to network repairing [23, 22, 4] where root-cause analysis is used to identify and fix configuration errors in networks. Network repairing is orthogonal to our work, but can benefit from our compression techniques to reduce the storage of provenance maintenance.

8. CONCLUSION & FUTURE WORK

In this paper, we propose an online, equivalence-based compression scheme for the maintenance of distributed network provenance. Equivalent provenance trees are identified at compile time through static analysis of the declarative program, whereas our runtime maintains only one concrete representative provenance tree for each equivalence class. Our evaluation results show that the compression scheme saves storage significantly, incurs little network overhead, and allows for efficient provenance query.

This paper focuses on compressing trees generated within one program executed at all nodes. In most network deployments, there may be multiple programs (or network protocols) running concurrently. As future work, we plan to explore the possibility of compressing provenance trees *across* programs that share execution rules.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback to help improve the paper. We also thank Ling Ding and Yang Li for comments on the work and proof-read of the paper. This work is jointly funded by NSF CNS-1513679, CNS-1218066, CNS-1065130, CNS-1513961, CNS-1453392 and CNS-1513734.

10. REFERENCES

[1] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. *ACM Trans. Database Syst.*, 37(4):30, 2012.

[2] Z. Bao, H. Köhler, L. Wang, X. Zhou, and S. W. Sadiq. Efficient provenance storage for relational queries. In *CIKM*, pages 1352–1361, 2012.

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

[4] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of ACM SIGCOMM*, Aug. 2016.

[5] C. Chen, L. Jia, H. Xu, C. Luo, W. Zhou, and B. T. Loo. A program logic for verifying secure routing protocols. In *Proceedings of FORTE*, pages 117–132, 2014.

[6] C. Chen, H. Lehri, L. K. Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou. Provably correct distributed provenance compression (cmu-cylab-17-001). Technical report, CyLab, Carnegie Mellon University, Jan. 2017.

[7] R. Droms. Dynamic host configuration protocol. 1997. RFC 2131.

[8] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of PODS*, pages 31–40, 2007.

[9] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Trans. Netw.*, 10(5):589–603, 2002.

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

[11] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking Language, Execution and Optimization. In *Proceedings of ACM SIGMOD*, 2006.

[12] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. In *Communications of the ACM*, 2009.

[13] P. V. Mockapetris. *Domain names - implementation and specification*, Nov. 1987. RFC 1035.

[14] S. C. Muthukumar, X. Li, C. Liu, J. B. Kopena, M. Oprea, and B. T. Loo. Declarative toolkit for rapid network protocol simulation and experimentation. In *SIGCOMM (demo)*, 2009.

[15] ns 3 project. Network Simulator 3. http://www.nsnam.org/.

[16] D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *Proceedings of TaPP*, 2011.

[17] D. Olteanu and J. Závodný. Factorised representations of query results: size bounds and readability. In *Proceedings of ICDT*, pages 285–298, 2012.

[18] D. C. Plummer. An ethernet address resolution protocol. 1982. RFC 826.

[19] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of ACM SIGCOMM*, pages 323–334, 2012.

[20] Robert Ramey. http://www.boost.org/doc/libs/1_61_0/libs/serialization/doc/index.html.

[21] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings of ICDE*, pages 91–102, 1997.

[22] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proceedings of HotNets*, pages 26:1–26:7, 2015.

[23] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *Proceeding of ACM SIGCOMM*, pages 383–394, 2014.

[24] Y. Xie, K. Muniswamy-Reddy, D. Feng, Y. Li, and D. D. E. Long. Evaluation of a hybrid approach for efficient provenance storage. *TOS*, 9(4):14, 2013.

[25] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings IEEE INFOCOM*, pages 594–602, 1996.

[26] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of SOSP*, pages 295–310, 2011.

[27] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. G. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. *PVLDB*, 6(2):49–60, 2012.

[28] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of ACM SIGMOD*, pages 615–626, 2010.

APPENDIX

In this appendix, we provide proof sketches of the key lemmas in the paper. We refer interested readers to our companion technical report [6] for more details.

