Unified Declarative Platform for Secure Networked Information Systems

Wenchao Zhou* Yun Mao* Boon Thau Loo* Martín Abadi[†][‡]
*University of Pennsylvania [†]UC Santa Cruz [‡]Microsoft Research
{wenchaoz, maoy, boonloo}@cis.upenn.edu, abadi@microsoft.com

Abstract— **We present a unified declarative platform for specifying, implementing, and analyzing secure networked information systems. Our work builds upon techniques from logic-based trust management systems, declarative networking, and data analysis via provenance. We make the following contributions. First, we propose the Secure Network Datalog (*SeNDlog*) language that unifies Binder, a logic-based language for access control in distributed systems, and Network Datalog, a distributed recursive query language for declarative networks. *SeNDlog* enables network routing, information systems, and their security policies to be specified and implemented within a common declarative framework. Second, we extend existing distributed recursive query processing techniques to execute *SeNDlog* programs that incorporate authenticated communication among untrusted nodes. Third, we demonstrate that distributed network provenance can be supported naturally within our declarative framework for network security analysis and diagnostics. Finally, using a local cluster and the PlanetLab testbed, we perform a detailed performance study of a variety of secure networked systems implemented using our platform.**

I. Introduction

In recent years, we have witnessed a proliferation of networked information 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 due to emerging security threats.

In response, there have been several proposals aimed at evolving the underlying network infrastructure to provide better support for accountability [1], efficient packet tracing [2] and flow analysis [3], all of which are geared towards better tools for analyzing and securing networks. Surprisingly few of these proposals have been integrated in a useful manner in existing networked information systems, or have any significant impact on the design of new distributed query engines. We argue that the reasons are two-fold. First, 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. Second, they are often afterthought, 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 networked information systems with security policies, we present a unified declarative platform for specifying, implementing, analyzing and auditing large-scale secure information systems. Our work has largely been inspired by recent efforts at using declarative languages that are aimed at simplifying the process of specifying and implementing security policies and networks. Our paper builds upon and unifies three bodies of work: (1) *logic-based trust management systems* [4], [5], [6], [7], [8], [9], (2) *declarative networking* [10], [11], [12], and (3) database techniques for analyzing data computations via the concept of *provenance* (or *lineage* [13]). From a practical standpoint, this integration has several benefits, ranging from ease of management, one fewer language to learn, one fewer set of optimizations, finer-grain control over the interaction between security and network protocols, and the potential of doing analysis, optimizations, and auditing across levels.

Access control is central to security and it is pervasive in computer systems. Over the years, logical ideas and tools have been used to explain and improve access control. Several logic-based languages (e.g. Binder [7], Cassandra [9], D1LP [5], RT [8], SD3 [4], SecPAL [6]) have been proposed to ease the process of expressing, analyzing and encoding access control policies. Similarly, the Network Datalog (*NDlog*) declarative networking language also has its roots in logic programming. *NDlog* is a distributed variant of Datalog for expressing recursive queries [14] over network graphs, hence allowing compact, clear formulations of a variety of routing protocols and overlay networks which themselves exhibit recursive properties.

Despite being developed by two different communities and used for different purposes, logic-based access control languages and *NDlog* extend Datalog in surprisingly similar ways: by supporting the notion of context (location) to identify components (nodes) in distributed systems. This suggests the possibility of unifying these languages to create an integrated system, exploiting good language features, execution engine, and optimizations. In addition, our unification will dispense with much of the special machinery proposed for access control, and instead rely on distributed database engines to process these policies, leveraging well-studied query processing and optimization techniques. It has been shown previously [15] that access control languages such as Binder share similarities to data integration languages, further suggesting that query processing and optimization techniques are directly applicable here. The contributions of this paper are as follows:

- **Unified declarative framework:** We propose the Secure Network Datalog (*SeNDlog*) language that unifies logic-based access control and declarative networking languages, hence enabling networked information systems to be specified within a unified declarative framework. We demonstrate the flexibility and compactness of *SeNDlog* via secure specifications of the path-vector routing protocol, Chord distributed hash table (DHT) [16], and the PIER [17] distributed query processor. Our system additionally provides support for dynamically layering multiple overlays at runtime (e.g. PIER over Chord), hence enabling security policies to be enforced and integrated across various layers.

- **Authenticated distributed query processing:** We extend the *pipelined semi-naïve* (PSN) [12] evaluation used in declarative networks for asynchronous pipelined evaluation of distributed recursive queries, to incorporate *authenticated communication* into query execution. In our proposed *authenticated PSN* extension, all communication generated by the execution plans is digitally signed and authenticated to ensure that the identities of participating nodes can be determined and validated.
- **Diagnostics with network provenance:** The dataflow framework used in declarative networking captures information flow as distributed queries. Hence, it is natural to utilize *data provenance* [13] to "explain" the existence of any network state, which is analogous to the use of *proof-trees* [4] in security audits. We propose a taxonomy of data provenance and demonstrate that they map into use cases for secure networking, including real-time diagnostics, forensics, and trust management.
- **Prototype evaluation:** We have developed a prototype based on the P2 declarative networking system [12]. On a local cluster and PlanetLab [18], we experimentally validate declarative secure implementations of the path vector protocol, Chord, and PIER query processor, as well as network provenance support in our system.

The paper is organized as follows. In Section II, we first present a background overview of trust management and declarative networking languages, focusing on the Binder [7] and *NDlog* [12]. In Section III, we present the unified *SeNDlog* language, and present several examples in Section IV that illustrate usage of the language. In Section V, we demonstrate how *SeNDlog* queries are compiled into authenticated dataflows to implement a variety of secure information systems. Section VI outlines analysis opportunities that are enabled with the integration of network provenance into our system. We present evaluation results in Section VII.

II. Background

As background, we first introduce *Binder*, a representative logic-based access control language, and the *NDlog* declarative networking language. These two languages serve as the basis for the unified declarative language described in the next section.

Binder and *NDlog* are query languages based on Datalog [14]. A Datalog program consists of a set of declarative *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and ... and `qn` implies `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. In Datalog, rule predicates can be defined with other predicates in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. 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. An optional `Query` rule specifies the output of interest (i.e. result tuples).

A. Binder: Access Control Language

Binder program is a set of Datalog-style logical rules. In addition, Binder has a notion of *context* that represents a component in a distributed environment and a distinguished operator `says`. For instance, in Binder we can write:

```
b1 access(P,O,read) :- good(P).
b2 access(P,O,read) :- bob says access(P,O,read).
```

The `says` operator implements a common logical construct in authentication [19], where we assert "`p says s`" if the principal `p` supports the statement `s`. The above rules `b1` and `b2` can be read as "any principal `P` may access any object `O` in `read` mode if `P` is good or if `bob` says that `P` may do so". The `says` operator abstracts from the details of authentication.

A principal in Binder refers to a component in a distributed environment. Each principal has its own local *context* where its rules reside. Binder assumes an *untrusted* network, where different components can serve different roles running distinct sets of rules. Because of the lack of trust among nodes, a component does not have control over rule execution at other nodes. Instead, Binder allows separate programs to interoperate correctly and securely via the export and import of rules and derived tuples across contexts. For example, rule `b2` can be a local rule that is executing in the context of principal `alice`, which imports derived `may-access` tuples from the principal `bob` into its local context via `bob says may-access(p,o,read)` in its rule body. In one specific implementation, communication happens via signed certificates, where derived tuples and rules signed using the private key of the exporting context can be imported by another context and checked using the corresponding public key.

