Tracking Adversarial Behavior in Distributed Systems with Secure Network Provenance

Wenchao Zhou[♭] Andreas Haeberlen[♭] Boon Thau Loo[♭] Micah Sherr[♯]
[♭]*University of Pennsylvania* [♯]*Georgetown University*
{wenchaoz,ahae,boonloo}@cis.upenn.edu msherr@cs.georgetown.edu

ABSTRACT

This paper presents *secure network provenance* (SNP), a novel technique for tracking down compromised nodes in a distributed system and assessing the damage that they may have caused to other nodes. SNP enables operators to ask the system *why* it is in a certain state – for example, why a suspicious routing table entry is present on a certain router, or where a given cache entry originated. SNP is robust to manipulation; its tamper-evident properties ensure that operators can detect when compromised nodes lie or falsely implicate correct nodes. Thus, compromised nodes can at worst refuse to participate, making their presence evident to operators. We describe an algorithm for answering SNP queries, as well as a proof-of-concept implementation.

1. INTRODUCTION

Operators of networks and distributed systems often find themselves needing to answer a diagnostic or forensic question. Some part of the system is found to be in an unexpected state – for example, a suspicious routing table entry is discovered or a proxy cache is found to contain an unusually large number of advertisements – and the administrator must determine the *causes* of this state before he can decide on an appropriate response. On the one hand, there may be an innocent explanation: the routing table entry may be caused by a misconfiguration, or the cache entries may simply be the result of a workload change. On the other hand, the unexpected state may be the symptom of an ongoing attack: the routing table entry may be the result of route hijacking, and the cache entries may be a side-effect of a malware infection. If the system is indeed under attack, the operators must act quickly to prevent further damage.

Once an attack or intrusion is discovered, a different challenge arises. Suppose the operators have discovered that a certain set of nodes has been compromised. To repair the damage, it may be insufficient to disinfect these specific nodes, since the damage may have already spread to the rest of the system. For example, the adversary may have already polluted a database, altered code, or installed a backdoor. Restoring a backup of the entire system would solve this problem, but such a solution is disruptive and might cause a considerable amount of work to be lost. It would be more preferable if the operators could determine the precise *effects* of the intrusion.

Recent work in the database community on *data provenance* [2] provides a promising new approach to answering these questions. In essence, data provenance tracks and records dependencies between data items in a database. A similar notion in the networking domain – *network provenance* [20, 21] – describes the history and derivations of network state (maintained as *declarative networks* [11]) that results from the execution of a distributed protocol. For example, a provenance system might record that a routing table entry on router `A` was created in response to a route announcement from router `B`, which in turn was triggered by the arrival of two other announcements from routers `C` and `D`, and so on. Conceptually, provenance forms a global dependency graph, whose nodes are the data items and whose edges represent message transmissions or processing steps.

Provenance graphs can easily be used to answer our earlier questions about causes and effects. To determine the causes of data item d, we start from d and follow the edges in the reverse direction until we arrive at a set of leaf nodes, such as local inputs, which have no further dependencies. To determine the effects of d, we follow the edges in the forward direction.

However, existing provenance techniques are designed for a cooperative setting in which all parties behave honestly. They cannot be applied in a setting with untrusted parties or compromised nodes because an adversary can lie about provenance. For example, the adversary can fabricate plausible provenance information for any state introduced, or worse, he can make it appear as if the correct nodes were the source of a detected attack. Since such malicious behavior may prevent or delay discovery of the intrusion, an *unsecured* provenance system can actually slow down the forensic process.

This paper proposes an approach towards *secure network provenance (SNP)*, a framework for utilizing provenance information in a potentially untrustworthy environment. SNP is an equivalent of data provenance in an adversarial setting, answering questions about both the causes and the effects of data items even in the presence of coalitions of Byzantine [9] nodes. Furthermore, to achieve scalability, SNP does not make use of trusted

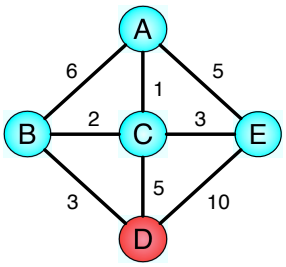

Figure 1: Example network with five routers. Each link is annotated with its cost, and each router is trying to find the best route to D.

third parties. This very conservative threat model allows us to give strong assurances to operators while minimizing an adversary's ability to evade or abuse the system.