A. PROVENANCE TREE EQUIVALENCE

We formally define tree equivalence below 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:

$$\begin{array}{rcl} \textit{Provenance tree } tr & ::= & \langle rID, P, ev, B_1::\cdots::B_n\rangle \\ & & \mid \langle rID, P, tr, B_1::\cdots::B_n\rangle \end{array}$$

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

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

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

B. CORRECTNESS OF STATIC ANALYSIS

Formal definition of equivalent keys. In this section, we explain the predicates used to define equivalence keys.

$\boxed{DQ \vdash \mathsf{joinFAttr}(p{:}i, q{:}j)}$

JoinF-Base
$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), b_1(\vec{x}_{b1}), \cdots, b_k(\vec{x}_{bk}), \cdots, b_n(\vec{x}_{bn}), \cdots \in DQ \quad q{:}i = p{:}j}{DQ \vdash \mathsf{joinFAttr}(p{:}i, q{:}j)}$$

$\boxed{DQ \vdash \mathsf{joinSAttr}(p{:}i)}$

Join-Base
$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), b_1(\vec{x}_{b1}), \cdots, b_k(\vec{x}_{bk}), \cdots, b_n(\vec{x}_{bn}), \cdots \in DQ \quad q{:}i = b_k{:}j}{DQ \vdash \mathsf{joinSAttr}(q{:}i)}$$

Join-Func-Attr
$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots, F_i : y := F(\vec{z}), \cdots \in DQ \quad q{:}j = \vec{z}{:}k}{DQ \vdash \mathsf{joinSAttr}(q{:}j)}$$

Join-Arith-Left
$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots, a_L(\vec{x}_{aL}) bop\ a_R(\vec{x}_{aR}), \cdots \in DQ \quad a_L{:}j = q{:}i}{DQ \vdash \mathsf{joinSAttr}(q.i)}$$

Join-Arith-Right
$$\frac{rID\ p(\vec{x}_p) \text{ :- } q(\vec{x}_q), \cdots, a_L(\vec{x}_{aL}) bop\ a_R(\vec{x}_{aR}), \cdots \in DQ \quad a_R{:}j = q{:}i}{DQ \vdash \mathsf{joinSAttr}(q{:}i)}$$

We define rules to derive how an attribute of a tuple is connected to an event trigger attribute.

$\boxed{DQ \vdash \mathsf{connected}(e{:}i, p{:}j)}$

$$\frac{DQ \vdash \mathsf{joinFAttr}(e{:}i, p{:}j)}{DQ \vdash \mathsf{connected}(e{:}i, p{:}j)}\ \text{Connected-Base}$$

Connected-Join
$$\frac{DQ \vdash \mathsf{connected}(e{:}i, q{:}j) \qquad DQ \vdash \mathsf{joinFAttr}(q{:}j, p{:}k)}{DQ \vdash \mathsf{connected}(e{:}i, p{:}k)}$$

$e{:}i$ is in the equivalent key K is defined as follows.

$\boxed{DQ \vdash e{:}i \in K}$

$$\frac{DQ \vdash \mathsf{joinSAttr}(e{:}i)}{DQ \vdash e{:}i \in K}\ \text{Equi-Direct}$$

Equi-Reachable
$$\frac{DQ \vdash \mathsf{joinSAttr}(q{:}j) \qquad DQ \vdash \mathsf{connected}(e{:}i, q{:}j)}{DQ \vdash e{:}i \in K}$$

Proof of correctness of equivalence keys (Lemma 2). The proof uses induction over the structure of tr_1. In the base case, tr_1 only uses one rule. We construct tr_2 using the same slow-changing tuples as tr_1 because ev_1 and ev_2 agree on every attribute that joins with a slow-changing tuple. By the definition of DELP, any attribute on the root of tr_1 and tr_2 whose values differ in those trees must come from (be connected to) the input event tuple, not from any slow-changing tuples; otherwise, they cannot differ. By the definition equivalence keys, this differing attribute is not in the equivalence keys.