B. NDlog: Declarative Networking Language

The high level goal of *declarative networks* is to build extensible network architectures that achieve a good balance of flexibility, performance and safety. Declarative networks are specified using *Network Datalog* (*NDlog*), which is 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 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 [10], and the Chord DHT in 47 lines of code [11]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations, while achieving orders of magnitude reduction in code size. To illustrate, the following two *NDlog* rules compute all pairs of reachable nodes:

```
r1 reachable(@S,D) :- link(@S,D).
r2 reachable(@S,D) :- link(@S,Z), reachable(@Z,D).
Query reachable(@S,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links, and rule `r2` expresses that "if there is a link from `S` to `Z`, and `Z` can reach `D`, then `S` can reach `D`." By modifying this example, it has been shown in previous work [10] that we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

NDlog supports a *location specifier* in each predicate, expressed with `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all `reachable` and `link` tuples are stored based on the `@S` address field. The output of interest (indicated by the rule `Query`)[1] is the set of all `reachable(@S,D)` tuples, representing reachable pairs of nodes from `S` to `D`. The above *NDlog* program is executed as distributed stream computations, where streams of `link` and `reachable` tuples are joined at different nodes to compute routing tables.

C. Comparing Binder and NDlog

Having introduced Binder and *NDlog*, we highlight some similarities and differences between these two languages to set the stage for the unified language design in the next section.

- **Trusted vs untrusted networks:** Since Binder and *NDlog* are designed for distributed settings, Binder's notion of *context* is similar to *NDlog*'s notion of *location specifier*. However, *NDlog* is designed for a fully trusted environment, where generated tuples are blindly accepted by nodes based on their specified locations. Binder assumes an untrusted network, where rules are executed with their own context, and communication happens via the use of "`says`" which requires authentication.
- **Export of derived tuples:** In Binder, security policies are not integrated with the policy for exporting data. To illustrate, in rule `b2`, it is not possible to express the restriction that principal `alice` can export `may-access(P,O,read)` only to principal `bob`. Such restrictions on messages to selected recipients are important in secure network protocols, both for performance and secrecy. *NDlog* achieves that with the use of location specifiers at the rule head.
- **Bottom-up vs top-down evaluation:** Most practical access control languages utilize a top-down (or backward-chaining) evaluation strategy. Specific requests are made as goals, which are then resolved against the security policies, hence minimizing the disclosure of sensitive information. On the other hand, declarative networking protocols adopt a bottom-up evaluation strategy, which is a better fit for the incremental continuous execution model of network protocols. To achieve goal-oriented operations in a bottom-up evaluation engine, there are well-known database optimizations such as magic-sets [20] which rewrites existing rules based on the output of interest to avoid computing redundant facts while leveraging set-at-a-time operations.

[1]Note that the `Query` rule is not mandatory, in which case, all derived output tuples will be continuously computed and maintained in the network.

III. Secure Network Datalog

The *SeNDlog* language unifies Binder and *NDlog* with the following goals in mind. First, the language is as expressive as Binder and *NDlog*. Second, *SeNDlog* must support authenticated communication and enable the differentiation of principals according to their roles in trusted and untrusted networked environments. Note that we have chosen Binder as a starting point for *SeNDlog* because of its simple language design and similarities to *NDlog*. Despite its simplicity, we show in the next section that the unified language can support a variety of networked systems and security policies.

A. Rules within a Context

In *SeNDlog*, we bind a set of rules and the associated tuples to reside at a particular node. We do this at the top level for each rule (or set of rules), for example by specifying:

```
At N,
  r1 p :- p1,p2,...,pn.
  r2 p1 :- p2,p3,...,pn.
```

The above rules `r1` and `r2` are in the context of `N`, where `N` is either a variable or a constant representing the principal where the rules reside. If `N` is a variable, it will be instantiated with local information upon rule installation. In a trusted distributed environment, `N` simply represents the network address of a node: either a physical address (e.g. IP addresses) or a logical address (e.g. overlay identifier). In a multi-user multi-layered network environment where multiple users and overlay networks may reside on the same physical node, `N` can include the user name and an overlay network identifier. This is unlike declarative networking, where location specifiers denote physical IP address. To differentiate from *NDlog*'s use of location specifiers, we refer to such forms of addressing in *SeNDlog* as *composite location specifiers* [21]. As we will demonstrate later, composite location specifiers enable one to specify networked information systems that involve multiple principals and authentication at different network layers.

B. Communicating Contexts

Similar to Binder, the *SeNDlog* language allows different principals or contexts to communicate via import and export of tuples. The communication serves two purposes: (1) maintenance messages as part of a network protocol's updates on routing tables, and (2) distributed derivation of security decisions. Imported tuples from a principal `N` are automatically quoted using "`N says`", to differentiate them from local tuples. During the evaluation of *SeNDlog* rules, we allow derived tuples to be communicated among contexts via the use of *import predicates* and *export predicates*:

Definition 1: An *import predicate* is of the form "`N says p`" in a rule body, where principal `N` asserts the predicate `p`.

Definition 2: An *export predicate* is of the form "`N says p@X`" in a rule head, where principal `N` exports the predicate `p` to the context of principal `X`. Here, `X` can be a constant or a variable. If `X` is a variable, in order to make bottom-up evaluation efficient, we further require that the variable `X` occur in the rule body. As a shorthand, we can omit "`N says`" if `N` is the principal where the rule resides.

The use of export predicates ensures confidentiality and prevents information leakage, by only exporting tuples to specified principals. With the above definitions, a *SeNDlog* rule is a Datalog rule where the rule body can include import predicates, and the rule head can be an export predicate. We provide a concrete example with the following rules:

```
At N,
 e1 p(X,Y) :- p1(X), p2(Y).
 e2 p(X,Y,W) :- Y says p1(X), Z says p2(W), Z!=N.
 e3 p(Y,Z)@X :- p1(X), Y says p2(Z).
 e4 Z says p(Y)@X :- Z says p(Y), p1(X).
```

Rule `e1` is a traditional Datalog rule. Rule `e2` contains two predicates `p1` and `p2` imported from `Y` and `Z` respectively. The output of `e1` and `e2` are stored locally as `p`. Rules `e3` and `e4` contain an import predicate each, and export their derived heads to `X`. Note that, in rule `e4`, the export principal `Z` differs from the principal `N`. To ensure that `p` is indeed asserted by `Z`, we introduce the *honesty constraint* in all *SeNDlog* rules:

Definition 3: A *SeNDlog* rule in the context of principal `N` is *honest* if the following condition is satisfied: if the rule head is "`X says p`", where `X` is a constant or a variable, either `X` is `N`, or "`X says p`" occurs in the body of the rule.

The honesty constraint enables a simple implementation. Specifically, for security, whenever a principal other than `N` exports `N says p`, it should provide a proof that this is the case; the proof is a signature by `N`. With this constraint, the principal may simply forward the signature that corresponds to the occurrence of `N says p` in the rule body. Like *NDlog*, *SeNDlog* allows derived tuples to be exported to *specific* nodes via the export predicates. This is done as a way of enforcing secrecy and also performance (avoiding broadcast of tuples).

C. Different Levels of Says

The implementation of "`says`" may depend on the system and its context. Ideally, *SeNDlog* should support a heterogeneous network where nodes have different security levels. In a hostile world, "`says`" may require digital signatures. For example, in rule `e3` from Section III-B, `N` should check that `p2` indeed came from `Y` by checking the signature of the imported tuple against `Y`'s public key. In a more benign world, "`says`" may simply append a cleartext principal header to a message—and this will of course be cheaper. Somewhere in between, the use of digital signatures may be applied only to certain important messages: there is a trade-off between security and efficiency, and the language does not provide any leverage in deciding how that trade-off should be made. Note however that the policy writer could easily provide hints along with rules, indicating that some "`says`" are more important than others.

Going further, one can support multiple "`says`" constructs with different security levels. This requires the export predicate be explicit about the security level, e.g. "`N says0 p@X`" exports predicate `p` to `X` with security level 0 (which may simply involve a cleartext principal header with no signatures), and `X` can only import `p` in its rule body via an authentication scheme at the same or higher security level.

IV. Practical SeNDlog Examples

We provide three *SeNDlog* example specifications of secure networked systems. Our examples are based on modifications to existing declarative networks to incorporate various security policies. This is by no means intended to be an exhaustive coverage of the possibilities. Our main goal here is to illustrate flexibility and compactness of *SeNDlog*, and to illustrate the key language features of *SeNDlog*.

A. Path-Vector Routing Protocol

Our first example is based on the declarative path vector protocol as presented in the original declarative routing [10] paper. At every node `Z`, this program takes as input `neighbor(Z,X)` tuples that contain all neighbors `X` for `Z`. The program generates `route(Z,X,P)` tuples, each of which stores the path `P` from source `Z` to destination `X`. The basic protocol specification is similar to the all-pairs reachable example presented in Section II-B, with additional predicates for computing the actual path using the `f_concat` function which prepends neighbor `X` to the input path `P`.

The input `carryTraffic` and `acceptRoute` tables are used to represent the export and import policies of node `Z` respectively. Each `carryTraffic(Z,X,Y)` tuple represents the fact that node `Z` is willing to serve all network traffic on behalf of node `X` to node `Y`, and each `acceptRoute(Z,Y,X)` tuple represents the fact that node `Z` will accept a route from node `X` to node `Y`. A more complex version of this protocol will have additional rules that derive `carryTraffic` and `acceptRoute`, avoid path cycles and also derive shortest paths with the fewest hop counts.