Ideally, we would like to guarantee that provenance queries return complete, accurate, and consistent responses. However, this seems infeasible in the absence of a trusted party because nodes can suppress information, lie about where their data came from or how they processed it, destroy or corrupt any local state, or refuse to engage in the protocol. In light of these challenges, SNP provides a slightly weaker property in order to achieve practicality. Specifically, our proposed SNP solution (i) operates only on *observable* data (i.e., data that has directly or indirectly affected at least one correct node); and (ii) may return incomplete responses. However, if a response is incomplete, SNP will eventually identify at least one faulty or compromised node.

A key contribution of this paper is SNP's provenance graph, which enables the use of provenance in an adversarial setting and differs substantially from the graph commonly used in the database community. For example, our provenance representation explicitly models time, message transmissions, and the information that is locally available at each node. A fortuitous consequence of this richer model is that we gain the ability to answer very rich queries (e.g., about the causes of a state *change*, or about consistent snapshots) that substantially generalizes the classical notion of data provenance. Supporting such rich queries further enhances an operator's ability to fight intrusions and malicious behavior.

2. NETWORK PROVENANCE

Network provenance is applicable to a wide range of systems and protocols, but for ease of exposition we describe it in the context of a simple protocol, MinCost, that computes the minimal path cost between all pairs of nodes. We consider the network shown in Figure 1, which consists of five routers that are connected by links of different costs. Each router attempts to find the lowest-cost path to the destination `D`. For now, we assume that there is no malicious behavior.

At any given time, the state of a router `X` is represented as a collection of *tuples*. In this simple example, we only need three types of tuples:

- `link(@X,Y,k)` indicates that router `X` has a direct link to router `Y` with cost `k`;
- `cost(@X,Y,Z,k)` indicates that `X` knows a path via `Y` to `Z` with total cost `k`;
- `bestCost(@X,Y,k)` indicates that the cheapest path known by `X` to `Y` has cost `k`.

The location specifier `@` denotes the node on which the tuple resides. We say that the `link` tuples are *base tuples* because they are part of the static configuration of the routers (we assume that routers have *a priori* knowledge of their local link costs), whereas the `cost` and `bestCost` tuples are *derived* from other tuples according to one of three rules: each router knows the cost of its direct links (R1); it can learn the cost of an advertised route from one of its neighbors that combines a `link` tuple with its `bestCost` to the destination (R2)[1]; and it chooses its own `bestCost` tuple according to the lowest-cost path it currently knows (R3).

To illustrate network provenance, consider the provenance of tuple `bestCost(@C,D,5)`. This tuple can be derived in two different ways. Router `C` knows its direct link to `D` via `link(@C,D,5)`, which trivially produces the tuple `cost(@C,D,D,5)`. Similarly, router `B` derives `cost(@B,D,D,3)` via its direct link with `D`, and since no other path from `B` to `D` offers a lower cost, `B` produces the tuple `bestCost(@B,D,3)`. `B` then combines the knowledge along with `link(@B,C,2)` to derive `cost(@C,B,D,5)` and communicates it to `C`. If we follow each derivation all the way back to base tuples, we obtain the *provenance tree* shown in Figure 2. This tree concisely explains how the tuple at its root came into existence.

3. SECURE NETWORK PROVENANCE

Existing network provenance systems are typically engineered for honest environments in which no party misbehaves. Unfortunately, when an adversary does exist, such provenance systems are easily manipulated. For example, malicious parties can destroy tuples, prevent the system from stabilizing, equivocate by giving different information to different nodes, and provide false provenance information. In this paper, we adopt a conservative threat model and assume that an unknown subset of nodes is Byzantine and can behave arbitrarily.

