January 2008

Unified Platform for Secure Networked Information Systems

Wenchao Zhou
University of Pennsylvania

Yun Mao
University of Pennsylvania

Boon Thau Loo
University of Pennsylvania, boonloo@cis.upenn.edu

Martín Abadi
University of California Santa Cruz, Microsoft Research

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-05

Unified Platform for Secure Networked Information Systems

Abstract
In this paper, we present a unified declarative platform for specifying, implementing, analyzing and auditing large-scale secure information systems. Our proposed system builds upon techniques from logic-based trust management systems, declarative networking, and data analysis via provenance. 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 (NDlog), 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 the notion of authenticated communication among untrusted nodes. Third, we demonstrate that an integrated declarative framework enables cross-layer analysis and auditing via the use of distributed network provenance. Finally, using a local cluster and the PlanetLab testbed, we perform a detailed performance study of a variety of declarative secure networked information systems implemented using our platform. We further perform an evaluation of network provenance via a SeNDlog-based packet tracing service within a local cluster.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-05

Unified 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

In this paper, we present a unified declarative platform for specifying, implementing, analyzing and auditing large-scale secure information systems. Our proposed system builds upon techniques from logic-based trust management systems, declarative networking, and data analysis via provenance. 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 (*NDlog*), 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 the notion of authenticated communication among untrusted nodes. Third, we demonstrate that an integrated declarative framework enables cross-layer analysis and auditing via the use of distributed network provenance. Finally, using a local cluster and the PlanetLab testbed, we perform a detailed performance study of a variety of declarative secure networked information systems implemented using our platform. We further perform an evaluation of network provenance via a *SeNDlog*-based packet tracing service within a local cluster.

1. 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 [6, 19], efficient packet tracing [30] and flow analysis [33], all 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* [32, 15, 20, 26, 11], (2) *declarative networking* [23, 22, 21], and (3) database techniques for analyzing data computations via the concept of *provenance* (or *lineage* [8]). 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 system. Over the years, logical ideas and tools have been used to explain and improve access control. Several logic-based languages such as Binder [11], SD3 [15], D1LP [20] and SecPAL [26] 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 to express recursive queries [29] 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 [1] that access control languages such as Binder share similarities to data integration languages such as that used in the Tsimmis [9] data integration system, further indicating that ideas and methods from

VLDB '08, August 24-30, 2008, Auckland, New Zealand

the database community are directly applicable. Specifically, the contributions of this paper are as follows:

Unified declarative language for secure networked information systems: We propose the Secure Network Datalog (*SeNDlog*) language that unifies logic-based access control and declarative networking languages. Our language builds upon our previous proposal [3] by incorporating runtime types for *location specifiers* which references data based on both IP addresses and logical overlay addresses. It provides support for dynamically layering multiple overlays at runtime, and enables security policies to be enforced across various layers. We further demonstrate that *SeNDlog* can be used to declaratively specify a variety of networked systems, including authenticated path-vector routing protocol, secure Chord distributed hash table [25], and a secure implementation of the PIER [14] distributed query processor, several of which can be layered on top of each other.

Authenticated distributed query processing: We extend existing distributed recursive query processing [21] techniques to execute *SeNDlog* queries, by incorporating the notion of *authenticated communication*. Our proposed *authenticated pipelined semi-naïve* (APSN) evaluation ensures that all communication among untrusted nodes is authenticated during query execution, and recursive queries are evaluated in a pipelined fashion suitable for asynchronous distributed settings. The generated distributed dataflows further supports runtime address types which can be used to support *layered authentication* at various layers. When implemented, these dataflows implement the distributed information system together with their security policies. Beyond our platform, these techniques are applicable to other distributed stream processing systems [10] that require access control and enforcement of security policies.

Diagnostics and forensics with network provenance: The dataflow framework used in declarative networks captures information flow as distributed streams computations. Hence, it is natural to utilize *data provenance* [8] to "explain" the existence of any network state. We classify data provenance into various types (e.g. local vs distributed, online vs offline, authenticated, etc.), and demonstrate that they map into various use cases for secure networks including real-time diagnostics, forensics, accountability, and trust management.

Implementation and PlanetLab experimentation: We have developed a secure declarative networking engine that supports unified platform via the *SeNDlog* language, authenticated distributed query processing capabilities, as well as basic support for constructing and querying network provenance. Using a local cluster as well as PlanetLab [27], we demonstrate the viability of our system via a detailed performance study on a variety of secure networked information systems. We further perform an evaluation of network provenance via a *SeNDlog*-based packet tracing service within a local cluster.

The paper is organized as follows. In Section 2, we first present a background overview of trust management and declarative networking languages, focusing on the Binder [11] and *NDlog* [21]. In Section 3, we present the unified *SeNDlog* language, and present several examples in Section 4 that illustrate usage of the language. In Section 5, we demonstrate how *SeNDlog* queries are compiled into authenticated dataflows to implement a variety of secure information systems. Section 6 outlines analysis opportunities that are enabled with the integration of network provenance into our system. We present evaluation results in Section 7.

2. BACKGROUND

As background, we briefly introduce *Binder*, a representative logic-based access control language, and *NDlog*, the query language for declarative networking. Binder and *NDlog* are query languages based on Datalog [29]. 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.

2.1 Binder: Access Control Language

Binder is a logic-based language for expressing access control policies. We select Binder as our representative language since it has a simple language design, and is most similar to *NDlog*. Basically, a Binder program is a set of Datalog-style logical rules. Unlike Datalog, 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 may-access(P,O,read) :- good(P).
b2 may-access(P,O,read) :-
          bob says may-access(P,O,read).
```

The `says` operator implements one of the common logical constructs in authentication [18], 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".

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 and message generation 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.

The `says` operator abstracts from the details of authentication. 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 into another context and checked using the corresponding public key.

2.2 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 [23], and the Chord DHT in 47 lines of code [22]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations. We illustrate *NDlog* using a simple example of two rules that computes all pairs of reachable nodes:

```
r1 reachable(@S,D) :- link(@S,D).
r2 reachable(@S,D) :- link(@S,Z), reachable(@Z,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 simple example, 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 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. Tuples can also be maintained as *soft-state*, where tables are first declared with lifetimes that indicate the duration each tuple is maintained in the system (see [22])).

2.3 Comparing Binder and NDlog

Having introduced Binder and *NDlog*, we now elaborate on the differences between these two languages, to set the stage and motivate the unified features of *SeNDlog* in Section 3.