```
At Z,
 z1 route(Z,X,P) :- neighbor(Z,X), P=f_initPath(Z,X).
 z2 route(Z,Y,P) :- X says advertise(Y,P),
      acceptRoute(Z,X,Y).
 z3 advertise(Y,P1)@X :- neighbor(Z,X), route(Z,Y,P),
      carryTraffic(Z,X,Y), P1=f_concat(X,P).
```

The path-vector protocol is used for inter-domain routing over the Internet, and it is known to be vulnerable to a variety of attacks due to the lack of mechanisms for verifying the authenticity and authorization of routing control traffic. One potential solution is to authenticate every routing control message, as proposed for Secure BGP [22].

In our example program, we can specify such authentication naturally via the use of "`says`" that ensures that all `advertise` tuples are verified by the recipients for authenticity. Rule `z1` takes as input `neighbor(Z,X)` tuples, and computes all the single hop `route(Z,X,P)` containing the path `[Z,X]` from node `Z` to `X`. Rules `z2-z3` are used to compute routes of increasing hop counts. Upon receiving an `advertise(Y,P)` tuple from `X`, `Z` uses rule `z2` to decide whether to accept the route advertisement based on its local `acceptRoute` table. If the route is accepted, a `route` tuple is derived locally, and this results in the generation of an `advertise` tuple which is further exported by node `Z` via rule `z3` to some of its neighbors `X` as determined by the policies stored in the local `carryTraffic` table.

B. Chord DHT

Our second example is based on a declarative Chord implementation originally presented in 47 rules. Our modifications avoid a security weakness in a DHT where malicious nodes can occupy a large range of the key space [23]. 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*[2] `LI`. Each node runs its respective set of rules as follows:

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

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

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

In rule `ni1`, a node `NI` that wishes to join the Chord network first exports a `requestCert` tuple to its `CA` (as indicated in the entry in its `MyCA` table) to request *nodeID certificates*. Upon receiving the request, the `CA` generates a `nodeIDCert(NI,N,K)` tuple containing the nodeID certificate, which is then exported back to node `NI`. The `nodeIDCert(NI,N,K)` tuple contains the address of node `NI`, the corresponding public key `K`, and a generated identifier `N` randomly chosen from the keyspace using the function `f_generateID(K,S)` that takes as input the public key of `K` and a previously exchanged secret `S` known only to the `CA` and `NI`.

Upon importing the `nodeIDCert` tuple from the `CA`, using rule `ni2`, node `NI` initializes its local node identifier which is stored as a `nodeID(NI,N)` tuple. It also forwards the `nodeIDCert` to its landmark node `LI` in order to join the chord network. At the landmark node `LI`, `nodeIDCert` is imported and checked for authenticity. If `nodeIDCert` is accepted, the landmark node derives an `acceptJoinRequest(NI)` tuple that can further be used to generate a lookup request to locate the successor node on behalf of node `NI`.

Once node `NI` successfully joins the Chord network, the rules as presented [11] can then be used by node `NI` to implement the rest of the Chord protocol (see [11] for detailed specification). Beyond addressing this specific weakness of Chord, one can further modify the current declarative Chord specifications such that authentication happens among *all* Chord nodes. In this case, all message exchanged between any Chord nodes has to be similarly communicated via the use of `says`, and digitally signed by each node.

C. PIER Distributed Query Processor

Our third example is based on the distributed join capabilities supported by the PIER [17] query processor. In an untrusted p2p environment, authenticated communication is essential for ensuring that every PIER node only processes tuples generated by other PIER nodes that they trust. Our program takes as input two tables `tableA` and `tableB` owned by principals `alice` and `bob` and performs a distributed join:

```
At alice,
 a1 storeA(X,Y)@NI :- tableA(X,Y), K=f_sha(X),
      NI=Chord::K.
```

[2]In any Chord DHT, there are a number of landmark nodes that are known apriori. All nodes join the Chord network via these landmarks.

```
At bob,
 b1 storeB(X,Y)@NI :- tableB(X,Y), K=f_sha(X),
      NI=Chord::K.

At NI,
 r1 result(X,Z)@r :- alice says storeA(X,Y),
      bob says storeB(Y,Z).
```

Rules `a1` and `b1` are executed by `alice` and `bob` to store all `tableA` and `tableB` tuples in the DHT. `Chord::K` is an example of a *composite location specifier* that we introduced in Section III-A, where `Chord` denotes the Chord overlay, and `K` denotes a logical address in the form of a Chord identifier. Both `alice` and `bob` compute the SHA-1 hash of the first attribute `X` to generate a Chord identifier `K`. The resulting `storeA` and `storeB` tuples are then sent to the node `NI` which is responsible for storing tuples for the Chord identifier `K`. These tuples include the signatures of `alice` and `bob`, which are verified before being stored at `NI`.

The join query (denoted by rule `r1`) is issued by node `r`. This rule is disseminated and executed on all PIER nodes `NI`, resulting in a distributed join of all `tableA` and `tableB` tuples indexed by the DHT. The `result` tuples computed from the join are sent back to the requesting node `r`.

Layered authentication

The above *SeNDlog* program demonstrates *layered authentication*: the PIER query processor is layered over the Chord DHT, and authentication can happen at either the PIER or Chord layer. For example, the rules `a1` and `b1` enable the authentication of every published tuple generated by `alice` and `bob`. This authentication happens at the PIER query processing layer, and the underlying Chord is agnostic to it. On the other hand, to index the tuples of both `alice` and `bob` in the DHT, the rules `a1` and `b1` themselves invoke Chord's lookup for the key `K`. In this case, Chord routing messages that result from executing these rules may require further authentication among the underlying Chord nodes.

Distributed join

The above rule `r1` specifies a distributed join of two tables. In the actual PIER system, this requires "rehashing" (or repartitioning) both tables based on their respective join attributes. The following rules perform this repartitioning step followed by the actual distributed join:

```
At NI,
 r1a alice says rehashA(X,Y)@RI :-
      alice says storeA(X,Y), K=f_sha(Y), RI=Chord::K.