SNP allows users to detect false or inaccurate provenance data through the use of tamper-evident logs [5] of communicated messages which we call *secure histories*. The details of our approach are described below, but at a high level, SNP achieves secure network provenance by augmenting the standard provenance tree as follows:

- To support historical queries (that is, provenance of past tuples), SNP annotates each tuple with a time interval during which it was valid.

[1]We assume that links are symmetric, that is, the existence of a link $S \rightarrow D$ implies the existence of a link $D \rightarrow S$ with the same cost.

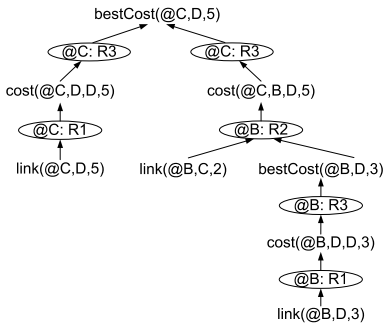

Figure 2: **Provenance of `bestCost(@C,D,5)` at router `C`. The notation `@X` indicates that the tuple was derived at router `X`, and Rx means that derivation x was used.**

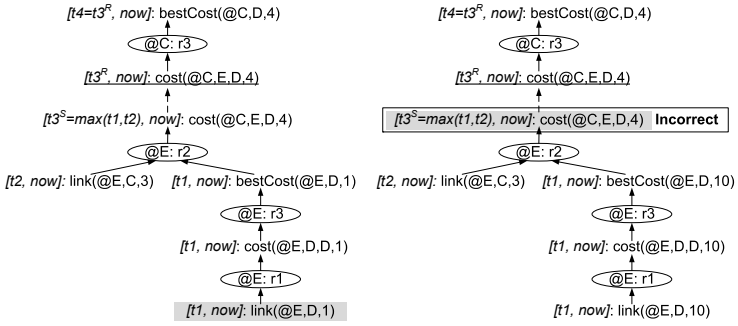

Figure 3: **Secure network provenance of `bestCost(@C,D,4)`. Each tuple is annotated with a time interval (shown as italic) during which the tuple was derivable. Dashed lines indicate network transmissions; and belief tuples are underlined. Shaded boxes highlight manipulations by the compromised router.**

- To indicate detected misbehavior, tuples that have been derived incorrectly are flagged in the tree.
- To reason about transient states, whenever a tuple τ_1 on a node A is partly derived from a tuple τ_2 that existed on another node B during interval $[t_1, t_3]$, SNP adds a *belief tuple* $\bar{\tau}_2$ on A with validity interval $[t_2, t_4]$, where t_2 is the time A learned of τ_2's insertion, and t_4 is the time A learned of τ_2's deletion. $\bar{\tau}_2$ represents A's belief that τ_2 exists on B; thus, τ_1 depends on $\bar{\tau}_2$, which in turn depends on τ_2. Such an annotation provides a convenient way to represent lies and equivocation: in this case, the belief of one node is simply different from the reality on another.

Below, we first outline how provenance trees may be annotated to support historical queries, and then discuss SNP's techniques for securing provenance data.

3.1 Historical Network Provenance

Traditional provenance techniques provide explanations as to a current network state. SNP adds the additional capability to query *historical network provenance*, enabling a user to determine the causes of a prior state.

Suppose each node has a complete and accurate record of all the messages it has sent or received. (We relax this assumption in the following section.) Then SNP can determine the provenance of a tuple τ that existed on a node X at time t as follows:

1. Replay all the messages X has received up to time t, and check whether τ can be derived from tuples that existed at t. If so, annotate τ with $I := [t_1, t_2]$, the largest interval with $t \in I$ during which τ was continuously derivable. If not, flag τ as incorrect.
2. For each tuple τ_i from which τ was derivable during I, do the following:
   i. If τ_i is a base tuple, add a dependency $\tau_i \rightarrow \tau$;
   ii. If τ_i is a local tuple on X, add a dependency $\tau_i \rightarrow \tau$ and recursively determine the provenance of τ_i;
   iii. Otherwise, τ_i must be from another node Y. Let m_i^+ be the message from Y that created τ_i, m_i^- the message from Y that deleted τ_i (if any), and t_i^{R+} and t_i^{R-} be their respective arrival times. Add a belief tuple $\bar{\tau}_i$, annotate it with interval $[t_i^{R+}, t_i^{R-}]$, and create dependencies $\tau_i \rightarrow \bar{\tau}_i \rightarrow \tau$;
   iv. Let t_i^{S+} and t_i^{S-} be the transmission times of m_i^+ and m_i^-. If τ_i was created on Y at t_i^{S+} and destroyed on Y at t_i^{S-} and continuously derivable in between, recursively determine the provenance of τ_i. Otherwise flag τ_i as incorrect.

