Declarative Secure Distributed Systems

Wenchao Zhou
supervised by Boon Thau Loo
CIS Department, University of Pennsylvania
Philadelphia, PA, 19104
wenchaoz@cis.upenn.edu

ABSTRACT

In the past decade, distributed systems have rapidly evolved and gained significant traction in the research community, with an increasing interest concentrated on developing and analyzing secure distributed systems. In this paper, we present DS2 (*Declarative Secure Distributed Systems*), a unified platform for specifying, implementing, and analyzing large-scale secure distributed systems. First, we propose the Secure Network Datalog (*SeNDlog*) language that enables distributed systems and their security policies to be specified and implemented within a same declarative framework. We show that the existing semi-naïve evaluation can be extended to execute *SeNDlog* programs that incorporate authenticated communication among untrusted nodes. Second, we demonstrate that *network provenance* – the metadata that explains the derivation of network state – can be naturally and concisely captured within the DS2 system. We extend existing data models for provenance to enable distribution at Internet-scale, and present techniques for efficient and customizable maintenance and querying of network provenance. Finally, the future research plans on secure provenance and its integration with legacy applications are presented for discussion.

1. INTRODUCTION

In the past decade, we have witnessed a proliferation of distributed systems deployed at Internet-scale for a variety of application domains ranging from Internet monitoring infrastructures, publish-subscribe systems, to content distribution networks. Despite their widespread usage, designing and implementing these large-scale systems remains a challenge, in part because of the sheer scale of deployment, but also resulting from emerging security threats.

In response, there have been several proposals aimed at evolving the underlying network infrastructure to provide better support for network diagnosis, flow analysis and accountability, all of which are geared towards better tools for analyzing and securing networks. However, most of these mechanisms are typically designed to tackle specific security threats at the underlying network, without taking into account content distribution and information processing at higher layers. In addition, they are often afterthoughts, implemented and enforced in a different language or environment from the networks that they are trying to protect, hence raising the barrier for adoption.

As a step towards the integration of distributed systems with security policies and analysis, we present DS2 (*Declarative Secure Distributed Systems*), a unified declarative platform for specifying, implementing, and analyzing large-scale secure distributed systems. Our work has largely been inspired by recent efforts at using declarative languages that are aimed at simplifying the process of system specification and implementation. Our work builds upon and unifies three bodies of work: (1) *declarative networking* [22, 21, 20], (2) *logic-based* trust management systems [11, 3, 17], and (3) database techniques for analyzing data computations via the concept of *provenance* (or *lineage*) [2].

From a practical standpoint, this integration has several benefits, ranging from fewer languages to learn, fewer sets of optimizations, finer-grain control over the interaction between security and network protocols, and the potential of crosslayer analysis and optimizations. Given its close tie with logic-based deductive languages, runtime monitoring and checking of distributed systems against formally specified properties are also achievable in our framework [36], where high-level safety properties are automatically compiled from platform-independent formal specifications into distributed monitoring queries for execution.

Additionally, the unified declarative rule-based framework captures information flow as distributed queries, enabling natural support for acquiring, maintaining and querying *provenance* – the metadata that explains where a tuple originated and traversed, how it was derived and what nodes are involved in the derivation [35]. Such capability is essential to a diverse set of network management tasks such as performing network diagnostics, identifying malicious users, and enforcing trust management policies. Each goal has led to a series of application-specific proposals [28, 14, 32, 8, 16] that focus on improving network support for accountability and providing efficient mechanisms to trace packets and information flows through the Internet. We explore the data management challenges posed by the distribution, querying, and maintenance of network provenance at Internet-scale, and propose techniques to enable the support for network provenance in the DS2 system.

We summarize our contributions as follows:

Unified framework for secure distributed systems: We propose the Secure Network Datalog (*SeNDlog*) language that unifies logic-based access control and declarative networking languages, hence enabling distributed systems and their security policies to be specified within a unified declarative framework [34]. We demonstrate the flexibility and compactness of *SeNDlog* via secure specifications of various distributed systems running at different network layers. We further demonstrate in Reference [24, 23] that various

VLDB 2010 PhD Workshop, September 13, 2010, Singapore

security constructs can be customized and composed in a declarative fashion, via the use of meta-programmability.