At RI:
 r1b result(X,Z)@r :- alice says rehashA(X,Y),
      bob says storeB(Y,Z).
```

In rule `r1a`, all `storeA` tuples are rehashed as `rehashA` tuples based on the hash of the join attribute `Y`. All rehashed tuples retain the original signature of `alice`, adhering to the *honesty constraint* (Section III-B). The join is performed at node `RI` with local `storeB` tuples, and the resulting `result` tuples are sent back to the requesting node `r`.

V. Secure Query Processing

Having presented the *SeNDlog* language and three representative protocols written in the language, we next describe query processing techniques for executing *SeNDlog* programs. Our proposed *Authenticated Pipelined Semi-naïve* (APSN) is an adaptation of the *Pipelined Semi-naïve* (PSN) [12] proposed for declarative networks.

A. Background on PSN

We briefly introduce PSN before describing APSN. Unlike traditional semi-naïve evaluation, PSN does not require computations in synchronous rounds (or iterations), a prohibitively expensive operation in distributed settings. We consider the following recursive Datalog rule:

$$d \text{ :- } d_1, d_2, ..., d_n, b_1, b_2, ..., b_m$$

where there are n derived predicates $(d_1, ..., d_n)$, and m *base predicates* $(b_1, ..., b_m)$ in the rule body. Derived predicates refer to intensional relations that are derived during rule execution, and may be mutually recursive with d. Base predicates refer to extensional (stored) relations whose values are not changed during rule execution. In PSN, a delta rule is generated for each derived predicate, where the k^{th} *delta rule* is of the form:

$$\triangle d \text{ :- } d_1, .., d_{k-1}, \triangle d_k, d_{k+1}, .., d_n, b_1, b_2, ..., b_m$$

where $\triangle d_k$ denotes a tuple $t_k \in d_k$ that is used as input to the rule for computing new d tuples.

In the simplest version of PSN with no buffering, 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 a 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. Duplicate evaluations are avoided using local arrival timestamps, where each new tuple is only processed with tuples with older timestamps.

B. Authenticated PSN

APSN differs from PSN in its processing of tuples that are sent from one node to another. Consider the following *SeNDlog* rule in the context of principal p:
$At\ p,$
$a \text{ :- } d_1, ..., d_n, b_1, ..., b_n, p_1\ says\ a_1, p_2\ says\ a_2..., p_o\ says\ a_o.$
where there are n derived and m base predicates as before, 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:
$p\ says\ \triangle a \text{ :- } d_1, ..., d_n, b_1, ..., b_m,$
$p_1\ says\ a_1, ..., p_k\ says\ \triangle a_k, ..., p_o\ says\ a_o.$

The delta rule is similar as PSN with the additional use of `says` to authenticate new a_k tuples imported from p_k and sign any derived a tuples by the local principal p.

In the rest of the section, we will describe the overall dataflow architecture, and then show how APSN delta rules are compiled into execution plans.

Dataflow Architecture

Figure 1 shows an exampled dataflow that is automatically generated from the *SeNDlog* rules by our query processor. The dataflow execution model is based on that of the P2 declarative networking system [11]. In P2, queries are compiled and executed as *distributed dataflows* and share a similar execution model with the Click modular router [24]. 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.

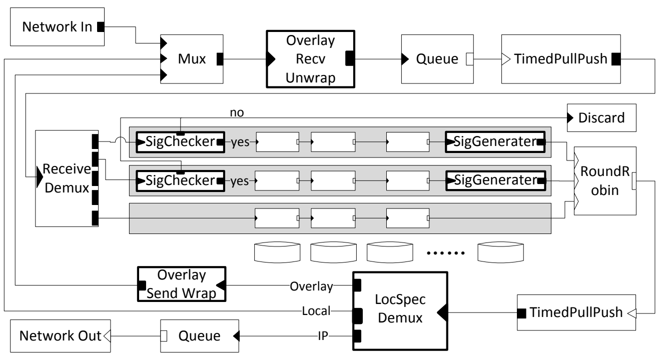

Fig. 1. Dataflow execution plan for a single node.

To support layering of multiple networks, the execution plan includes three additional operators shown in Figure 1. The operators `OverlayRecvUnwrap` and `OverlaySendWrap` are used for de-encapsulation and encapsulation of tuples that are sent across different network layers. They are used in conjunction with the `LocSpecDemux` operator, which demultiplexes tuples based on the composite location specifier to the rule strands executing the appropriate network protocol. For example, if a tuple has a composite location specifier `Chord::K`, it will be routed to the local dataflow strands for Chord which will process and forward this tuple based on the Chord protocol. More implementation details on layering support is available in reference [21].

APSN Rule Strands

At the core of the dataflow are *rule strands* shown within the gray box, which are directly compiled from the PSN and APSN delta rules into a series of relational operators such as joins, aggregations, selections, and projections. Messages flowing among dataflows are executed at different nodes, resulting in updates to local tables, or query results that are returned to the hosts that issued the queries. The local tables store the state of intermediate and computed query results, which include the network state of various network protocols. Each tuple has an associated lifetime declared at creation time of each table[3]. Each incoming tuple is then stored locally for its lifetime.

In traditional PSN, each delta rule will be compiled into two rule strands: one for incremental insertion and one for deletion. Given an input tuple, the output of executing a strand would either be local state modifications (insertions/deletions

[3]A zero lifetime tuple is treated as an event that will trigger rules and be discarded. An infinite lifetime tuple is stored locally until explicitly deleted. In between, tuples may be maintained as soft-state with a fixed lifetime, similar to time-based sliding windows in stream processing systems (e.g. [25]).

to local tables), or generation of new messages which are then transmitted via the `Network-Out` operator.

APSN rule strands (shown as the first two gray boxes in Figure 1) differ from PSN in the use of authenticated communication. This requires two additional operators shown in **bold** in the dataflow:

- The `SigGenerator` operator is used to sign outgoing tuples based on the private key of the local principal. Any outgoing tuple `t` that requires authentication is communicated as a (`p,s,t`) triplet, where `p` corresponds to the source principal, `s` is the signature generated by encrypting a message digest (essentially a cryptographic hash of `t`) with `p`'s private key.
- At the recipient node, the `SigChecker` operator authenticates incoming (`p,s,t`) triplets by decrypting `s` with the public key of `p`, and verifying that the decrypted contents matches the corresponding message digest.

The `SigGenerator` and `SigChecker` use a public-key cryptography scheme that can be computationally expensive. The `SigChecker` operator requires knowledge of the public key of the principal whom the dataflow is importing tuples from. Note that key management is an orthogonal problem to our work. In the simplest model, each node has full apriori knowledge of all the public keys of all other communicating peers. An alternative approach involves the runtime system to fetch public keys on demand (upon receiving a tuple that requires authentication) from a known CA, and cache the public key for future use.