Reverse Provenance. The algorithm for reverse provenance (that is, determining the *effects* of a datum) is analogous; the main differences are that it considers outgoing rather than incoming messages, and that it terminates at currently extant tuples rather than at base tuples.

3.2 Secure Histories

The algorithm presented in Section 3.1 assumes the existence of an accurate, distributed record of all communicated messages. Unfortunately, reliably capturing all traffic is infeasible in an adversarial setting without relying on a trusted third party.

SNP maintains secure histories by recording messages in tamper-evident logs [5] that are stored at participating nodes. Briefly, the logs involve signing and acknowledging all messages, as well as exchanging information about each message with some other nodes to detect inconsistencies. A useful property of such tamper-evident logs is that they can provably guarantee that *either* a complete and accurate trace of all observable messages is obtained, *or* the identity of at least one compromised node will eventually be learned [5]. In either case, the operators obtain useful information.

Once the identity of the compromised node is established, provenance information derived using SNP can be used to understand the flow of attack, by tracing all messages and network state that result from the compromised node.

There are two points to note here. First, the tamper-evident logs are only guaranteed to contain all *observable* messages, i.e., all messages that directly or transitively affect at least one correct node. If two com-

promised nodes secretly exchange messages but do not otherwise change their behavior, the correct nodes may never learn about this (but they also remain unaffected by it). Second, if a compromised node equivocates, i.e., sends conflicting tuples to different correct nodes, a provenance query that involves one of the tuples may initially appear correct (e.g., while the message with the second tuple is still in transit). However, the tamper-evident logs guarantee that the equivocation is eventually revealed.

3.3 Example

To illustrate how our prototype works, we now describe an example scenario. Suppose the adversary has compromised router `E` in the network depicted in Figure 1, and he wants to snoop on traffic from router `A`. Neither of `A`'s best routes traverses `E`, but the adversary can change this by making it appear that the cost of link `D`–`E` has been reduced to 1. He can do this in at least two ways: by changing the local base tuple to `link(@E,D,1)` or by lying to `C` about the cost of the advertised route to `D`.

Now suppose the operator of router `C` notices that the cost of the route at `C` has changed to 4, and he runs a provenance query on `bestCost(@C,D,4)`. Figure 3 shows the resulting provenance trees, assuming that router `E` is cooperating. If the adversary has changed the base tuple, the operators can see the new tuple (left tree); if `E` had earlier lied to `C`, the corresponding tuple is flagged as incorrect (right tree). If router `E` had refused to cooperate with the query or returned bogus information, the tamper-evident log would have directly identified `E` as faulty. In all cases, the malicious behavior is detected.

3.4 Summary

SNP ensures that, if the actions of an adversary directly or indirectly affect at least one tuple τ on a correct node, then the operators can either 1) obtain a correct and complete provenance tree for τ, which is useful for diagnostics and forensics, or they can 2) eventually identify a compromised node `N`, either because the provenance tree contains a derivation on `N` that is marked as incorrect, or because `N` is found to have tampered with its log.

4. PROOF OF CONCEPT

To show how SNP might be applied to real distributed systems, we have built a proof-of-concept implementation of SNP, which we describe next. We note that our prototype still has many limitations; for example, it is unoptimized, and it is tied to one particular programming model. Hence, our results are not meant as an evaluation of SNP, but rather as evidence that the SNP approach is feasible.

4.1 Prototype Implementation

Programming model: For our prototype, we assume that the distributed system to which SNP is being applied is written in Network Datalog (NDlog) [11], a declarative networking language. While SNP is not specific to a particular programming language, this choice simplifies our prototype since provenance is easy tracked in the declarative model.