Trusted vs untrusted networks: One of the important requirements of both Binder and *NDlog* is the ability to support rules that express distributed computations, where nodes can communicate with each other. Hence, *NDlog* supports the notion of location that is similar to Binder's notion of context. On the other hand, *NDlog* is designed for a fully trusted environment, where the location relates to data placement. Each *NDlog* rule takes as input predicates with different location specifiers, and generate tuples are blindly accepted by other nodes. On the other hand, Binder assumes an untrusted network, where rules are executed with their own context, and communication happens via the use of "`says`"; unlike in *NDlog*, reliable authentication is required.

Export of derived tuples: In Binder, there is no integration of the security policy with the policy for exporting data. To illustrate, we consider the rule `b2` from the above example. The principal `alice` that runs these rules may wish only to export `may-access(P,O,read)` to the principal `P`, and not all nodes. It is not possible to express this restriction in Binder. Hence, any principal can import the `may-access(P,O,read)` tuple derived by `alice`. Being able to restrict the sending of messages to selected recipients is an important requirement in secure network protocols, both from performance and secrecy standpoints. *NDlog* achieves that with the use of location specifiers at the rule head.

Bottom-up vs top-down evaluation: Most access control languages including a practical implementation of Binder and SD3 utilize a goal-oriented top-down evaluation (backward-chaining from head to body) strategy. Specific requests are made as goals, which are then resolved against the security policies. On the other hand, network protocols are long-running processes, and incrementally recompute and repair routes based on changes to the underlying network. Hence, *NDlog* programs are executed in a bottom-up (or forward-chaining) [28] evaluation where the bodies of the rules are evaluated to derive the heads. This has the advantage of permitting set-oriented optimizations while avoiding infinite recursive loops, and at the same time, is a better fit for the incremental continuous execution model of network protocols. In addition, there are well-known database optimizations such as magic-sets [4] where can achieve the benefits of bottom-up evaluation, while avoid computing redundant facts in a goal-oriented fashion.

3. SECURE NETWORK DATALOG

Next we present the *SeNDlog* language based on the unification of Binder and *NDlog*. The goals of *SeNDlog* are as follows. First, the language should be easy and intuitive to use, but still as expressive as Binder and *NDlog* to support security policies and network protocols that have been previously supported by both languages. Second, the language constructs of *SeNDlog* should support authenticated communication and also enable the differentiation of nodes according to their roles. Third, *SeNDlog* should support both trusted and untrusted environments. Fourth, to leverage existing execution engines and fit the incremental continuous execution model of network protocols, *SeNDlog* must be amenable to efficient execution and optimizations by a distributed query engine using a bottom-up evaluation strategy.

3.1 Rules within a Context

In the *SeNDlog* language, 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.
```

In the example above, the 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 world, `N` can simply be the address of a node. For a typical network, IP addresses can be assumed. In order to support layered authentication, we additionally support *dynamic location specifiers* [24]: `N` can reference a logical address denoted by `[oID]::nID`, where `oID` is an optional overlay identifier, and `nID` is an mandatory overlay node identifier. In a multi-user environment where multiple users may reside on the same physical node, `N` can further include the user name. We can attach additional conditions `c1,c2,...,cn` that are used to determine at runtime whether a node serves a certain role:

```
At N, c1,c2,...,cn
  r1 p :- p1,p2,...,pn.
  r2 p1 :- p2,p3,...,pn.
```

In the above example, a principal `N` can execute the rules `r1` and `r2` only if all the conditions `c1,c2,...,cn` are satisfied at runtime. This allows the role of a principal to be defined based on runtime conditions.

3.2 Communicating Contexts

Similar to Binder, the *SeNDlog* language allow different principals or contexts to communicate via import and export of tuples. The movement of tuples 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 `N` is the principal that is asserting 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 a specified principals. Similar to principals, our location specifiers need not be IP addresses. They can flexibly refer to logical overlay addresses whose types are determined at runtime. This feature is necessary for dynamically layering multiple secure overlays at runtime, e.g. a distributed query processor over a secure DHT. Detailed description of dynamic overlay composition is outside the scope of this paper (see [24] for more details).

With the definitions above, 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 four *SeNDlog* rules `e1-e4`:

```
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 the honesty 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).

3.3 Language Extensions

We aim to keep the *SeNDlog* language as simple as possible. Nevertheless, in this section we briefly discuss some possible extensions. 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 3.2, `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.

There are several other security constructs such as *secrecy* [2], *delegation* [20], *speaks-for* [18], and *privilege revocation* that we would like to explore in future. Another language extension includes support for security protocols that utilize distributed vote-based agreements. This has been formalized in various trust management languages [20], 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.

4. PRACTICAL SENDLOG EXAMPLES

In this section, we provide three example specifications using *SeNDlog* to demonstrate its language features. Our examples build upon previous examples, starting from a secure routing protocol, to a secure DHT overlay, followed by secure p2p query processing layered over a secure DHT. We present evaluation results of these examples in Section 7. In addition to these examples, we have also explored using *SeNDlog* to implement a secure version of DNS as presented in SD3 [15], and enforcing access control policies in p2p file-sharing networks and shared testbeds [27].

4.1 Authenticated Path-Vector Protocol

```
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).
```

Our first example shows the basic path vector protocol as presented in the declarative routing paper [23], with the additional use of the "`says`" operator. At every node `Z`, this program takes as input `neighbor(Z,X)` tuples that contain all neighbors `X` for `Z`. 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`.

At every node `Z` that runs the above program, the `Z says advertise(Y,P)` tuples containing the path to destination node `Y` is communicated among neighboring nodes. As noted in Section 3.2, we omit "`Z says`" for brevity in rule `z3`. The use of "`says`" ensures that all `advertise` tuples are verified by the recipient for authenticity. The eventual outcome of executing the program is the generation of `route(Z,X,P)` tuples, each of which stores the path `P` from source `Z` to destination `X`.

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 count. 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 selected neighbors X based on the policies of the local carryTraffic table. Each exported advertise tuple has a new path P1 which is computed by prepending neighbor X to the input path P using the f_concat function[1] 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 count.

With a basic authenticated routing protocol in place, we can add additional rules to implement large-scale information systems that require a secure routing layer. The secure version of the BGP [31] inter-domain routing protocol is an example of a large-scale information system where autonomous AS gateways exchange route advertisements. Other information systems include p2p file sharing applications that require authenticated routing among neighbors, or secure content distribution networks.