Our declarative framework and runtime system do not preclude the use of other lightweight key management schemes or alternative schemes such as the *message authentication code* (MAC) which is computationally less expensive. MAC enables APSN to be more efficient but requires the use of shared symmetric keys among principals that wish to communicate. To support different levels of "`says`", an optional security level attribute can be included to the output of the `SigGenerator` operator. The `SigChecker` operator can then apply the appropriate level of authentication.

C. Example: Path-Vector Routing

We provide an example based on rule `z2` used in the path vector protocol presented in Section IV-A. The APSN delta rules for rule `z2` are as follows:

$z2a\ \triangle route(Z,Y,P)$:- $X\ says\ \triangle advertise(Y,P)$,
$\qquad acceptRoute(Z,X,Y)$.
$z2b\ \triangle route(Z,Y,P)$:- $X\ says\ advertise(Y,P)$,
$\qquad \triangle acceptRoute(Z,X,Y)$.

Since rule `z2` takes as input two predicates `advertise` and `acceptRoute`, two delta rules (shown above) are generated. These two delta rules are used to generate the dataflow strands labeled `z2a@Z` and `z2b@Z` in Figure 2. The first strand (`z2a@Z`) takes as input `advertise(Y,P)` tuples from the network. After each `advertise` tuple is verified by the `SigChecker` operator based on the public key of `X` retrievable from the local store, the `advertise` tuple is inserted via the `Insert` operator. All new `advertise` tuples not seen previously will be joined with matching tuples from the local `acceptRoute` table to generate new `route` tuples which are further inserted into the local `route` table.

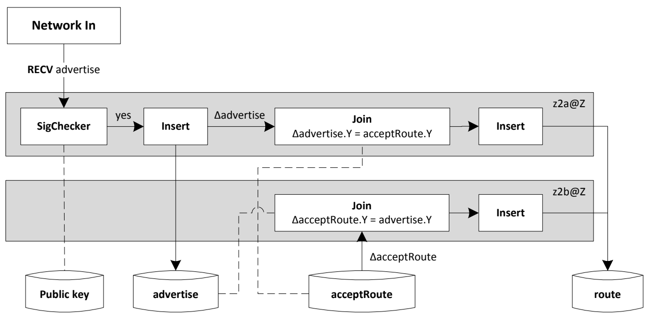

Fig. 2. Generated Rule Strands for rule `z2`.

The second strand (`z2b@Z`) takes as input any tuple updates stored in its local `acceptRoute` table. Since the table is local, no authentication is required. A similar join is performed with the local `advertise` table to generate new `route` tuples. Since both delta rules insert into the local `route` table rather than export the output tuples to remote principals, they need not be signed. The rule strand for rule `z3` would require a `SigGenerator` operator to generate the signature using the private key of `Z` before being sent.

APSN evaluation preserves the *authenticity* of tuples derived by *SeNDlog* rules. A tuple is authentic if it is derived from a *SeNDlog* rule that takes as input local tuples (that are trusted by the local principal) or authenticated tuples imported from other principals. In our execution framework, each tuple can be derived in two ways: (1) a *delta rule* generated from a regular Datalog rule that takes as input only local tuples to generate new derivations, or (2) an *authenticated delta rule* that takes as input a combination of local tuples and authenticated tuples imported from remote principals. For example, in Figure 2, both rule strands `z2a@Z` and `z2b@Z` take as input local `acceptRoute` tuples, and perform a join with incoming `advertise` tuples which are authenticated via the `SigChecker` operator.

VI. Network Provenance

In this section, we demonstrate a variety of opportunities for cross-layer analysis on both security policies and network state within our unified declarative platform. The dataflow framework used in our system captures information flow naturally as distributed queries, which allows one to track the derivation of any network state for real-time and forensic analysis. *Network provenance* in *SeNDlog* is inspired by the database notion of *data provenance* to "explain" the existence of any network state. We capture network provenance naturally within our declarative framework as the building blocks of provenance such as derivation rules and authenticated derivations are already being used throughout the system. A similar notion of provenance in security has previously been formalized as *proof-trees* (e.g. [4], [26]), where the focus is on checking the correctness and authenticity of derived security policies.

A. Taxonomy and Use Cases

To illustrate network provenance, we make use of a *SeNDlog* version of the all-pairs reachable program introduced in Section II-B, with the additional use of `says` language construct. Given a three-node network consisting of three unidirectional links, Figure 3 shows the provenance in the form of a derivation tree for `reachable(a,c)` derived at node `a`.

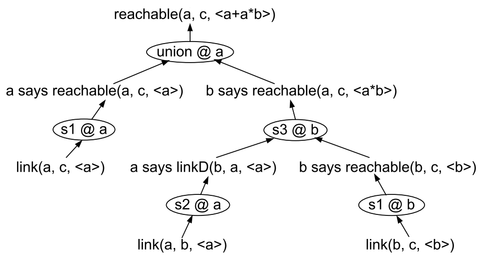

Fig. 3. *SeNDlog* provenance for `reachable(a,c)` in `a`'s context.

The base tuples are at the leaves of the tree, namely `link(a,b)` and `link(b,c)`. We annotate each operator (denoted by the oval) with the location (or context) where the rule is executed. Each node in the tree is asserted by a principal using "says". The provenance is *authenticated*, where individual nodes in the provenance tree need to validate the authenticity of the computed provenance using digital signatures. Each tuple in the derivation tree has an additional field denoted by `<...>` that encodes the principals used for deriving the tuple as an algebraic expression [27]. Here, we utilize `+` to represent a union, and `*` to represent a join. Based on this algebraic expression, each principal can determine the other principals involved in deriving the fact, and hence determine whether to accept or reject this fact based on existing trust relationships.

In addition to encoding authenticity, the above derivation tree can be stored in a *local or distributed* fashion. In *local provenance*, the entire derivation tree is stored at node `a`. This requires each tuple to contain its entire provenance when communicated from one node to another. Meanwhile, in *distributed provenance*, provenance information is fetched on demand, and only points to the child derivations which are potentially stored on another node. Distributed provenance requires no extra communication overhead during query execution, but incurs the additional cost of querying the provenance in a distributed fashion. On the other hand, local provenance has the advantage of enabling trust policies (e.g. accepting or rejecting incoming tuples based on source origin) and at each node locally without having to execute a potentially expensive distributed query.

We can further classify provenance as either *online* or *offline*. Online provenance is maintained for network state that is currently valid (i.e. not expired), and *offline* provenance is kept even when the derivations have expired. Online provenance enables real-time diagnostics to detect anomalies at runtime, whereas offline provenance enable one to perform forensic analysis where historical data is used to correlate traffic patterns of attackers [3].

B. Example: Network Traffic Traceback

We provide an example based on the *SeNDlog* routing program presented in Section IV-A. We modify the program to add additional rules for performing network traffic traceback [2]. The following *SeNDlog* program performs packet forwarding given the `forwarding` table at every router:

```
At Router,
 f1 packet(Pid,Dest,Data)@NextHop :-
         forwarding(Dest,NextHop), Router != Dest,
         P says packet(Pid,Dest,Data).
```

The `forwarding` table is computed by modifying the path vector protocol in Section IV-A to compute the `nextHop` router along the shortest path to any given destination (`Dest`). Rule `f1` forwards a packet `Pid` along the best path to `Dest` via the `nextHop` router. This payload is recursively routed by rule `f1` to the destination.

In order to traceback potential malicious traffic, one can annotate each outgoing `packet` tuple with its provenance. This can be in the form of *local provenance*, in which case the entire path (encoded in the packet's derivation tree) is forwarded with each `packet` tuple. In *distributed provenance*, states are maintained at every router to traceback the reverse path of each packet that is uniquely identified by its `pid`. If routers are untrusted, authentication can be achieved by adding a signature generated from each `packet`. Annotating every packet with its provenance can be expensive, both in terms of computation and bandwidth consumption. In the original IP traceback proposal, sampling is performed, where 1/20,000th packets are annotated with its provenance. Exploring such automatic optimizations in provenance computation is an interesting avenue of future work.

We note that a naïve implementation of network provenance may result in information leakage: the provenance of each tuple reveals information on the derivation rules used for its derivation. The leakage is especially problematic in the case of local provenance, where each tuple is shipped with its entire derivation tree, and nodes at remote locations can learn about derivation rules of nodes that are several hops away. For example, in the above traceback example, storing the provenance with each packet reviews the entire path traversed by each packet, and this may violate confidentiality policies of individual routing domains, and in addition, reveal import and export policies. If distributed provenance is used, such leakage can be mitigated since the child derivations require a network traceback which triggers a distributed query, and the source of each rule invocation can enforce access control over such queries. Another possibility that we are exploring is providing mechanisms where provenance information can be hidden based on security levels of principals, or stored only on specific trusted nodes.

VII. Evaluation

In this section, we present the evaluation of several secure networked information systems developed using our platform. Our evaluation is based on the P2 declarative networking system [11], with enhancements to support the *SeNDlog* language and APSN. Our dataflow execution utilizes two signature schemes: (1) the OpenSSL v0.9.8b cryptographic libraries that generate 1024-bit RSA signatures given input data, and (2) keyed-Hash Message Authentication Code (HMAC) that generates a 160-bit SHA-1 cryptographic hash from the input data and a secret key. This allows us to validate that our implementation can support signature schemes as described in Section V-B. For example, 1024-bit RSA signatures are the most expensive, while the HMAC approach is computationally less expensive but requires pair-wise symmetric keys. We further added runtime support to the P2 system to compute online, local and authenticated provenance (Section VI).

In previous work [10], [11], [12], it has been shown that performance of declarative networks is comparable to traditional imperative implementations. For example, a N-node declarative Chord implementation [11] resolves lookups via $O(logN)$ messages as expected, and the latency numbers are within the same order of magnitude as the published numbers [16] of the MIT Chord deployment. Hence, we focus on experimentally quantifying the additional overheads incurred by *SeNDlog*'s security extensions and network provenance computations, by comparing directly with existing declarative network implementations.

Our performance metrics are as follows:

- **Execution time (s):** Time taken for a *SeNDlog* program to reach a distributed fixpoint in a static network.
- **Average bandwidth utilization (KBps):** The per-node average bandwidth utilization for executing a distributed *SeNDlog* program in a dynamic setting where nodes are continuously joining and leaving the network.
- **Aggregate communication overhead (MB):** The total traffic generated when executing a single *SeNDlog* program to fixpoint in a stable network.

A. Experimental Setup

Our first set of experiments are executed within a local cluster with sixteen quad-core machines with Intel Xeon 2.33GHz CPUs and 4GB RAM running Fedora Core 6 with kernel version 2.6.20, which are interconnected by high-speed Gigabit Ethernet. The LAN environment allows us to isolate and examine CPU overhead of executing our platform within a controlled environment with high bandwidth. Our second set of experiments are conducted on the PlanetLab [18] testbed, where 80 geographically distributed nodes in Asia, Europe and North America are selected to execute our system. The PlanetLab testbed allows us to examine the real-world effects, such as bandwidth constraints and propagation delays imposed by geographic distances and queueing delays.

Our workload consists of the three *SeNDlog* programs presented in Section IV. The *Path-Vector* program involves a distributed recursive query for computing all-pairs shortest paths. The rules used in this query are similar to that presented in Section IV-A, with two additional local rules for computing the path with the minimum hop count for each source/destination pair of nodes. The *Chord* program extends the declarative Chord DHT [11] in *SeNDlog* with authenticated communication among all nodes (Section IV-B). The *PIER* program implements distributed query processing over the *Chord* DHT (Section IV-C).

To get an accurate measure of actual bandwidth utilization, we make use of the `tcpdump` system tool that allow us to intercept and measure all network packets transmitted from one P2 node to another.

B. LAN Experiments

In this section, we present the results of executing the *Path-Vector* query and *Chord DHT* on the local cluster. These two workloads are different: the first involves executing a distributed recursive query given a static network as input, while the second performs incremental maintenance of a Chord network and continuously issues Chord lookup requests in the background. Due to space constraints, we do not present PIER evaluation results on a LAN environment where the observations on performance overhead are similar to these two workloads. We will revisit an evaluation of PIER on PlanetLab later in this section.

Path-Vector Routing Protocol

Our first experiment consists of executing the *Path-Vector* query on all 16 cluster nodes. We vary the network size from 16 to 128, where each cluster machine runs up to 8 P2 virtual nodes. As input to the *Path-Vector* query, we initialized a `neighbor` table where each node has an outdegree of 3.

Figures 4 and 5 show the aggregate communication overhead and completion latency for the *Path-Vector* query averaged over 8 experimental runs. In *Auth-RSA1024*, RSA signatures of 1024 bits are generated from each tuple and verified by the recipient nodes. *Auth-MAC* represents the HMAC-based authentication scheme while *Auth-None* does not utilize any authentication. As a performance/security tradeoff illustration, we include *Auth-RSA512*, which uses 512-bit RSA signatures but is less secure compared to *Auth-RSA1024*.

We make the following observations. For the largest network size of 128, a lightweight authentication scheme such as *Auth-MAC* results in a 10% increase in aggregate communication overhead, and a negligible increase in execution completition time. On the other hand, the most expensive signature scheme *Auth-RSA1024* leads to a 40% increase in communication overhead, and increases the completion time from 10s to 34s.

Since our experiments are carried out in a high-speed network, the additional overhead is due largely to the generation (and not transmission) of the signatures. Given that authentication with 1024-bit signatures represents an upper-bound in performance overhead, we focus our evaluation on authentication overheads on this scheme, denoted as *Auth* in the rest of this section.

Chord DHT