Maintenance and querying of network provenance: We introduce the notion of network provenance [35], and demonstrate its mapping to various use cases, including real-time diagnostics, forensics, incremental view maintenance, and trust management. We further present and demonstrate, based on the DS2 system, the support for efficient distribution, maintenance and querying of network provenance at Internet scale. As an ongoing project, we actively investigate the techniques for enforcing the integrity and confidentiality of provenance.

The paper is organized as follows. In Section 2, we first present as background the declarative networking language. Section 3 introduces the unified *SeNDlog* language, and illustrates its usage via a series of example secure distributed systems. In Section 4, we present the maintenance and querying of network provenance, and show its natural mapping to use cases in a variety of application scenarios. We discuss our ongoing work and potential future research directions in Section 5 and conclude in Section 6.

2. BACKGROUND

Given DS2's use of declarative networking, we briefly introduce declarative networking and the query language that will be used as a basis for enabling network provenance. The high level goal of *declarative networks* [22, 21, 20] is to build extensible network architectures that achieve a good balance of flexibility, performance, and safety. Declarative networks are specified using *Network Datalog* (*NDlog*), a distributed recursive query language used for querying network graphs. *NDlog* queries are executed using a distributed query processor to implement the network protocols and are continuously maintained as distributed views over existing network and host state. Declarative queries such as *NDlog* are a natural and compact way to implement a variety of routing protocols and overlay networks. For example, traditional routing protocols can be expressed in a few lines of code [22], and the Chord [31] distributed hash table in 47 lines of code [21]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations.

The techniques proposed in this paper can be generally realized using any sufficiently expressive distributed query processor. The advantage of using declarative networking is that several robust implementations exist that can be straightforwardly leveraged to develop DS2. Moreover, since distributed protocols can themselves be expressed as declarative statements, declarative networking represents a natural means for unifying the synthesis and analysis of distributed protocols.

The declarative *NDlog* language used by DS2 is based on Datalog [26]. A Datalog program consists of a set of *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and `...` and `qn` imply `p`". Here, `p` is the *head* of the rule, and `q1, q2,...,qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* with *attributes* (which are bound to variables or constants by the query), or Boolean expressions that involve function symbols (including arithmetic) applied to attributes. Predicates in *NDlog* are typically relations, although in some cases they may represent functions. Commas are interpreted as logical conjunctions (*AND*). The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter.

NDlog is a distributed variant of the traditional Datalog, primarily designed for expressing distributed (recursive) computations. *NDlog* supports a *location specifier* in each predicate, expressed with the `@` symbol followed by an attribute. This attribute denotes the location where the tuple resides (e.g., tuple `link(@S,D,C)` is located at node `S`, as indicated by its first field).

```
sp1 pathCost(@S,D,C) :- link(@S,D,C).
sp2 pathCost(@S,D,C1+C2) :- link(@Z,S,C1),
       bestPathCost(@Z,D,C2).
sp3 bestPathCost(@S,D,min<C>) :- pathCost(@S,D,C).
```

Figure 1: The MINCOST program in *NDlog*

Consider the three-rule MINCOST program shown in Figure 1. MINCOST computes the best path cost between each pair of nodes in a network. Rule `sp1` and `sp2` specify the definition of the derived tuple `pathCost`. Rule `sp1` computes all one-hop path cost based on the base tuples from the `link` relation. Rule `sp2` expresses that "if there is a link from `S` to `Z` of cost `C1`, and the best path cost from `Z` to `D` is `C2`, then there is a path from `S` to `D` with cost `C1+C2`"[1]. Rule `sp3` aggregates all paths with the same pair of source and destination to compute the best path cost. By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

When executed, MINCOST forms a distributed stream computation where streams of `link`, `pathCost`, and `bestPathCost` tuples are joined at different nodes to compute the best path costs. To maintain tuples as the inputs to the rules are updated (e.g., insertion of `link` tuples), these queries are continuously executed.

3. SECURE NETWORK DATALOG

SeNDlog is based on a unification of logic-based access languages and distributed recursive query languages for declarative networking. In this section, we introduce its key langurage features and the query processing techniques that we have adopted in DS2. An example secure protocol is presented to illustrate the flexibility and compactness of *SeNDlog*.