4.2 Secure Distributed Hash Tables

In our second example, we use *SeNDlog* to specify the assignment of node identifiers in secure DHT overlays. We base our example on a declarative version of Chord originally implemented in 47 rules [22]. Our version of the code avoids a security weakness in a DHT where malicious nodes can occupy a large range of the key space. We focus on showing *SeNDlog* versions of the relevant rules from the original Chord specification.

This example also demonstrates the use of *SeNDlog* to specify the different roles for nodes. There are three sets of rules for three types of nodes: (1) a new node NI joining the chord ring, (2) the certificate authority CA, and (3) the landmark node 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 ring 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 IP 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 stored as a nodeID(NI,N) tuple. It also forwards the nodeIDCert to its landmark node LI in order to join the chord ring. 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. The rules as presented [22] can then be used by node NI to implement the rest of the Chord protocol.

In the above example, CA generates a certified node identifier for a node NI to join the network. To take this one step further, one can express the *SeNDlog* rules that result in authenticated *all* communication among all Chord nodes. In this case, each message exchanged between nodes are communicated via says, and digitally signed by each node.

[1]In both *NDlog* and *SeNDlog*, function calls are additionally prepended by f_.

4.3 Secure DHT-based Join Processing

In our third example, we present a distributed database join operation that can be performed securely over a DHT. Joins over DHTs are widely studied and used in p2p query processors such as PIER [14] for the purposes of p2p keyword search and performing network event correlations. In an untrusted environment, authenticated communication is essential for ensuring that malicious nodes do not inject tuples that will corrupt the final answer. Our example demonstrates that additional query processing capabilities at the application level can be written elegantly as *SeNDlog* rules, often with the use of logical location specifiers to address network state at the lower levels. Moreover, authentication can be performed at various layers in the network. The following rules implement a distributed join operation in the PIER query processor:

```
At alice,
 a1 storeA(X,Y)@NI :- tableA(X,Y), K=f_sha(X),
          NI=Chord::K.
At bob,
 b1 storeB(X,Y)@NI :- tableB(X,Y), K=f_sha(X),
          NI=Chord::K.
At NI,
 r1 results(X,Y)@r :- alice says storeA(X,Y),
          bob says storeB(Y,Z).
```

Our example consists of two tables tableA and tableB, owned by principals alice and bob respectively. Rules a1 and b1 are executed in the context of alice and bob respectively. For the ease of exposition, our rules indicate the two principals as alice and bob constants. In practice, the principle may additionally include the address of the principals, as described in Section 3.1.

Ownership-based publishing: Rules a1 and b1 result in indexing all of tableA and tableB in the DHT, based on the hash of the first attribute. To do that, both alice and bob compute the SHA-1 hash of the first attribute X to generate a Chord identifier K, denoted as the logical address Chord::K in the rules. The resulting storeA and storeB tuples are then sent to the node NI responsible for storing tuples for the identifier Chord::K. In an untrusted environment, these published tuples also include the signatures of alice and bob, which are verified before being stored at NI.

Layered authentication: This example demonstrates the possibility of *layered authentication*. The execution of rules a1 and b1 would result in a Chord lookup to locate NI, and the communication for the underlying Chord protocol can be authenticated as described in Section 4.2. Layered authentication ensures that alice and bob can authenticate in an end-to-end fashion at the query processor layer, in addition to any authenticated communication at the underlying Chord overlay.