Next, we study the performance characteristics of a *SeNDlog* implementation of the Chord protocol. The main difference compared to the previous *Path-Vector* query is the continuous execution flavor as opposed to a single program execution, where routing tables are incrementally maintained as nodes either and leave the network. Our experimented Chord network size consist of 128 nodes, and after all nodes have joined the Chord overlay at 1600s, random Chord lookup queries are issued simultaneously from 8 different nodes at three seconds interval for the next 400 seconds.

Figure 6 shows the per-node bandwidth utilization over time, obtained by averaging `tcpdump` statistics gathered across all nodes at any point in time. After all nodes have joined the Chord network (at time 800s), the per-node bandwidth utilization stabilizes at 0.2KBps and 0.4KBps respectively for

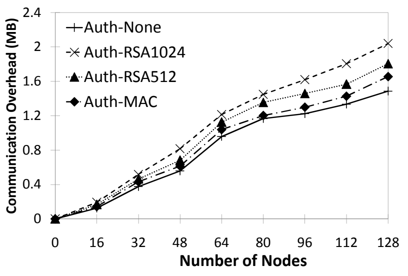

Fig. 4. Per-Node aggregate communication overhead (MB) for Path-Vector.

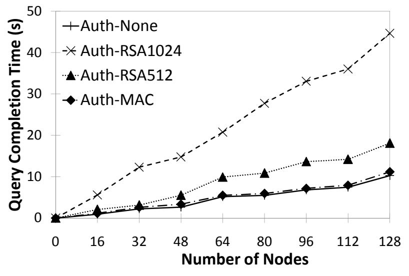

Fig. 5. Query completion time (s) for Path-Vector.

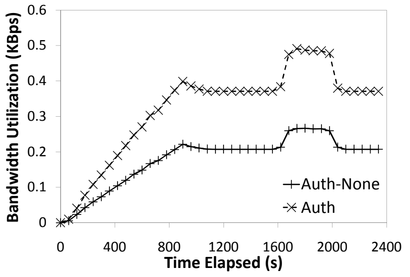

Fig. 6. Per-Node bandwidth utilization (KBps) for Chord on LAN.

Auth-None and *Auth*. At 1600s, Chord lookups are issued, where each lookup with random key is issued from a randomly chosen node to retrieve the IP address of the node responsible for the key. The bandwidth consumption increased due to the additional Chord lookups which require hop counts that are logarithmic [28] the size of the network on average.

We make two observations attributed to the smaller Chord messages (180 bytes per lookup message) and the continuous nature of the Chord protocol. First, *Auth* incurs twice as much bandwidth compared to *Auth-None*, which is larger compared to *Path-Vector*. Second, in Figure 7, the CDF of *Auth-None* and *Auth* of Chord lookup latency show that signature generation leads to a marginal increase in latency.

We observe that our earlier *Path-Vector* query workload incurs higher authentication overhead compared to Chord. This is largely due to the bandwidth intensive nature of the *Path-Vector* query, which is a distributed recursive query that executes to fixpoint over a large dataset. At 128 nodes, the average bandwidth utilization of the *Path-Vector* query is 45KBps, which is significantly higher compared to the bandwidth utilization of Chord which does periodic routing table maintenance. As a result, Chord incurs far lesser overhead in terms of signature generation, and this is reflected by the lower authentication overhead incurred by Chord.

C. PlanetLab Experiments

To examine the effects of bandwidth constraints and propagation delays over the wide-area, we deploy our *SeNDlog* implementations of Chord and PIER on PlanetLab. The Chord setup on PlanetLab is similar to that of the local cluster. In the PIER setup, we execute a distributed join of two tables with 100 tuples each, using the underlying *SeNDlog* Chord implementation as its routing layer. The distributed join query is issued after the Chord overlay has stabilized and the two input tables have been indexed by Chord.

Figure 8 shows the average bandwidth utilization of Chord on PlanetLab. Compared to a similar LAN experiment in Figure 6, the bandwidth utilization of 0.2KBps and 0.4KBps for *Auth-None* and *Auth* are identical, validating that Chord is executing correctly on PlanetLab in terms of protocol messages. In terms of Chord lookup latencies, we note that Chord lookups on PlanetLab (Figure 9) are an order of magnitude slower compared to a LAN environment (Figure 7). We attribute the increase to real-world effects, where competing slices on PlanetLab contend for limited bandwidth and CPU resources. Hence, the performance penalty of authenticated communication is more apparent.

We make similar observations on executing the PIER distributed join on PlanetLab. Figure 10 shows the execution progress of PIER in terms of percentage of results received: *Auth-None* requires an additional 30s to receive 50% of the result tuples. On PlanetLab, effects of congestion are more apparent, as reflected by the long-tailed distribution of the query results.

D. Network Provenance for Traffic Traceback

In our final experiment, we perform an evaluation of the *SeNDlog*-based traffic tracing query as described in Section VI-B in both the LAN and PlanetLab settings. This experiment serves to validate our runtime system support for network provenance, and also experimentally quantify the overhead of provenance computations concurrently with authenticated communication.

The experiment consists of 128 P2 virtual nodes executing on our 16-node cluster. The P2 nodes are used to emulate routers, where the *Path-Vector* query is executed on all 128 nodes to compute the `forwarding` tables used to determine the next hop along the shortest path from any given source to destination node. After the *Path-Vector* query has reached a distributed fixpoint, we execute the *SeNDlog* packet forwarding rule described in Section VI-B, where 800-byte packets are continuously injected and forwarded from random sources to destinations at a rate of 160 packets/second. The forwarding rules are executed with the following settings (based on the labels in Figures 11 and 12):

- **Regular** communicates all packets without authentication and provenance.
- **Prov** annotates *local provenance* (the derivation tree storing information of the entire path traversed) to every packet that is being forwarded.
- **AuthProv-MAC** and **AuthProv-RSA** annotates *local provenance with authentication* to every packet using HMAC and RSA signatures.

Figure 11 shows the CDF of packets delivery latency (the time elapsed between sending and receiving a packet) of the various schemes in a LAN environment. The additional overhead due to provenance generation and communication is negligible with marginal increase in the median latency. The use of HMAC for authenticated provenance leads to a 5% increase in median latency. In the worst case, using 1024-bit RSA signatures increases the median latency significantly,

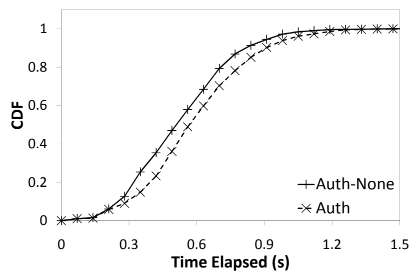

Fig. 7. CDF of Chord lookup latency on LAN.

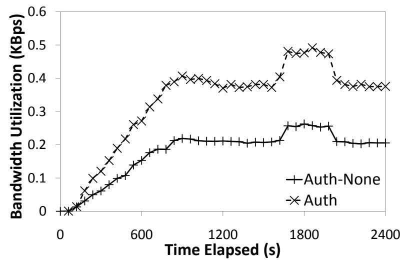

Fig. 8. Per-Node bandwidth utilization (KB/s) for Chord on PlanetLab.

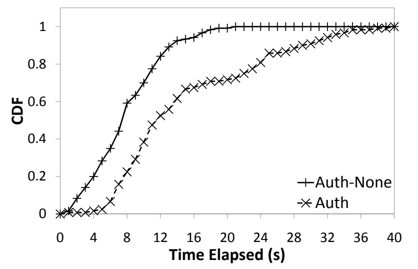

Fig. 9. CDF of Chord lookup latency on PlanetLab.

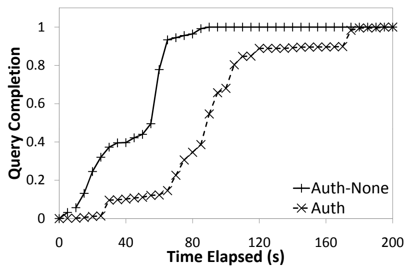

Fig. 10. Query progress over time for PIER on PlanetLab.

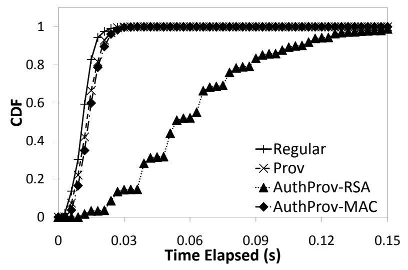

Fig. 11. CDF of packet delivery latency for traffic traceback on the local cluster.

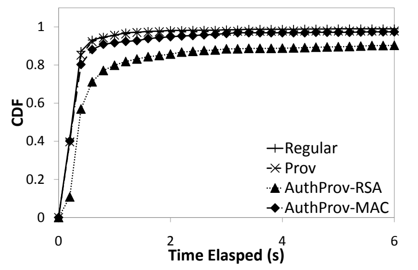

Fig. 12. CDF of packet delivery latency for traffic traceback on PlanetLab.

although in a high-bandwidth LAN environment, the absolute increase is less than 0.05 seconds.

On PlanetLab, the relative overhead of authenticated provenance is mitigated by the bandwidth limitations and larger WAN latencies, as shown in Figure 12. On PlanetLab, the relative increase in median latency is marginal, but on all schemes, we observed a long-tailed distribution in latencies due to network congestion.

E. Summary of Results

In summary, our experiments validate *SeNDlog* implementations of the path-vector protocol, Chord DHT, and distributed DHT-based joins. We note that the performance overhead of authentication depends on the type of authentication scheme used (e.g. HMAC vs RSA), and also the nature of the *SeNDlog* protocol being executed. A bandwidth-intensive distributed recursive query of large input tables will incur large overhead due to signature generation, while a continuous incremental protocol such as Chord DHT incurs less overhead. The overhead due to authentication may be mitigated with alternative authentication schemes that achieve different performance/security tradeoffs, and the use of tuple batching to amortize the cost of signature generation (at the expense of increased convergence times).

In addition, we validate and evaluate experimentally our system support for distributed authenticated provenance. On PlanetLab, the relative overhead of authentication and provenance is lower compared to a LAN environment, as the increased overhead is amortized by the larger inter-node latencies.

VIII. Related Work

This paper builds upon our initial *SeNDlog* language design [29] with an exploration of new use cases in various networked information systems such as performing distributed joins in an authenticated fashion, layered authentication, query processing techniques, as well as an evaluation of a prototype implementation. Additionally, reference [30] discusses the notion of *quantifiable provenance* where a trust value can be computed based on a tuple's lineage, and proposes optimization techniques such as sampling to compute and maintain distributed provenance efficiently. We adopted the use of *composite location specifiers* from the MOSAIC [21] declarative network composition platform.