3.1 Language Features

The *SeNDlog* language unifies Binder and NDlog with the following goals in mind. First, the language should maintain the features of Binder and NDlog, and remain compatible with these two languages. Second, *SeNDlog* must support authenticated communication and enable the differentiation of principals according to their roles in trusted and untrusted networked environments. We have chosen Binder for its simple language design and similarities to NDlog. Despite its simplicity, we show that the unified language can support a variety of networked systems and security policies.

SeNDlog extends the basic declarative networking language by adding support for authenticated communication. *SeNDlog* integrates two commonly used constructs in distributed trust management languages: (1) the notion of *context* to represent a principal in a distributed environment and (2) a specific operator `says` that abstracts away the details of authentication [3, 15].

To demonstrate the language features of *SeNDlog*, we present the authenticated version of the MINCOST example shown in Figure 1:

```
At S:
 sp1 pathCost(S,D,C) :- link(S,D,C).
 sp2 pathCost(Z,D,C1+C2)@Z :- link(S,Z,C1),
       bestPathCost(S,D,C2).
 sp3 bestPathCost(S,D,min<C>) :- Z says pathCost(S,D,C).
```

Figure 2: Authenticated MINCOST program in *SeNDlog*

The `says` primitive in rule `sp3` specifies that the authenticity of the received `pathCost` tuple should be checked, ensuring that it originated from `Z`.

[1]In this example, we assume links are symmetric, i.e. if there is a link from `S` to `D` with cost `C`, then a link from `D` to `S` with the same cost `C` also exists.

Communication context: Due to the distributed nature of network queries, a principal does not have control over rule execution at other nodes. *SeNDlog* achieves secure distributed query processing by allowing programs to inter-operate correctly and securely via the export and import of rules and derived tuples across contexts.

In the above example, the rules are in the context of `S`, where `S` is a variable assigned upon rule installation. In a distributed environment, `S` represents the network address of a node. In a multi-user multi-layered network environment, `S` can further include a username and network identifier.

Import/export predicates: The *SeNDlog* language allows different contexts to communicate by importing and exporting tuples. The communication serves two purposes: (1) to disseminate maintenance messages as part of the protocol updates, and (2) to distribute the derivation of security decisions.

During the evaluation of *SeNDlog* rules, derived tuples can be communicated among contexts via the use of *import predicates* and *export predicates*. An import predicate is of the form "`N says p`", indicating that principal `N` asserts the predicate `p`. The use of export predicates provides confidentiality by exporting tuples only to specified principals. An export predicate is of the form "`N says p@X`", where principal `N` exports the predicate `p` to principal `X`. In rule `sp2`, node `S` exports `pathCost` tuples to node `Z` (as a shorthand, "`S says`" is omitted as `S` is where the rule resides).

3.2 Secure Query Processing

We extend the *Pipelined Semi-naïve* (PSN) evaluation [20] proposed for declarative networks, to incorporate authenticated communication into query execution. We start with a description of our proposed *Secure Pipelined Semi-naïve* (SPSN) evaluation, followed by a brief overview of the workflow architecture.

Secure Pipelined Semi-naïve Evaluation: Consider the following *SeNDlog* rule in the context of principal p:

a :- $d_1, ..., d_n, b_1, ..., b_m, p_1$ *says* a_1, p_2 *says* $a_2 ..., p_o$ *says* a_o.

where there are n derived predicates ($d_1, ..., d_n$), m base predicates ($b_1, ..., b_m$), and o additional import predicates of the form "p_k *says* a_k" in the rule body, and an export predicate in the rule head. For each k^{th} import predicate, an *authenticated delta rule* is generated as follows:

p *says* $\triangle a$:- $d_1, ..., d_n, b_1, ..., b_m,$
p_1 *says* $a_1, ..., p_k$ *says* $\triangle a_k, ..., p_o$ *says* a_o.

The delta rule uses `says` to authenticate new a_k tuples imported from p_k and sign any derived a tuples by the local principal p.

In SPSN, tuples are processed tuple-at-time in a pipelined fashion. Each node maintains a FIFO queue (ordered by arrival timestamp) of new input tuples. Each new tuple is dequeued and is used as input to its respective delta rule. The execution of a delta rule may generate new tuples which are either inserted into the local queue or sent to a remote node for further execution.