For the inductive case, $tr_p{:}p(t_1, ..., t_n)$ contains a subtree $tr_q{:}q(s_1, ..., s_m)$, in which $q(s_1, ..., s_m)$ is the event of the rule r that derived $p(t_1, ..., t_n)$. By I.H., we know that we can construct $tr'_q{:}q(s'_1, ..., s'_m)$, which is equivalent to $tr_q{:}q(s_1, ..., s_m)$, and for any j s.t. $s_j \neq s'_j$, $\exists l$, $q{:}j$ is connected to $e{:}l$, and $e{:}l \notin K$. Next, we show that we can construct a $tr'_p{:}p(t'_1, ..., t'_n)$ that is equivalent to $tr_p{:}p(t_1, ..., t_n)$. We can construct $tr'_p{:}p(t'_1, ..., t'_n)$ by using the same rule r and the same slow-changing tuples from the last layer of $tr_p{:}p(t_1, ..., t_n)$. All the attributes that have different values for $q(s_1, ..., s_m)$ and $q(s'_1, ..., s'_m)$ do not join with the attributes in slow-changing tables; otherwise, it contradicts with the condition that $e{:}l \notin K$. The condition that $\forall i$, $t_i \neq t'_i$, $\exists l$, $p{:}i$ is connected to $e{:}l$, and $e{:}l \notin K$ follows from the fact that such $p{:}i$ must satisfy $\mathsf{joinFAttr}(p{:}i, q{:}j)$ for some $q{:}j$, where $\exists l$, $q{:}j$ is connected to $e{:}l$, and $e{:}l \notin K$.

C. EXAMPLE DEPENDENCY GRAPH

Figure 17 shows an example attribute-level dependency graph for the packet forwarding program in Figure 1. Based on Section 5.2, the equivalence keys are (packet:0, packet:2).

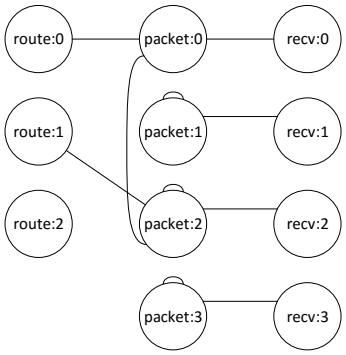

Figure 17: The attribute-level dependency graph for the packet forwarding program in Figure 1.

D. CORRECTNESS OF COMPRESSION

The key to showing that our online compression algorithm stores the required provenance trees is to define a bisimulation relation $\boldsymbol{\mathcal{R}_C}$ between the network state of the online compression execution ($\mathcal{C}_{cm}$) and the network state of seminaïve evaluation ($\mathcal{C}_{sn}$) introduced in prior work [11].

A bisimulation relation between $\mathcal{C}_{sn}$ and $\mathcal{C}_{cm}$. We define $\boldsymbol{\mathcal{R}_C}$, a bisimulation relation that corresponds the derivation trees tr in $\mathcal{C}_{sn}$ (where $\mathcal{C}_{sn} = \mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}$) to the provenance trees $\mathcal{P}$ in $\mathcal{C}_{cm}$ (where $\mathcal{C}_{cm} = \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$). We present the formal definition below.

$$\frac{\begin{array}{l}\mathcal{E}_1 :: \mathcal{S}_{cm}.\Gamma \vdash \mathcal{Q}_{sn} \sim_{\mathcal{U}} \mathcal{Q}_{cm} \\ \mathcal{E}_2 :: \forall i \in [1, N], \mathcal{S}_{cm}.\Gamma \vdash \mathcal{S}_{sn_i}.\mathcal{U}_{sn}\ \boldsymbol{\mathcal{R}_{\mathcal{U}}}\ \mathcal{S}_{cm_i}.\mathcal{U}_{cm} \\ \mathcal{E}_3 :: {\mathcal{U}_{cm}}^F \subseteq \mathcal{Q}_{cm} \cup \left(\cup_{i=1}^{N} \mathcal{S}_{cm_i}.\mathcal{U}_{cm}\right) \\ \mathcal{E}_4 :: \mathcal{S}_{cm}.\Gamma, \mathcal{S}_{cm}.DQ, {\mathcal{U}_{cm}}^F \\ \qquad \vdash \cup_{i=1}^{N} \mathcal{S}_{sn_i}.\mathcal{M}\ \boldsymbol{\mathcal{R}_{\mathbf{re}}}\ \mathcal{S}_{cm_i}.(\text{Table ruleExec}) \\ \mathcal{E}_5 :: \mathcal{S}_{cm}.\Gamma, \mathcal{S}_{cm}.DQ, {\mathcal{U}_{cm}}^F, \cup_{i=1}^{N} \mathcal{S}_{cm_i}.\Upsilon \\ \qquad \vdash \cup_{i=1}^{N} \mathcal{S}_{sn_i}.\mathcal{M}_{prov}\ \boldsymbol{\mathcal{R}_{\mathbf{prov}}}\ \cup_{i=1}^{N} \mathcal{S}_{cm_i}.(\text{Table prov})\end{array}}{\mathcal{Q}_{sn} \rhd \mathcal{S}_{sn1} \cdots \mathcal{S}_{snN}\ \boldsymbol{\mathcal{R}_C}\ \mathcal{Q}_{cm} \rhd \mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}}$$