An NDlog program consists of a set of *rules* of the form `h :- p1,p2,...,pn`, where `h` is a tuple and the `pi` are predicates on other tuples. The rule stipulates that the tuple `h` is derived when *all* the predicates `pi` are satisfied. For example, the MinCost protocol from Section 2 can be written as

```
r1 cost(@S,D,D,C) :- link(@S,D,C).
r2 cost(@Z,S,D,C1+C2) :- link(@S,Z,C1),
                         bestCost(@S,D,C2).
r3 bestCost(@S,Z,min<C>):-cost(@S,D,Z,C).
```

Implementation: Our prototype is based on existing code from ExSPAN [21] for tracking provenance, on RapidNet [13] for compiling NDlog programs, and on PeerReview [5] for the tamper-evident log. In addition, we implemented a new query engine that uses the algorithm from Section 3.1 to answer provenance queries using the tamper-evident log.

Experimental setup: Our experiments are performed on simulated networks. We instantiate N logical nodes, randomly insert links and link costs such that the average degree of each node is four, and connect the nodes using a simple network simulator that passes each transmitted message directly to the destination. Our setup is sufficient to measure two important overheads of SNP: the log size and the communication cost.

We compare three implementations of the MinCost routing protocol specified above: `MC` is a baseline implementation in RapidNet without provenance, `MC-Prov` is an implementation in ExSPAN that provides provenance in a cooperative setting (without histories), and `MC-SNP` is our SNP prototype. At the start of each run, the nodes only have the base tuples, and we run the system until no more tuples can be derived.

4.2 Overhead

Figure 4 shows the average size of the logs kept on each node as a function of the number of nodes. Although SNP maintains a history of all sent or received messages, the per-node log size is reasonable (e.g., 3.5 MB for a network with 100 nodes) and scales linearly with the number of nodes. Also, note that the logs contain only forensic information, require no further processing, and can be kept on an inexpensive hard disk until they are needed to answer a query.

Figure 5 plots the average amount of data transmitted by a node during an execution. `MC-Prov` requires more communication than `MC` because it tags each tuple with provenance information; SNP causes approximately twice as much communication as `MC-Prov` due to its tamper-evident log, which requires each message to be cryptographically signed and acknowledged. Figure 6 shows a CDF of the total (network-wide) communication cost for answering a provenance query; the median cost is around 15 MB.

Despite the fact that our proof-of-concept implementation of SNP is entirely unoptimized, its overhead may already be practical for some applications. The capac-

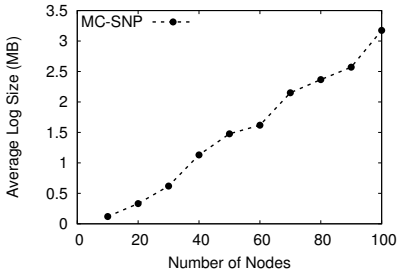

Figure 4: **Average log size (MB) for provenance maintenance.**

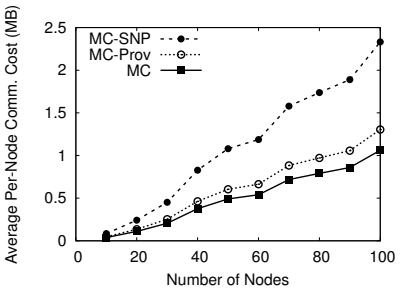

Figure 5: **Average communication cost (MB) for provenance maintenance.**

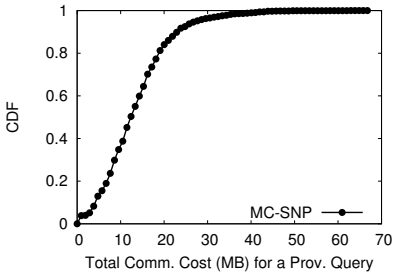

Figure 6: **CDF of total communication cost (MB) for provenance querying.**

ity of commodity hard drives is currently measured in terabytes, so spending a few mega- or even gigabytes on forensic information is not unreasonable. Answering provenance queries requires substantial communication; however, our initial implementation downloads the *entire* log for all relevant nodes, when a small portion of the log may suffice. A few straightforward optimizations (e.g., taking snapshots periodically [5], partitioning of independent tuples [4], batching multiple messages [4] to reduce cryptographic overhead, and caching and maintaining provenance eagerly [21]) will likely reduce this cost dramatically.