Distributed joins: Rule r1 is a query issued by node r to perform a distributed join of storeA and storeB tuples. This rule is disseminate as a query, and upon reaching all nodes NI, a distributed DHT-based join [14] is performed as follows. The rules for rehashing these tuples can themselves be automatically rewritten by a PIER query processor implemented in *SeNDlog* as follows:

```
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,Y)@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 join attribute `Y`. Rule `r1a` rehashes all `storeA` tuples while retaining the original signature of `alice`, adhering to the *honesty constraint* described in Section 3.2. The join is performed at node `RI` with local `storeB` tuples, and the resulting `result` tuples are sent back to node `r`.

5. SECURE QUERY PROCESSING

In this section, we describe query processing techniques for executing *SeNDlog* programs. Our proposed *Authenticated Pipelined Semi-naïve* (APSN) is an adaptation of the *Pipelined Semi-naïve* (PSN) [21] proposed for processing distributed recursive queries in declarative networks.We first provide a brief background on PSN followed by APSN. We will additional describe opportunities for future extensions.

5.1 Background on PSN

Unlike traditional semi-naïve evaluation, PSN relaxes the requirement to perform computations in synchronous rounds (or iterations), a prohibitively expensive operation in a distributed setting. We describe the simplest version of PSN, where tuples are processed tuple-at-a-time in a pipelined fashion, and duplicate evaluations are avoided using local arrival timestamps. We consider the following recursive Datalog rule p :- $d_1, d_2, ..., d_n, b_1, b_2, ..., b_m$.

In the rule body, there are n derived predicates ($p_1, ..., p_n$), and m *base predicates* ($b_1, ..., b_m$). Derived predicates refer to intensional relations that are derived during rule execution, and may be mutually recursive with p. 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: $\triangle p$:- $d_1, .., d_{k-1}, \triangle d_k, d_{k+1}, .., d_n, b_1, b_2, ..., b_m$.

$\triangle d_k$ denotes a tuple $t_k \in d_k$ that is used as input to the rule for computing new d tuples. In PSN, each node maintains a FIFO queue (ordered by arrival timestamp) of new input tuple. 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.

5.2 Authenticated PSN

In APSN, import and export predicates (Section 3.2) denoted by the "`says`" keyword require additional operators for authentication. Consider the following *SeNDlog* rule: $p\ says\ a$:- $d_1, ..., d_n, b_1, ..., b_n, p_1\ says\ a_1, p_2\ says\ a_2 ..., p_o\ says\ a_o$. There are 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, a *authenticated delta rule* is generated as follows:

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

In the rest of the section, we will show how delta rules are compiled into execution plans.

5.2.1 Basic Dataflow Generation

Figure 1 shows an exampled dataflow that is automatically generated from the *SeNDlog* rules by our query processor. The dataflow

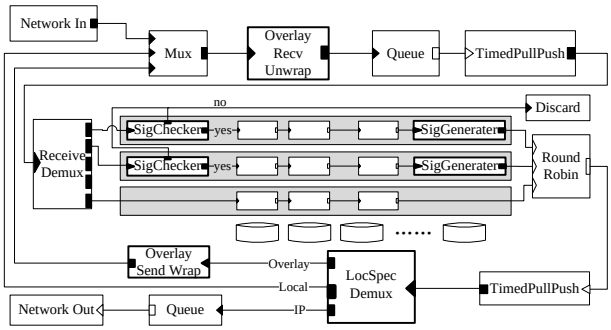

Figure 1: Dataflow execution plan for a single node.

execution model is based on that of the P2 declarative networking system [22]. In P2, queries are compiled and executed as *distributed dataflows* and share a similar execution model with the Click modular router [17]. 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 `Mux`, `Queue`, `TimedPullPush`, and `Demux` supports 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 PSN and APSN delta rules into a series of relational operators such as joins, aggregations, selections, and projections. Messages flowing among dataflows 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 P2 tuple has an associated lifetime declared at creation time of each table[2].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 respectively. Given an input tuple, the output of executing a strand would either be local state modifications (insertions/deletions to local tables), or generation of new messages which are then transmitted via the `Network Out` operators.

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. All outgoing tuples `t` that require authentication are communicated as `p,s,t` triplets, where `p` corresponds to the source principle, `s` is the signature generated by encrypting a message digest (essentially a cryptographic hash of `t`) with `p`'s private key.
- At the receiving node, if authentication is required, an additional `SigChecker` operator checks all incoming tuples `p,s,t`, by decrypting `s` with the public key of `p`, and verifying that the decrypted contents matches the cryptographic hash of `t`.

[2]A zero lifetime tuple is treated as an event stream 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 [10].

The SigGenerator and SigChecker uses a public-key cryptography scheme that can be computationally expensive. The SigChecker operator requires knowledge of the public key of the principle whom the dataflow is importing tuples from. Note that Key management is an orthogonal problem to our work. In the simplest model, each node have full apriori knowledge of all the public keys of all other communicating peers. This is evidently not practical. A more feasible approach that we have adopted (e.g.in Section 4.2) 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. Other lightweight key management schemes are possible, and integrating them into our execution plans is an avenue for future work.

5.2.2 Example APSN Strand

Figure 2 provides a closer look at individual operators that comprise each rule strand shown in Figure 1. Our example is based on rule z2 used in the authenticated path vector protocol presented in Section 4.1. The APSN delta rules are as follows:

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

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

The first rule strand z2a (shown as strand z2a@Z in Figure 2) 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, and if this is a tuple not seen previously,joined with matching tuples from the local acceptRoute table to generate new route tuples which are further inserted into the local route table. Rule 2b (shown as strand z2b@Z) is triggered by changes to the local acceptRoute table, and hence requires no authentication. 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. On the other hand, the rule strand for rule z3 from Section 4.1 would require a SigGenerator operator to generate the signature using the private key of Z before being sent.

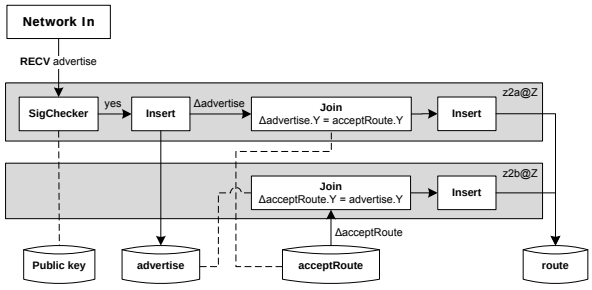

Figure 2: Generated Rule Strands for rule z2.

5.3 Layering Support and Security Extensions

SeNDlog support *layered authentication*, as shown by the secure DHT-based joins in Section 4.3, where secure communication can happen at different layers (e.g. layering a secure p2p query processor on top of a secure Chord DHT). This requires dataflow support for layering [24] and dynamic location specifiers.

The dataflow includes three additional operators shown in **bold** in Figure 1: OverlayRecvUnwrap, OverlaySendWrap, and LocSpecDemux. The operators OverlayRecvUnwrap and OverlaySendWrap are used for de-encapsulation and encapsulation of tuples from an underlying network specified in *SeNDlog*.