The network state $\mathcal{C}_{sn}$ consists of the local states $\mathcal{S}_{sn1}$ $\cdots$ $\mathcal{S}_{snN}$ of each node in the distributed system that the program DQ is evaluated on, and a queue of unprocessed updates $\mathcal{Q}_{sn}$ representing the rule events in DQ that are

to be sent to various nodes in the distributed system to trigger rules in DQ. Similarly, a network state $\mathcal{C}_{cm}$ consists of a set of local states $\mathcal{S}_{cm1} \cdots \mathcal{S}_{cmN}$ for each node in the distributed system, and a queue of unprocessed updates $\mathcal{Q}_{cm}$ representing the same rule events in $\mathcal{Q}_{sn}$.

The local states $\mathcal{S}_{sn}$ and $\mathcal{S}_{cm}$ of both evaluations contain the same copy of DQ and also a declaration Γ that maps relations in DQ to their types as specified by the network administrator (i.e. whether they are input events or slow-changing tuples).

A semi-naïve local state $\mathcal{S}_{sn}$ is written as $\langle @\iota,\ DQ,\ \Gamma,\ \mathcal{DB},\ \mathcal{E},\ \mathcal{U}_{sn},\ equiSet,\ \mathcal{M},\ \mathcal{M}_{prov}\rangle$. Node ι is the identifier of the node in the distributed system whose local state is represented by $\mathcal{S}_{sn}$. DQ and Γ have been introduced above. $\mathcal{DB}$ is a local database of slow-changing tables and instances of relations of interest derived on node ι that are used as non-event inputs to a rule during the execution. $\mathcal{E}$ is a list of unprocessed input event tuples that will eventually be used to trigger the execution of DQ on node ι. $\mathcal{U}_{sn}$ are the unprocessed updates for the execution that have already been triggered by an input event tuple. *equiSet* records the equivalence keys that have been seen on node ι. The storage structures are $\mathcal{M}$ that store the proofs of the tuples P that are derived locally, and $\mathcal{M}_{prov}$ that record the proofs of the relations of interest that are stored locally.

The local state $\mathcal{S}_{cm}$ of the online compression scheme is similar to $\mathcal{S}_{sn}$. It is defined as $\langle @\iota,\ DQ,\ \Gamma,\ \mathcal{DB},\ \mathcal{E},\ \mathcal{U}_{sn},\ equiSet,$ Table ruleExec, Table prov$\rangle$. Because the online compression scheme stores *compressed* provenance trees, while semi-naïve evaluation does *not* perform compression, the structures used for storage in $\mathcal{S}_{cm}$ necessarily differ from that of $\mathcal{S}_{sn}$. In $\mathcal{S}_{cm}$, the ruleExec table (introduced in Section 5.3) corresponds to $\mathcal{M}$ in $\mathcal{S}_{sn}$, while the prov table corresponds to $\mathcal{M}_{prov}$ in $\mathcal{S}_{sn}$.