Dataflow Architecture: Figure 3 shows an example dataflow that is automatically generated from *SeNDlog* rules. Queries are compiled and executed as *distributed dataflows* and share a similar execution model with the Click modular router [13]. At the edges of the dataflow, there are several network processing operators (denoted by `Network-In` and `Network-Out`) used to process incoming and outgoing messages. Flow control operators such as `Queue`, `Mux`, `Demux`, and `TimedPullPush` support buffering, multiplexing, demultiplexing, and periodic flow of tuples within the dataflow.

At the core of the dataflow are *rule strands* shown within the gray box, which are directly compiled from the SPSN delta rules into a series of relational operators such as joins, aggregations, selections, and projections. Messages flowing among dataflows are

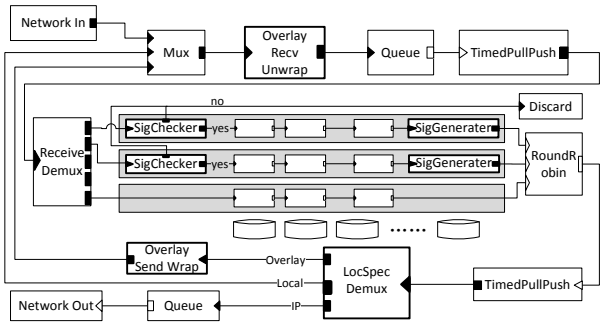

Figure 3: Datalow execution plan for a single node

executed at different nodes, resulting in updates to local tables, or query results that are returned to the hosts that issued the queries.

Specifically, two additional operators (i.e. `SigGenerator` and `SigChecker`) are introduced to support authenticated communication. Any outgoing tuple `t` that requires authentication is communicated as a (`p,s,t`) triplet, where `p` corresponds to the source principal, and `s` is the signature generated by signing the message digest (essentially a cryptographic hash of `t`). At the recipient node, the `SigChecker` operator authenticates incoming (`p,s,t`) triplets by verifying that the signature `s` matches the corresponding message digest. Note that key management is an orthogonal problem that is beyond the scope of the paper.

3.3 Example Secure Protocol

To illustrate the flexibility and compactness of *SeNDlog*, we present an example secure protocol based on the Chord Distributed Hash Table (DHT) [31]. Our modifications avoid a security weakness in a DHT where malicious nodes can occupy a large range of the key space. There are three types of nodes: (1) a *new node* `NI` joining the chord network, (2) the *certificate authority* `CA`, and (3) the *landmark node* `LI`. Each node runs its respective set of rules:

```
At NI,
 ni1 requestCert(NI,K)@CA :- startNetwork(NI),
      publicKey(NI,K), MyCA(NI,CA).
 ni2 nodeID(NI,N) :- CA says nodeIDCert(NI,N).
 ni3 CA says nodeIDCert(NI,N)@LI :-
      CA says nodeIDCert(NI,N), landmark(NI,LI).

At CA,
 ca1 nodeIDCert(NI,N)@NI :- NI says requestCert(NI,K),
      secret(CA,NI,S), N=f_generateID(K,S).

At LI,
 li1 acceptJoinRequest(NI) :- CA says nodeIDCert(NI,N).
```

Figure 4: Secure Process of a Node Joining the Chord Network

In rule `ni1`, a node `NI` that wishes to join the Chord network exports a `requestCert` tuple to its `CA` to request a *nodeID certificate*. Upon receiving the request, the `CA` generates a `nodeIDCert(NI,N)` tuple containing the nodeID (indicated by `N`) and its certificate, which is then exported back to node `NI`. Upon importing the `nodeIDCert` tuple from the `CA`, using rule `ni2`, node `NI` initializes its local node identifier. It also forwards the `nodeIDCert` to its landmark node `LI` in order to join the chord network. The landmark node `LI` verifies the authenticity of `nodeIDCert` and derives an `acceptJoinRequest(NI)` tuple, indicating the request from node `NI` is accepted.

In Reference [34], we present more example secure protocols that run at different network layers, ranging from the path-vector routing protocol, the PIER [9] distributed query processor, and an authenticated MapReduce execution. In addition, we further demonstrate the usability of *SeNDlog* in the A3 [29] (Application-Aware

Anonymity) system, where *SeNDlog* is extensively used to leverage path selection, and secure path instantiation.

4. NETWORK PROVENANCE

In database systems, data provenance [2] is a well-known concept, primarily used to answer questions concerning how query results are derived and which data sources they come from. A similar notion – *network provenance* [35] – is emerging in the networking domain, describing the history and derivations of network state resulting from the execution of a distributed protocol.