At the top of Figure 1, the Mux multiplexes incoming tuples received locally or from the network. These tuples are processed by the OverlayRecvUnwrap operator that will extract the overlay payload for all tuples of the form overlay.recv(Packet), where Packet is the payload with type tuple. Since the payload may be encapsulated by multiple headers (for layered overlays), this operator needs to "unwrap" until the payload is retrieved. The Packet payload is then used as input to the dataflow via the ReceiveDemux operator, and used as input to various rule strands for execution. Executing the rule strands results in the generation of output tuples that are sent to a LocSpecDemux operator. This operator checks the runtime type of the location specifier, and then demultiplexes accordingly. Tuples designated for the local node are directed back to the same dataflow, and IP-based tuples are sent over the network. To support layering, outgoing tuples designated for a local overlay node are de-encapsulated, and reinserted back to the same dataflow.

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. For example, other computationally less expensive schemes such as the use of cryptographic *message authentication code* (MAC) are also possible, if nodes apriori agree upon a shared secret key. Our declarative framework and runtime system does not preclude the use of these alternative schemes: they will correspond to different levels of "says" and the query planner will automatically generate the appropriate SigGenerator and Checker operators to implement the authentication mechanism. In addition, while we focus our discussion to supporting the "says" operator described in Section 3. Other security mechanisms (e.g. public-key encryption for confidentiality) can be similarly supported by additional operators that encrypt/decrypt incoming and outgoing tuples.

6. DIAGNOSTICS AND FORENSICS WITH NETWORK PROVENANCE

The dataflow framework used in our platform captures information flow naturally as distributed streams computations. Hence, it is natural to utilize the database notion of *data provenance* to "explain" the existence of any network state. We summarize a variety of diagnostics and analysis opportunities [34] that arise from the use of our platform. The main goal of this section is to illustrate the potential for doing analysis and auditing across levels given a unifying declarative framework for network routing, information systems, and security policies.

6.1 Use Cases

We focus on use cases ranging from *real-time diagnostics*, *forensics*, and *trust management*, and relate them to different types of network provenance enabled by our platform.

Real-time Diagnostics: Provenance is useful for real-time diagnostics and debugging of distributed systems, in order to detect run-time anomalies resulted from divergent protocols and network intrusion. For example, a continuous query specified in *SeNDlog* can be used to compute the number of changes to a routing table entry over past T seconds, and generate an alarm event when the number of changes exceed a threshold as an indication of possible divergence. Upon receiving the alarm, the system may generate a distributed recursive query over the network provenance to detect-

ing the source of malicious activities.

Forensics: In addition to real-time data, historical data is often required to correlate traffic patterns of attackers. A common area of research has been in providing IP "traceback"[30] of traffic, either by the receiver or by an involved third party, to determine where packets are originated from without trusting the unauthenticated IP headers. One can store annotations either in the packet (i.e. piggyback each tuple with its complete "path" or "provenance"), or maintain state at each router, to allow for subsequent traceback via a distributed query during forensic analysis. To reduce the storage and communication overhead, summarization (via bloom filters) and sampling techniques respectively to compress the provenance.

Trust Management: In our final use case, network provenance is useful for enforcing distributed trust management [7] policies in networked information systems. Using an example from Internet routing, the path-vector protocol used in BGP carries the entire path during route advertisement, in order to allow for ASes to enforce their respective policies. More generally, provenance in our system enables any networked information node to trace the origins of its data via a "*chain of custody*", and hence enforce trust policies to accept or reject incoming updates based on the source origins.

6.2 Network Provenance

Given the above use cases, we present a partial list of possible types of network provenance [34] that can be incorporated into our system. We make use of an example network which consists of three nodes `a`, `b`, `c` and three unidirectional links `link(a,b)`, `link(a,c)`, `link(b,c)`. Figure 3 shows the derivation tree for `reachable(@a,c)` that results from executing the *NDlog* program in Section 2.2. This derivation tree essentially represents the provenance of the tuple, where the initial input base tuples are at the leaves of the tree. The ovals in the diagram represent the rules (`r1`, `r2`, or `union` to combine their results) that are used for the derivation of `reachable(@a,c)`. In order to incorporate provenance into *SeNDlog* executions, we annotate each derivation with its location (denoted by the location specifier.

Local vs Distributed Provenance The derivation tree shown in Figure 3 can be stored either locally or in a distributed fashion. In *local provenance*, the tree is stored at node `a`, which is the final storage location of `reachable(@a,b)`. In order to have a locally complete provenance, each tuple that is derived needs to piggy-back its entire provenance when shipped from one node to another. On the other hand, one can utilize *distributed provenance*, which only stores pointers to the previous node to reconstruct its provenance on demand. Hence, node `a` only needs to maintain the fact that it is derived from its local `link(@a,b)` and node `b`'s `reachable(@b,c)`. The analogy here is IP traceback, where one can either store the entire traversed path within each packet (similar to local provenance), or only maintain enough state at each router to traceback the route on demand (distributed provenance). There are evidently tradeoffs between local and distributed provenance. In local provenance, computation is more expensive for each tuple, but queries can be computed locally. Also, trust policies can be enforced without having to contact other nodes. On the other hand, distributed provenance requires no extra communication overhead, but incurs expensive cost of querying the provenance.

Online vs Offline Provenance. 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. The purpose of online provenance is to detect network anomalies at runtime. For example, when a node is detected to be suspicious, one can query the online provenance to delete all routing entries associated with the malicious node. However, online provenance by itself has limited usage given that most networked data are maintained as soft-state with limited lifetimes. In this case, we can additionally maintain offline provenance for data that has long expired. Offline provenance is useful also for real-time diagnostics, and can additionally be used to support forensics and enforce accountability.

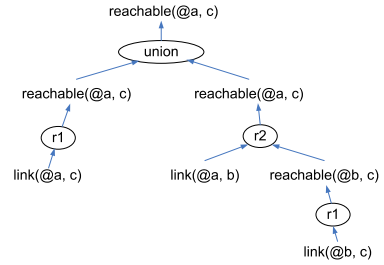

Figure 3: ***NDlog*** **provenance for `reachable(@a,c)`.**

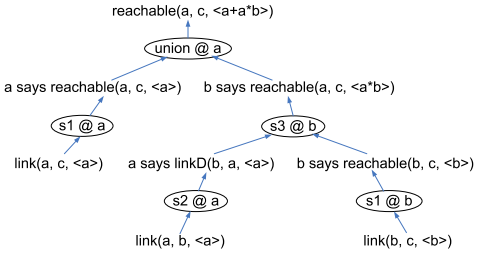

Figure 4: ***SeNDlog*** **provenance for `reachable(a,c)` in `a`'s context.**

Authenticated Provenance: In an untrusted network, authentication may be required to ensure the validity of provenance computed by other nodes (e.g. to prevent spoofing of messages from malicious attackers). Figure 4 shows an alternative derivation tree based on a *SeNDlog* version of the same reachable query. We note the following differences. First, since all rule bodies are localized within the context of a security principal, we can omit the location specifiers for each tuple. However, we annotate each operator (denoted by the oval) with the location (or context) where the rule is executed. Second, each node in the tree is asserted by a principal using "says". In an untrusted environment, this means that individual nodes in the provenance tree need to validate the authenticity of the computed provenance using digital signatures. As notation, each tuple has an additional field denoted by `<...>` that stores the principals used for deriving the tuple are encoded as algebraic expressions [12]. $+$ represents union, and $\star$ represents a join. Based on this algebraic expression, each principal can determine the principals involved in deriving the fact, and hence determine whether to accept this fact based on existing trust relationships.