Our bisimulation relation uses relation $\boldsymbol{\mathcal{R}}_{\mathbf{re}}$ to related derivation trees tr stored in the $\mathcal{M}$ of local states of $\mathcal{C}_{sn}$ to provenance trees $\mathcal{P}$ stored in the ruleExec tables of the local states of $\mathcal{C}_{cm}$ ($\mathcal{E}_4$). We define the following relation $tr \sim_d \mathcal{P}$ between a derivation tree and a compressed provenance in order to formalize $\boldsymbol{\mathcal{R}}_{\mathbf{re}}$:

$$\frac{\begin{array}{c}\forall i \in [1,n], \texttt{vID}_i = \mathsf{hash}(B_i) \\ vids = \texttt{vID}_1 :: \cdots :: \texttt{vID}_n \qquad rid = \mathsf{hash}(r :: vids) \\ ruleExec = \langle @\iota, rid, r, vids, NULL, NULL\rangle\end{array}}{\langle r, P, e(@\iota, \vec{t}), B_1 :: \cdots :: B_n\rangle \sim_d ruleExec}$$

$$\frac{\begin{array}{c}ruleExec_q = \langle @\iota_q, rid_q, r_q, vids_q, @\iota_\rho, rid_\rho\rangle \\ tr_q \sim_d \mathcal{P} :: ruleExec_q \qquad \forall i \in [1,n], \texttt{vID}_i = \mathsf{hash}(B_i) \\ vids_p = \texttt{vID}_1 :: \cdots :: \texttt{vID}_n \qquad rid_p = \mathsf{hash}(r_p :: vids_p) \\ ruleExec_p = \langle @\iota_p, rid_p, r_p, vids_p, @\iota_q, rid_q\rangle\end{array}}{\begin{array}{c}\langle r_p, P, tr_q{:}q(@\iota_q, \vec{t_q}), B_1 :: \cdots :: B_n\rangle \\ \sim_d \mathcal{P} :: ruleExec_q :: ruleExec_p\end{array}}$$

$\boldsymbol{\mathcal{R}}_{\mathbf{re}}$ is one of several intermediary relations used to relate structures in each local state $\mathcal{S}_{sn}$ to structures in local state $\mathcal{S}_{cm}$ in order to define the bisimulation relation $\boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{C}}}$. These intermediary relations are used in $\mathcal{E}_1$ to $\mathcal{E}_4$ to specify the required correspondences. Due to space limitations, we elide most of their definitions.

- Semi-naïve evaluation and online compression execution have the same set of updates ($\mathcal{E}_1$).
- Every local state contains the same set of unprocessed updates during both executions ($\mathcal{E}_2$).
- All derivations derived by semi-naïve evaluation can either be found in the set of existing rule provenances of the online compression execution, or will eventually be generated by the unprocessed updates ($\mathcal{E}_3$ and $\mathcal{E}_4$).
- For each tuple whose provenance information we record during online compression execution, we also record its proof during semi-naïve evaluation, and vice versa ($\mathcal{E}_5$).

The last three judgments are very complicated because of possible out of order execution, thus we omit the details.

Proof of Lemma 4. We present the proof details of Lemma 4, from which Theorem 3 immediately follows. The proof uses induction on the number of steps and relies on a lemma relating two states if one takes a single step. The lemma shows that starting from related states ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{C}}}\ \mathcal{C}_{cm}$), if semi-naïve evaluation takes a step ($\mathcal{C}_{sn} \rightarrow_{SN} \mathcal{C}_{sn}'$), then online compression can also take a step (i.e. $\mathcal{C}_{cm} \nearrow_{CM} \mathcal{C}_{cm}'$) and the resulting states are again bisimilar ($\mathcal{C}_{sn}'\ \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{C}}}\ \mathcal{C}_{cm}'$) and vice versa.

There are two key cases where new rule provenances are generated and provenance storage data structures are updated: one is when all the rules that an update can fire are processed and the update is stored into the database if it is the result tuple, the other is when a rule is fired and a new update is generated.

In the first case, in semi-naïve evaluation, a derivation tr of tuple P is stored. In the online compression execution, the prov table will be updated. In this case, the derivation of P has already been stored in the respective provenance data structures when the rule that generates the update is processed. Therefore, we only need to check that the hash values in the prov table match corresponding fields in the derivation stored in $\mathcal{M}_{prov}$. This can be shown because the updates relate to each other in the initial execution states.