We conceptualize and develop, based on the DS2 system, a platform that enables *generic* provenance support. It provides a flexible framework for distributed querying of network meta-data, allowing provenance to be represented in various format (derivation trees, binary decision diagrams [1], algebraic polynomials, etc.).

4.1 Provenance Maintenance

We model provenance as an acyclic graph $G(V, E)$. The vertex set V consists of *tuple vertices* and *rule execution vertices*. Each tuple vertex in the graph is either a base tuple or a computation result, and each rule execution vertex represents an instance of a rule execution given a set of input tuples. The edge set E represents dataflows between tuples and rule execution vertices. To uniquely identify each vertex in the graph, we assign a vertex ID (VID) to each tuple vertex and a rule ID (RID) to each rule execution vertex.

To illustrate, consider an example network consisting of three nodes `a`, `b` and `c` connected by three bi-directional links `(a,b)`, `(a,c)` and `(b,c)` with costs `3`, `5` and `2` respectively. Figure 5 shows the provenance graph for a specific derived tuple, `bestPathCost(@a,c,5)`. In the figure, ovals represent the rule execution vertices and rectangles depict tuple vertices. The graph encodes how tuples are derived during the execution of the MINCOST protocol. For instance, `bestpathCost(@a,c,5)` is generated from rule `sp3` at node `a` taking `pathCost(@a,c,5)` as the input. To trace further, `pathCost(@a,c,5)` has two derivations, i.e. the locally derivable one-hop path $a \rightarrow c$ and the two-hop path $a \rightarrow b \rightarrow c$ that requires the distributed join (in rule `sp2`) at `b`.

Storage Model: Our storage model builds upon current work on representing provenance information as relational tables [10, 6], with extensions to support distributed storage and querying. In DS2, provenance information is stored in the network using two tables, `prov` and `ruleExec`, that are distributed and partitioned across all nodes in the network.

The `prov` table maintains provenance information, where each entry in the relation represents a direct derivation of a tuple. Specifically, a `prov` entry is of the form `prov(@Loc,VID,RID,RLoc)`, with `VID` and `RID` as its keys, indicating that the tuple vertex `VID` located at node `Loc` is directly derivable from the rule execution vertex `RID` for a rule that resides at `RLoc`.

A separate table, `ruleExec(@RLoc,RID,R,VIDList)`, stores the actual meta-data of a rule execution (at location `RLoc`). For a given `RID`, the table stores the actual rule identifier `R`, as well as the VIDs for all the input tuples used in the rule derivation.

Distributed Maintenance: Given a declarative networking program, an automatic rewrite [35] is performed to augment the original program with additional queries for maintaining provenance information. Essentially, provenance information (i.e. the `prov` and `ruleExec` tables) are defined as views of base and derived tuples. As DS2 adopts SPSN – a variant of PSN, views are incrementally recomputed due to new insertions or deletions. Each new derivation or rule execution automatically results in the creation of new `prov` and `ruleExec` entries. Similarly, whenever a base tuple is deleted, all derivations resulted from *NDlog* rules that depend on the base tuple in the program are incrementally deleted, resulting in cascaded deletions of the respective `prov` and `ruleExec` entries in the provenance graphs of deleted tuples.

4.2 Provenance Querying

Network provenance can be queried by issuing distributed queries. These queries traverse provenance graphs (in the form the `prov` and `ruleExec` tables) in a distributed fashion, returning results to the querying node.

The following *NDlog* program demonstrates a generic distributed graph traversal operation on tables `prov` and `ruleExec`. The entire program is written in ten *NDlog* rules: two base rules (`edb1` and `c0`), and two pairs of four rules for recursively querying the `prov` (`idb1-idb4`) and `ruleExec` (`rv-rv4`; not shown) tables. The rules are continuous, long-running queries that are installed at every DS2 node for handling distributed provenance queries.

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

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

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

// Recursive case
idb2 eRuleQuery(@RLoc,RQID,RID,X) :-
       prov(@X,VID,RID,RLoc),
       eProvQuery(@X,QID,VID,Ret), RQID=f_sha1(QID+RID).