To illustrate different types of network provenance, we provide an example based on network packet traceback. The following *SeNDlog* rule forwards packets in the network based on the local routing tables:

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

The `forwarding` table is stored at every router, and is computed by adding additional rules to the path vector protocol in Sec-

tion 4.1 to compute the nextHop router along the best path to any given destination (Dest). Given an incoming packet unique by its Pid to Dest with payload Data, rule f1 retrieves from the forwarding table the nextHop router along the best path to Dest. 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 (stored as the derivation of packet) is forwarded with each packet tuple. In *distributed provenance*, state is maintained at every router to traceback the reverse path of each packet uniquely identified by its pid back to the source. If routers are untrusted, authentication can be achieved by adding a signature generated from each packet including any additional provenance information, hence resulting in the use of *authenticated provenance*. 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.

7. EVALUATION

In this section, we present the evaluation of our platform on a local cluster and on PlanetLab. The goals of our evaluation are as follows. First, we validate that *SeNDlog* specifications achieve the expected functionality of a variety of secure networked information systems. Second, we experimentally quantify the performance overheads of using authenticated communication, both within a local cluster and on a wide-area testbed. Third, we perform a preliminary proof-of-concept evaluation of network provenance support using our platform via a *SeNDlog*-based packet tracing service within a local cluster. Our evaluation is based on the P2 declarative networking system [22], with enhancements to execute *SeNDlog* rules as described in Section 5. To generate signatures used in APSN, we utilize the OpenSSL v0.9.8b cryptographic libraries that generate 640-bit signatures given input data. We further added runtime support to the P2 system to compute online, local and authenticated provenance (Section 6) during rule execution.

7.1 Experimental Setup

Evaluation platform: Our first set of experiments are conducted by executing our 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 sufficient bandwidth. Our second set of experiments are conducted on the PlanetLab [27] 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.

Query workload: Our query workload consists of three *SeNDlog* example queries presented in Section 4. The *Best-Path* distributed query takes as input rules z1-z3 from Section 4.1 to generate route tuples, with an additional cost attribute that is the sum of all links along a path. Two additional local rules are used to compute the minimum path cost given the local route table, and select for each source/destination pair, the path with minimum cost. The *Chord* query is a declarative version of the Chord DHT [22] in *SeNDlog* that supports authenticated communication as described in Section 4.2, and continuously maintains the Chord routing tables as new nodes enters and leaves the Chord network. The *PIER query* consists of the rules in Section 4.3 that implements a declarative version of the PIER [14] distributed query processor that supports distributed indexing and joins of relational tables over the DHT.

Performance metrics: Given the above query workload, our performance metrics are as follows:

- **Query completion time:** Time taken for a query to finish execution. As our example queries are recursive, this means the time elapsed before the system reaches a distributed fixpoint, where all nodes finish their computations on queries.
- **Average Bandwidth:** The per-node average bandwidth utilization for executing a distributed *SeNDlog* protocol. This metric is used for the *Chord* query which is continuously executing and maintaining its routing tables.
- **Aggregate Communication Overhead:** The total amount of traffic generated when executing a single distributed query. The *Best-Path* and *PIER* queries fall into this category.

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. To explore extremes in performance due to the use of authenticated query processing, we explore two extremes in all our experiments: *Auth-None* communicates all tuples without authentication, while *Auth* represents the other extreme where all communication that occurs during query evaluation requires a RSA signature to be generated from the *entire* tuple and verified by the recipient node.

7.2 LAN Experiments

In this section, we present the results of executing the *Best-Path*, *Chord* and *PIER* queries on our local cluster, based on the performance metrics described above.

7.2.1 Best-Path Query

Our first experiment consists of executing the *Best-Path* distributed recursive query to a fixpoint on all 16 cluster nodes. We vary the network size from 16 to 128, where at its largest, each cluster machine runs 8 P2 virtual nodes. As input to the *Best-Path* query, we initialized a neighbor table where each node has an outdegree of 3. To ensure a connected network, we first connect all nodes in a a ring, and then randomly add the remaining neighbor links to achieve the network outdegree.

Figures 5 and 6 show the aggregate communication overhead and query completion latency for the *Best-Path* query averaged over 8 experimental runs. We make the following observations. For the largest network size of 128, *Auth* incurs 30% additional aggregate communication and an additional 7 seconds is required for the query to complete execution as compared to *Auth-None*. Since our experiments are carried out in a LAN where machines are connected by high-speed Gigabit Ethernet, the additional time required for query completion can be attributed to the computational overhead of generating the signatures for each message tuple.

7.2.2 Chord DHT

Next, we study the performance characteristics of a *SeNDlog* implementation of the Chord protocol. The main difference compared to the previous *Best-Path* query is the continuous execution flavor as opposed to a single query 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

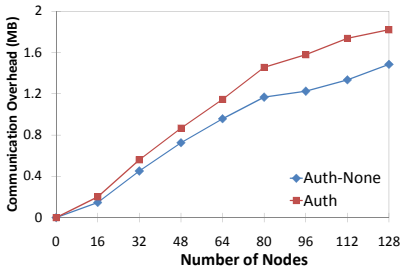

Figure 5: Per-Node aggregate communication overhead (MB) for the *Best-Path* query

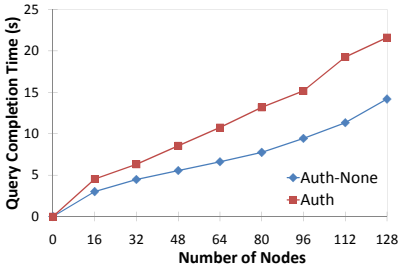

Figure 6: Query completion time (s) for the *Best-Path* query

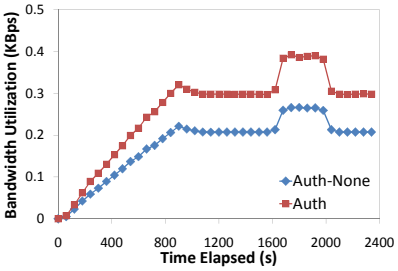

Figure 7: Per-Node average bandwidth utilization (KB/s) for *Chord*

lookup queries are issued simultaneously from 8 different nodes at three seconds interval for the next 400 seconds.