In the second case, an update P' triggers a rule r and generates a new update P. Semi-naïve evaluation stores the derivation tr for P. The online compression execution updates the ruleExec table. We need to show that (1) the new updates in both executions still relate to each other, (2) the new derivation tr and the new entry in the ruleExec table relate to each other, and (3) other relations are not affected by the new entries. All of the above can be shown by examining our relations.

E. CORRECTNESS OF QUERY

To show that our query algorithm (QUERY in Figure 18) is able to recover the correct derivation tree of a given tuple from our compressed provenance storage, we state and prove Correctness of the Query Algorithm (Theorem 5). 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 tr 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 tr) that are proofs of P and that were derived using ev.

Theorem 5 tells us that QUERY always returns all the derivations in $\mathcal{C}_{sn}$ for P and $evid$.

The proof relies on Lemma 3 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 ($\mathcal{C}_{sn}\ \boldsymbol{\mathcal{R}}_{\boldsymbol{\mathcal{C}}}\ \mathcal{C}_{cm}$). Given such an $\mathcal{C}_{sn}$, we pick an arbitrary derivation tree tr for tuple P in $\mathcal{C}_{sn}$ that was generated by event tuple ev with event ID $evid$.

```
1: function QUERY(C_cm, P, evid)
2:     htp ← hash(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:             tr_i ← TRANSFORM_TO_D(P_i, evid)
10:            M ← M ∪ tr_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 18: Pseudocode for querying a provenance tree.

Because $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{C}_{cm}$, there exists a tuple *prov* for P in a specific prov table in $\mathcal{C}_{cm}$ storing an association to a specific provenance $\mathcal{P}$, and furthermore $tr \sim_d \mathcal{P}$. 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 tr 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 6 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 6 (Correctness of Qr)**.** *Given that $\mathcal{C}_{sn}$ $\boldsymbol{\mathcal{R}_C}$ $\mathcal{C}_{cm}$ and tr:$P \in \mathcal{C}_{sn}$ and ruleExec = ⟨loc, rid, r, vids, nloc, nrid⟩ and $\mathcal{P}$ :: ruleExec is stored in the ruleExec tables of the local states of $\mathcal{C}_{cm}$ and $tr \sim_d \mathcal{P}$ :: ruleExec, then* $\text{QR}(\mathcal{C}_{cm}, (nloc, nrid)) = \mathcal{P} :: ruleExec$.

We prove Lemma 6 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, $\nexists tr \in \mathcal{C}_{sn}$ s.t. $tr \sim_d []$. 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 tr:$P \sim_d$ *ruleExec*, tr 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* = ⟨*loc*, *rid*, *r*, *vids*, *nloc*, *nrid*⟩) by the uniqueness property. The query $\text{QR}(\mathcal{C}_{cm}, (nloc, nrid))$ 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 $\text{QR}(\mathcal{C}_{cm}, (loc, rid)) = [] :: ruleExec$ as desired.

Inductive Case: $\ell = k + 1 \geq 2$. 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* = ⟨*loc*, *rid*, *r*, *vids*, *nloc*, *nrid*⟩) 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 $tr \sim_d \mathcal{P} ::$ *ruleExec*′, exists tr' in $\mathcal{C}_{sn}$ that is a subderivation of tr s.t. $tr' \sim_d \mathcal{P}$ and *ruleExec*′ = ⟨*nloc*, *nrid*, *r*′, *vids*′, *nloc*′, *nrid*′⟩. Using I.H. we can obtain $\text{QR}(\mathcal{C}_{cm}, (nloc, nrid)) = \mathcal{P}' ::$ *ruleExec*′. By Line 22 of QR, we have $\text{QR}(\mathcal{C}_{cm}, (loc, rid)) = \mathcal{P}' ::$ *ruleExec*′ :: *ruleExec* = $\mathcal{P} ::$ *ruleExec* as desired.

F. DELP FOR DNS RESOLUTION

r1 request(@$RT, URL, HST, RQID$) :−
request(@$HST, URL, RQID$).
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 19: DELP for DNS resolution.

Figure 19 shows the DELP encoding of the recursive DNS resolution. The program 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*.