// Buffer sub-results
idb3 pResultTmp(@X,QID,Ret,VID,Buf) :-
       pResultTmp(@X,QID,Ret,VID,Buf1),
       eRuleResults(@X,RQID,RID,Prov),
       RQID=f_sha1(QID+RID), Buf=f_concat(Buf1,Prov).

// Calculate and return results
idb4 eProvResults(@Ret,QID,VID,Prov) :-
       pResultTmp(@X,QID,Ret,VID,Buf), numChild(@X,VID,C),
       C=f_size(Buf), Prov=f_pIDB(Buf,VID,X).
```

The initial provenance query is indicated by the event `eProvQuery(@X,QID,VID,Ret)`, where node `Ret` issues a query (uniquely identified by `QID`) to retrieve the provenance information of tuple `VID` stored at node `X`.

Rule `edb1` is the base case and applies when the tuple `VID` is a base tuple (EDB), as indicated by the fact that it has no associated rule execution instance (i.e., `RID` is null). In such cases, the provenance information is `f_pEDB(VID)` – the result of applying the user-defined function for EDBs to `VID`.

Rule `idb1` initializes the `pResultTmp` table, which is later used to buffer intermediate query results. Rule `idb2` represents the recursive case in which the `prov` table is retrieved. Each entry with matching `VID` in the `prov` table indicates a rule execution instance that leads to the derivation of `VID`. These rule execution instances are retrieved and buffered in `pResultTmp` table. Rule `idb3` is applied when all children derivations have returned with the provenance information. The resulting provenance information is then combined in rule `idb4` using the user-defined `f_pIDB` function and the results are returned to the query node.

Additional four rules `rv1-rv4` (similar to `idb1-idb4`) perform a similar traversal of the `ruleExec` tables. We omit these rules due to space constraints. The intuition behind these rules is that the user recursively traverses `prov` and `ruleExec` tables across nodes until the entire provenance tree has been obtained. An additional user-defined function `f_pRULE` enables the user to customize how the inputs to the rule can be combined in the provenance tree.

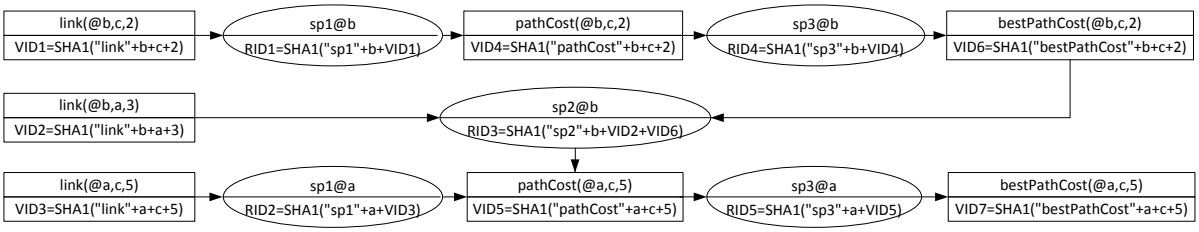

Figure 5: The provenance graph of the tuple `bestPathCost(@a,c,5)` derived from the execution of the MINCOST program. Ovals represent rule execution vertexes and rectangles denote tuple vertexes.

Customization: Users may customize provenance queries to meet various application requirements, by configuring the three user-defined functions, namely `f_pEDB`, `f_pIDB`, and `f_pRule`. The underlying *NDlog* program (shown above) used for querying provenance is sufficiently general to support a diverse set of application requirements.

As an example, consider a query that returns the number of possible derivations of a given tuple. For each base tuple, `f_pEDB` is configured to return an integer "1", indicating each of the base tuples has one derivation. For each intermediate derived tuple, the number of its derivations (i.e. the result of `f_pIDB`) can be calculated as the sum of the sub-results returned by the direct derivations. For rule execution instances, the evaluation result of the `f_pRule` functioin is the product of the sub-results.

Optimizations: We further investigate a variety of query optimization techniques [35], including a) caching previously acquired provenance to allow subsequent queries to leverage the results of prior ones; b) altering traversal orders in provenance graph to reduce bandwidth consumptions for threshold-based queries; and c) applying lossy condensation of provenance [18], while still maintaining sufficient information for applications such as incremental view maintenance and trust management.

4.3 Use Cases of Provenance

The capability of efficient maintenance and customizable querying of network provenance enables a variety of applications in developing and analyzing secure distributed systems. In this section, we survey a (non-exhaustive) list of potential use cases of network provenance:

Diagnosis and forensics: In addition to trust management, provenance information is useful for debugging and error detection. For example, tracing backwards in a network-level provenance graph may yield the discovery of the (possibly malicious) causes of suspicious query results. In all scenarios, the querying of provenance can be automatically *triggered* by an anomalous behavior (e.g., a spike in traffic) that is detectable using a continuous query over existing network state.

Efficient incremental view maintenance: Once incorrect or untrusted data is identified, provenance enables the efficient propagation of corrections to the appropriate destinations, without imposing expensive recomputation.

Provenance-based trust management: A secure query processor may decide to process or discard an incoming tuple based on its derivation. For instance, a node may make the decision based on its trust relationship with the tuple's original owner. Additionally, provenance allows the adoption of a *quantitative* approach for trust management: a derived tuple is assigned a trust value evaluated from its provenance, based on which the decision is concluded. Interestingly, the quantitative approach is computable and customizable by representing provenance in an algebraic form.