Figure 7 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 consumption stabilizes at 0.2KBps and 0.3KBps respectively for *Auth-None* and *Auth*. At 1600s, the bandwidth consumption increased due to the additional Chord lookups that we generated to measure lookup latencies. We make the following additional observations. First, *Auth* incurs 47% more bandwidth compared to *Auth-None*, which is larger as expected compared to *Best-Path*, given that Chord message tend to small (180 bytes on average). Despite incurring higher relative bandwidth usage, in a LAN environment with sufficient bandwidth, most of the additional performance penalty can still be attributable to CPU overhead of signature generation. Given that Chord messages are small, the actual processing penalty due to the signature generation is negligible, as reflected in Figure 8 where the latency CDF of *Auth-None* and *Auth* are almost equivalent.

7.2.3 DHT-based Join Query

Building upon our Chord experiment, we experiment with a *SeNDlog* implementation of a DHT-based PIER distributed join operation operation. This workload involves the continuous execution of a 128-node Chord network. At each node, we also execute a PIER query processing node that uses the Chord node for routing. Running PIER and Chord concurrently allows us to validate the ability of our platform to layer the PIER overlay over Chord, and experiment with *layered authentication* across two layers (PIER and Chord), as described in Section 5.

Once Chord has stabilized, we issued a bandwidth intensive PIER query which involves a distributed join over multiple DHT nodes. The actual *SeNDlog* query is shown in Section 4.3. First, tables `tableA` and `tableB`, each with N tuples varied from 100 to 800, are indexed over the Chord DHT based on the hash of their first attribute value using rules `a1` and `b1` respectively. Once these two tables have been published and indexed over the DHT, a distributed join operation is carried out. This join incurs communication overhead due to the rehashing of `tableA` and `tableB` tuples based on their join key, and the results of the join are sent back to the node initiating the query. To emulate large sizes of tuples and hence explore the performance limits of authentication using our system, we additionally padded each tuple with an additional 640 bytes.

Figure 9 shows the aggregate communication overhead incurred by the distributed join operation as the input table sizes increases from 100 to 800. We make the following observations. First, the increase is quadratic as expected, given that the join of two tables result in a quadratic increase of intermediate and final results (our join selectivity is set to 1/16). Second, *Auth* incurs only an additional 8% additional communication overhead comparing with *Non-Auth*, much lower compared to the earlier *Best-Path* query and Chord, due to larger sizes of tuples being shipped. To understand the performance impact of authentication on the query processing throughput of PIER, Figure 10 shows the progress of query execution (measured by percentage of total final results received) over time. For table sizes of 500 and 800, we observe that *Auth* increases the query completion time by only 1 and 6 seconds respectively.

7.3 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. Apart from using PlanetLab, our experimental setup was kept identical to that in our local cluster. As input to the PIER query, we utilize a distributed join of two tables with 100 tuples each.

Figure 11 shows the average bandwidth utilization for executing PIER distributed join query over Chord on PlanetLab. Compared to a similar LAN experiment in Figure 7, the bandwidth utilization of 0.2KBps and 0.3KBps for *Auth-None* and *Auth* are identical, validating that Chord is executing correctly on PlanetLab in terms of protocol messages. The PIER query is issued at time 2300s, after the Chord overlay has stabilized and the two tables `tableA` and `tableB` published and indexed by Chord.

Figures 12 and 13 shows the absolute performance in terms of Chord lookup latency and PIER query completion time on PlanetLab. In terms of absolute lookup latency, Chord lookups on PlanetLab are an order of magnitude slower compared to a LAN environment. We attribute the increase to real-world effects, where competing slices on PlanetLab contend for limited bandwidth and CPU resources. In addition, the performance penalty of authenticated communication is more apparent due to similar constraints. For example, the median Chord lookup latency increases by 1.5 seconds for *Auth* compared to *Auth-None*. We observe a similar effect for the PIER query on PlanetLab in Figure 13: while the time to receive 50% of result tuples are roughly equivalent for *Auth* and *Auth-None*, the eventual completion time for *Auth* is 40s larger. Overall, we observe that in a shared networked environment with limits on bandwidth and computational resources, authenticated query processing incurs observable performance penalty for bandwidth intensive distributed joins.

7.4 Network Provenance Experiment

In our final experiment, we perform a preliminary evaluation of a *SeNDlog*-based packet tracing service described in Section 6.2. This experiment validates and evaluates our runtime system support for computing and communicating *local* and *authenticated* prove-

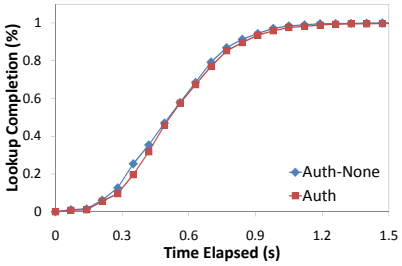

Figure 8: CDF of Chord lookup latency

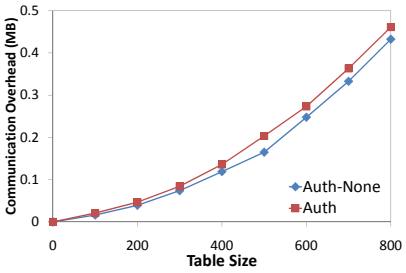

Figure 9: Per-Node aggregate communication overhead (KB) for the PIER Query

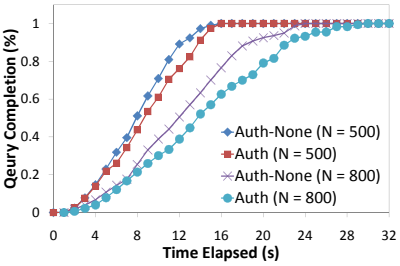

Figure 10: Query completion (percentage of final results) over time for the PIER Query

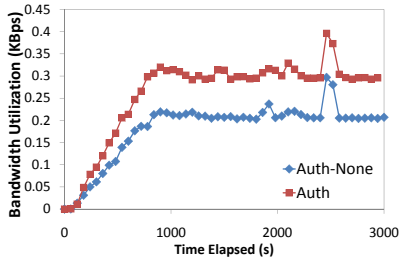

Figure 11: Per-Node Bandwidth Utilization (KB/s) for PlanetLab PIER query

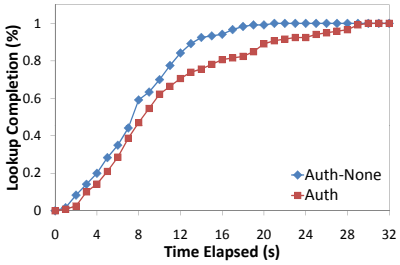

Figure 12: CDF of PlanetLab Chord lookup latency

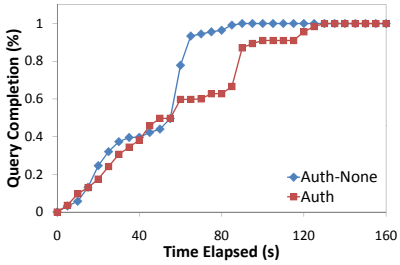

Figure 13: Query completion over time for PlanetLab PIER query

nance. In local provenance, each packet is annotated with its entire derivation, essentially all visited routers along its path from source to destination. This provenance can be additionally authenticated and signed with signatures with the use of authenticated provenance. Distributed provenance as described in Section 6.2 is currently not supported in our system.

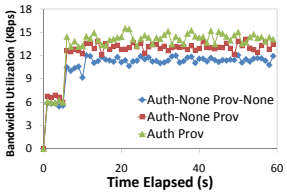

Figure 14: Per-node average bandwidth utilization (KBps) for packet tracing experiment

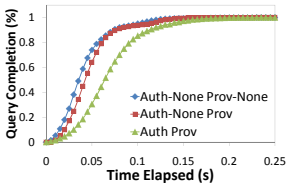

Figure 15: CDF of packet delivery latency for packet tracing experiment

The experiment consists of 128 P2 nodes executing on our 16-node cluster, and the *Best-Path* query is executed to compute the `forwarding` tables used to determine the next hop along the shortest path from a given source to the destination. After this query has converged, we issue a *SeNDlog* continuous query on all nodes that result in 800-byte packets to be injected and forwarded from random sources to destinations at a rate of 160 packets/second. We execute *SeNDlog* packet forwarding with three settings: *Auth-None Prov-None* communicates all packets without authentication and provenance; *Auth-None Prov* annotates local provenance to every packets; and *Auth Prov* additionally signs each packet storing its provenance for authentication purposes.

Figure 14 shows the bandwidth overhead incurred in delivering packets. We observe a 12% increase in bandwidth utilization by *Auth-None Prov* compared to *Auth-None Prov-None*. *Auth Prov* incurs an additional 10% overhead over *Auth-None Prov*. Given that the diameter of the network is small, the provenance consisting of all intermediate routers along the path from source to destination is relatively small comparing with the size of packet. Hence, the additional bandwidth utilization due to provenance is relatively low.