Our work is related to a large literature on access control in distributed systems (e.g. [4], [5], [8], [6], [7], [9]). In addition, through our use of declarative networking techniques, our work is related to a large literature of network specification languages. Our system extends and unifies these bodies of work to enable a unified declarative platform for developing secure networked information systems.

While access control in database systems is a well-explored topic, to our best knowledge, the usage has been limited to traditional database management systems. In recent months, security and access control policies are becoming increasingly integrated with stream processing systems (e.g. [31], [32]), as distributed stream processing systems become pervasively deployed on potentially untrusted networked environments (e.g. RFID, sensors, wireless mobile devices). While our paper focuses on securing declarative networks, as future work, we would like to further explore applying our framework to other distributed stream processing environments.

IX. Conclusion and Future Directions

In this paper, we present a unified declarative platform for specifying, implementing, analyzing and auditing large-scale secure information systems. Our three main contributions are as follows: (1) a unified declarative language, (2) authenticated query processing techniques for distributed settings, and (3) support for network provenance within a declarative framework. We validated our contributions via an extensive prototype evaluation on a LAN and the PlanetLab testbed.

Our proposed *SeNDlog* language draws its inspirations from declarative languages for access control and declarative networking, and lays the groundwork for richer explorations into the language design space. We have focused on the Binder access control language due to its simplicity and similarities with the *NDlog* declarative networking language. Despite its simplicity, *SeNDlog* supports a variety of practical uses cases in secure networking and distributed query processing.

We briefly survey some possible language extensions for future explorations. While we have focused on authentication, one can introduce additional security constructs for *secrecy* [33] and *encrypted facts* [26], hence ensuring that only authorized principals can interpret facts in distributed settings. Given our use of roles among principals, one can incorporate the notion of *restricted delegation* [5] and *speaks-for* [19] to the language. Another language feature involves security protocols that utilize distributed vote-based agreements. This has been formalized in various trust management languages such as DL [5], where a fact in the rule head is derived only when *k-out-of-n principals* in a rule body predicate derive a similar fact concurrently. *Privilege revocation* is another practical aspect of access control that we would like to support. Interestingly, time-based revocation can be enforced naturally using the declarative networking feature of *soft-state* [34] derivations where each derived tuple receives an associated lifetime, after which the tuple is deleted.

One of our overarching goals is to exploit the declarative framework for cross-layer optimizations in a unified manner. First, a query optimizer with knowledge of network state and security policies can exploit the performance/security tradeoffs, for example, by prioritizing the evaluation of facts with increased trust or security levels. Second, traditional database optimizations such as magic-sets [20] can potentially bridge the top-down evaluation approach used in access control, versus the typical bottom-up continuous evaluation of network protocols. This will enable us to utilize a query optimizer to adaptively choose between two different approaches: the "need-to-know" approach where sensitive information is on-demand given a specific access control request, and the "authorized-to-know" approach where a bottom-up evaluation engine computes and propagates facts limited by export predicates. Third, there is an inherent tradeoff between expressiveness and complexity in *SeNDlog*, and while each extension may be intriguing in isolation, one would not want to add them all indiscriminately. One can perhaps exploit this tension in a structured manner, as proposed in the Cassandra [9] system for trust management.

We are currently working towards bandwidth efficient techniques for querying and maintaining network provenance in distributed settings. Additionally, we plan to further explore the interplay between security and privacy of network provenance that we have highlighted in Section VI-A.

X. Acknowledgments

This work was partially supported by NSF grants CNS-0721845 and IIS-0812270.

References

[1] P. Laskowski and J. Chuang, "Network monitors and contracting systems: Competition and innovation," in *SIGCOMM*, 2007.

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

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

[4] T. Jim, "SD3: A Trust Management System With Certified Evaluation," in *IEEE Symposium on Security and Privacy*, May 2001.

[5] N. Li, B. N. Grosof, and J. Feigenbaum, "Delegation Logic: A logic-based approach to distributed authorization," *ACM TISSEC*, Feb. 2003.

[6] Moritz Y. Becker and Cedric Fournet and Andrew D. Gordon, "SecPAL: Design and Semantics of a Decentralized Authorization Language," Microsoft Research, Tech. Rep. MSR-TR-2006-120, 2006.

[7] J. DeTreville, "Binder: A logic-based security language," in *IEEE Symposium on Security and Privacy*, 2002.

[8] N. Li, W. H. Winsborough, and J. C. Mitchell, "Distributed credential chain discovery in trust management," *Journal of Computer Security*, vol. 11, no. 1, pp. 35–86, 2003.

[9] M. Y. Becker and P. Sewell, "Cassandra: Distributed Access Control Policies with Tunable Expressiveness," in *5th IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004.

[10] Boon Thau Loo et. al, "Declarative Routing: Extensible Routing with Declarative Queries," in *SIGCOMM*, 2005.

[11] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing Declarative Overlays," in *ACM SOSP*, 2005.

[12] Boon Thau Loo et. al., "Declarative Networking: Language, Execution and Optimization," in *ACM SIGMOD*, June 2006.

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

[14] R. Ramakrishnan and J. D. Ullman, "A Survey of Research on Deductive Database Systems," *Journal of Logic Programming*, vol. 23, no. 2, pp. 125–149, 1993.

[15] M. Abadi, "On Access Control, Data Integration and Their Languages," *Computer Systems: Theory, Technology and Applications, A Tribute to Roger Needham*, vol. Springer-Verlag, pp. 9–14, 2004.

[16] 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.

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

[18] PlanetLab, "Global testbed," http://www.planet-lab.org/.

[19] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in Distributed Systems: Theory and Practice," *ACM TOCS*, 1992.

[20] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," in *SIGMOD*, 1986.

[21] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith, "MOSAIC: Unified Declarative Platform for Dynamic Overlay Composition," in *ACM CONEXT 2008*.

[22] Secure BGP, http://www.ir.bbn.com/sbgp/.

[23] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach, "Secure Routing for Structured Peer-to-peer Overlay Networks," in *OSDI*, 2002.

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

[25] Daniel J. Abadi et. al., "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005.

[26] K. Minami and D. Kotz, "Secure context-sensitive authorization," in *PERCOM*, 2005.

[27] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *ACM Symposium on Principles of Database Systems*, 2007.

[28] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking Up Data in P2P Systems," *Communications of the ACM, Vol. 46, No. 2*, 2003.

[29] M. Abadi and B. T. Loo, "Towards a Language and System for Secure Networking," in *NetDB*, 2007.

[30] W. Zhou, E. Cronin, and B. T. Loo, "Provenance-aware Secure Networks," in *NetDB*, 2008.

[31] R. V. Nehme, E. A. Rundensteiner, and E. Bertino, "A security punctuation framework for enforcing access control on streaming data," in *ICDE*, 2008.

[32] Barbara Carminati and Elena Ferrari and Kian-Lee Tan, "Specifying Access Control Policies on Data Streams," in *DASFAA*, 2007.

[33] M. Abadi and B. Blanchet, "Analyzing security protocols with secrecy types and logic programs," in *POPL*, 2002.

[34] S. Raman and S. McCanne, "A model, analysis, and protocol framework for soft state-based communication," in *SIGCOMM*, 1999.