5. FUTURE WORK

Having presented our unified declarative framework for specifying and implementing secure distributed systems and its capability of supporting efficient provenance maintenance and querying. We briefly discuss our future research plans on integrating security guarantees into the provenance support and the interaction between DS2 and legacy applications that are not necessarily implemented using our framework.

5.1 Secure Provenance

Provenance has wide applications in network diagnosis, flow analysis and trust management, where the systems are usually assumed to be running in a potentially adversarial environment. Therefore, it is crucially important to enforce the *integrity* and *confidentiality* of network provenance.

Threat Model and Desired Security Guarantees: We enforce the integrity of provenance by assuring that malicious manipulation of provenance will be eventually detected. In particular, we currently consider the following list of security threats:

- *T1 – Forge updates of base tuples:* A compromised node may forge fake updates of the base tuples maintained in its local database. For instance, in a BGP system, a malicious AS may falsely announces the origination of an arbitrary prefix.
- *T2 – Deviate from expected behaviors:* In the execution of a distributed system, a misbehaving node may intentionally deviate from its expected behaviors. Without noticing the deviation, a system administrator may draw false conclusions from retrieved provenance, even if it faithfully captures the derivation of tuples.
- *T3 – Manipulate received provenance:* When a provenance query is issued, a compromised node involving in the distributed execution of the query can covertly manipulate its received provenance data, and return a framed results back to the query issuer.
- *T4 – Ignore/replay updates of provenance:* A compromised node may ignore the insertion / deletion of a tuple's alternative derivations. On the other hand, a replay attack can be easily launched by leveraging a stale update.

In addition to integrity, another aspect of secure provenance is in the form of confidentiality. A naïve implementation of provenance may result in information leakage by exposing sensitive information (e.g. routing policies in an inter-domain routing protocol) to the recipients of the tuples, some of which are not supposed to access to these information. Therefore, provenance information should be hidden (encrypted) based on roles, security levels or customized requirements.

Preliminary Solution - Integrity: Noting that the internal faults – threats *T1* and *T2* – are not observable from other nodes, we make two assumptions: 1) Each base tuples is tagged with a certificate authorized by a trusted certificate authority; and 2) Each participating node's expected behavior can be modeled as a deterministic

state machine[2], and is publicly known as a *reference implementation*. Assumption 1 allows nodes to prevent forged base tuples by checking the tagged certificates, whereas Assumption 2 enables one to verify, by leveraging deterministic replay, whether nodes strictly adhere to their expected behaviors in deriving a given tuple.

In addition, inspired by PeerReview [7], we maintain a linear event log of the incoming and outgoing communication at each node. Mechanisms proposed in PeerReview ensure that the logs are *tamper-evident* (thus preventing threat *T3*), and *consistent* – node cannot deny having received provenance updates (thus preventing threat *T4*). Once an violation is discovered, verifiable evidences are generated against the faulty nodes.

Preliminary Solution - Confidentiality: To enforce the confidentiality of provenance, we explore an existing approach [25] to support *information hiding* (fine-granularity confidentiality control) in provenance. This approach was previously applied to leverage access controls over sensitive information, which is maintained in a published XML file, by *partially* encrypting the file in a key-based tree-structured architecture.

5.2 Integration with Legacy Applications

Another direction we are actively exploring is to integrate DS2 more closely with legacy applications, which are not necessarily implemented within our declarative framework. We would like to explore approaches that allow seamless support for legacy applications, with minimal efforts required from users. For instance, it would be preferable if it requires no modification to the source code of the applications or the operating systems.

Preliminary Solution: One feasible approach is to intercept a set of related system state and cross-node communications, and feed these information as streams of base tuples to the DS2 system.