Figure 15 presents the CDF of packets delivery latency (the time elapsed between sending and receiving a packet). Compared with *Auth-None Prov-None*, *Auth-None Prov* incurs 10% increase in median latency due to the additional cost of computing and shipping the provenance information with each packet. On the other hand, *Auth Prov* incurs high overhead, resulting in 60% increase in median latency. Wince we are running our experiments in a local cluster, the absolute increase in median latency is negligible (less than 0.05 seconds). As follow up, we plan to better understand the performance implications of network provenance support in realistic wide-area networks, where optimizations such as sampling will be vital for efficient provenance processing.

7.5 Summary of Results

Revisiting the goals of our evaluation articulated earlier in this section, we validate that *SeNDlog* specifications for routing, Chord DHT, and PIER distributed join result in correct implementations. Our network provenance experiment on packet tracing with local and authenticated provenance also result in expected behavior. Within a local cluster, we note that the performance penalty of using authenticated query processing depends largely on the type of query being executed. For a continuous executing query that implements Chord with small message transfers, the overhead in terms of latency is negligible. For a bandwidth intensive query such as computing best paths or executing a distributed DHT-based join, the performance penalties due to authentication are more apparent, especially on PlanetLab where bandwidth and computation resources may be limited. These differences may be mitigated with alternative authentication schemes that achieve different performance/security

tradeoffs. Integrating these tradeoffs in an application-tunable fashion into a query optimizer is an interesting area for future work.

8. RELATED WORK

This paper extends our initial *SeNDlog* proposal [3] on secure networking language to a unified platform to build secure information systems at various layers, including the use of dynamic location specifiers that enables layered authentication. In addition, we have a full-fledged implementation, and experimentations in a LAN cluster and on PlanetLab.

Our work is related to a large literature on access control in distributed systems (e.g.[15], [20] and [26]). In addition, through our use of declarative networking techniques, our work is related to a large literature of network specification languages (e.g.d3log [16], metarouting [13]). Our system extends and unifies these bodies of work to enable an extensible platform for secure networked information systems that integrates declarative routing, overlays, and security policies.

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, we have observed new application scenarios where security and access control policies are integrated with distributed stream processing (e.g. [5]), as in-network stream processing become pervasively deployed on new and potentially untrusted networked environments (e.g. RFID, sensors, wireless mobile devices). While our paper focuses on secure declarative networking and applications supported within this framework, at its core, a declarative networking engine is essentially a distributed stream processor that incrementally maintains recursive stream views in a distributed setting. As future work, we would like to further explore applying our framework to other stream processing environments.

9. CONCLUSION

In this paper, we present a unified declarative platform for specifying, implementing, analyzing and auditing large-scale secure information systems. Our proposed *SeNDlog* language unifies logic-based access control languages with query languages for declarative networking. We extend existing distributed recursive query processing techniques to execute *SeNDlog* programs that incorporate the notion of authenticated communication among untrusted nodes. We conduct a detailed performance evaluation of various systems developed using our platform within a LAN environment as well as PlanetLab. We further perform an evaluation of network provenance via a *SeNDlog*-based packet tracing service within a local cluster.

Beyond language extensions to support additional security constructs described in Section 3.3, we believe that there are exciting cross-layer optimizations and analysis opportunities that can arise with a unified declarative framework, and we have barely scratch the surface with this paper. While preliminary, our initial exploration of network provenance lays the groundwork for future exploration in developing bandwidth efficient techniques for querying and maintaining network provenance, and grounding its practical usage for analyzing and securing networks. We would also like to further explore other practical aspects of our system, including query optimizations that exploit security/performance tradeoffs, scalable key management, extensible security type system, and also validation of our platform with a greater variety of networked information systems. To this end, we are working towards a publicly available deployment service on PlanetLab usable by the community to experiment with new secure networked systems.

10. REFERENCES

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

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

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

[4] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1986.

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

[6] A. Bender, N. Spring, D. Levin, and B. Bhattacharjee. Accountability as a service. In *USENIX Steps to Reducing Unwanted Traffic on the Internet*, 2007.

[7] M. Blaze, J. Feigenbaum, and A. D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, 1999.

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

[9] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS Project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan*, Tokyo, Japan, 1994.

[10] Daniel J. Abadi et. al. The Design of the Borealis Stream Processing Engine. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

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

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

[13] T. G. Griffin and J. L. Sobrinho. Metarouting. In *ACM SIGCOMM*, 2005.

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

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

[16] T. Jim and D. Suciu. Dynamically Distributed Query Evaluation. In *ACM Symposium on Principles of Database Systems*, 2001.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), 2000.

[18] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

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

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

[21] 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 *ACM SIGMOD*, June 2006.

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

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

[24] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Multiple Overlay Selection and Intelligent Composition. 2007. UPenn CIS Technical Report No. MS-CIS-07-22, 2007.

[25] MIT. The Chord/DHash Project. http://pdos.csail.mit.edu/chord/.

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

[27] PlanetLab. Global testbed. `http://www.planet-lab.org/`.

[28] Raghu Ramakrishnan and S. Sudarshan. Bottom-Up vs Top-Down Revisited. In *Proceedings of the International Logic Programming Symposium*, 1999.

[29] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

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

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

[32] M. Winslett, C. C. Zhang, and P. A. Bonatti. Peeraccess: a logic for distributed authorization. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 168–179, New York, NY, USA, 2005. ACM Press.

[33] Y. Xie, V. Sekar, M. K. Reiter, and H. Zhang. Forensic analysis for epidemic attacks in federated networks. In *Proc. of the 2001 IEEE Symposium on Security and Privacy*, pages 43–53. IEEE Computer Society, May 2001.

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