Previous work has extensively explored mechanisms for acquiring user-specified system state from runtime systems. For instance, MaceMC [12] and P2 Monitor [30] require users to implement their systems using specific languages that can automatically capture system state; In Pip [27] and X-Trace [5], the source code of a target application needs to be modified or annotated. D3S [19] and MODIST [33] treat target applications as black box, and inject monitors in the underlying operating systems (or a middle-ware) to intercept user-specified system state.

Once the system state are captured, users can subsequently specify high-level logical derivation rules within our declarative platform, and leverage provenance information, which can be efficiently maintained and queried within DS2, to perform system analysis.

6. CONCLUSION

In this paper, we present DS2, a unified declarative platform for specifying, implementing and analyzing large-scale secure distributed systems, with built-in support for efficient provenance maintenance and querying. Our contributions are: (1) We introduce a unified language for distributed systems and security policies, and we show secure query processing techniques for distributed settings; (2) The notion of network provenance is conceptualized and developed within DS2, with flexibility for query customization and optimizations. We also discuss future research plans on secure provenance and DS2's integration with legacy applications.

Our preliminary evaluation [34, 35] on LAN and the PlanetLab testbed has demonstrated that the authenticated communication, as well as the provenance support, incurs little overhead. An initial DS2 prototype is available as open-source [4].

[2]Specifically in our DS2 system, the deterministic state machine can be presented as a *SeNDlog* program.

7. REFERENCES

[1] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

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

[3] J. DeTreville. Binder: A logic-based security language. In *IEEE S&P*, 2002.

[4] DS2 open-source implementation. `http://netdb.cis.upenn.edu/rapidnet/downloads.html`.

[5] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[6] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, 2009.

[7] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical Accountability for Distributed Systems. In *SOSP*, 2007.

[8] M. Huang, A. Bavier, and L. Peterson. PlanetFlow: Maintaining accountability for network services. *Communications of the ACM*, 32(6), 1989.

[9] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.

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

[11] T. Jim. SD3: A Trust Management System With Certified Evaluation. In *IEEE S&P*, May 2001.

[12] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM TOCS*, 18(3), 2000.

[14] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *IMC*, 2005.

[15] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM TOCS*, 10(4), 1992.

[16] P. Laskowski and J. Chuang. Network monitors and contracting systems: Competition and innovation. In *SIGCOMM*, 2007.

[17] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM TISSEC*, 6(1), 2003.

[18] M. Liu, W. Zhou, N. Taylor, Z. Ives, and B. T. Loo. Recursive Computation of Regions and Connectivity in Networks. In *ICDE*, 2009.

[19] X. Liu, Z. Guo, X. Wang, F. Chen, X. L. J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.

[20] 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 *SIGMOD*, June 2006.

[21] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[22] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.

[23] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. SecureBlox: Customizable Secure Distributed Data Processing. In *SIGMOD*, 2010.

[24] W. R. Marczak, D. Zook, W. Zhou, M. Aref, and B. T. Loo. Declarative reconfigurable trust management. In *CIDR*, 2009.

[25] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *VLDB*, 2003.

[26] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2), 1993.

[27] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[28] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *SIGCOMM*, 2000.

[29] M. Sherr, A. Mao, W. R. Marczak, W. Zhou, B. T. Loo, and M. Blaze. A3: An Extensible Platform for Application-Aware Anonymity. In *NDSS*, 2010.

[30] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using Queries for Distributed Monitoring and Forensics. In *EuroSys*, 2006.

[31] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[32] Y. Xie, V. Sekar, M. Reiter, and H. Zhang. Forensic analysis for epidemic attacks in federated networks. In *ICNP*, 2006.

[33] J. Yang, T. Chen, M. Wu, Z. Wu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

[34] W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *ICDE*, 2009.

[35] 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 *SIGMOD*, 2010.

[36] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. DMaC: Distributed Monitoring and Checking. In *RV*, 2009.