4.3 Functionality Check

We implemented the attacks described in Section 3.3 and ran the corresponding queries. As expected, our prototype returned the provenance trees shown in Figure 3, which can be used by operators to track down the compromised routers.

5. RELATED WORK

Provenance: SNP builds on recent work on the ExSPAN distributed provenance system [21], and in particular, its use of distributed queries for efficiently maintaining and querying provenance in a distributed setting. However, ExSPAN was designed for a cooperative environment and can return inaccurate provenance trees when compromised nodes are present. Additionally, ExSPAN cannot reliably answer provenance queries on tuples that are unstable or have already disappeared. Sprov [6] implements *secure* provenance for individual documents; however, it lacks essential features that are required in a distributed system, e.g., a consistency check to ensure that nodes are processing messages in a way that is consistent with their current state. Pedigree [16] captures provenance at the network layer in the form of per-packet *tags* that store a history of all nodes and processes that manipulated the packet. It assumes a trusted environment, and its set-based provenance is less expressive compared to SNP's graph-based dependency structure.

Distributed debugging: Our work differs from most existing work on debugging distributed systems in that we consider a broader class of potential problems. Systems such as Magpie [1], D3S [10], Pip [17], and P2 Monitor [18] can diagnose symptoms that arise from a problem with the distributed algorithm (such as a bug or a race condition), while SNP can also diagnose symptoms that are caused by certain nodes not *following* the algorithm, e.g., by making incorrect state transitions or by fabricating messages.

Intrusion detection: Distributed intrusion detection systems (IDS) such as BackTracker [8] can be used to establish causal relationship among events at different nodes, based on information gathered from communication and system logs. Given an IDS alert, this enables one to detect the source or effects of an attack indicated by the alert. These systems often assume that logs at each node are trusted, an assumption that may not hold true when nodes are compromised or span multiple administrative domains. SNP also enables a wider range of queries beyond simply querying for causal relationships among different hosts.

Accountability: Accountability systems like PeerReview [5] and NetReview [4] can detect when nodes in a distributed system deviate from the algorithm they are expected to run. However, unlike SNP, these systems cannot detect or diagnose problems that result from interactions between multiple nodes (such as an instance of Bad Gadget [3] in interdomain routing), or problems that are related to inputs or unspecified aspects of the algorithm. Finally, accountability systems focus on detecting faults, whereas SNP also offers support for assessing the extent of, and recovering from, their effects.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed secure network provenance (SNP), a novel approach to tracking adversarial behavior in distributed systems. SNP helps operators with forensics and mitigation by answering questions about the *causes* and the *effects* of specific system states. One of the key contributions of SNP is a secure provenance model, that is made possible by adapting the concept of data provenance, which usually assumes a cooperative setting, to systems where an unknown subset of the nodes is controlled by a Byzantine adversary. We have proposed an algorithm for answering SNP queries,

and early results are reported from a proof-of-concept implementation based on ExSPAN and PeerReview.

Although our initial results seem encouraging, there are several challenges that must be addressed before SNP can be deployed in a production environment. This includes adding optimizations to reduce overheads, as well as some more fundamental challenges discussed below.

Other programming models: Our initial prototype assumes that the distributed protocol is specified in a declarative language. This representation is very convenient for tracking provenance, but it also prevents our prototype from being used with legacy applications or new applications written in traditional imperative languages. However, the SNP concept itself is general and can be applied to other programming models. For example, if a legacy system exposes its relevant network state and inter-node communication to our system, our techniques for constructing provenance and verifying logs can still be applied. To validate legacy support via a concrete application, we plan to integrate SNP with open-source routing suites (e.g. Quagga [15]). By securely establishing dependency relationships between BGP advertisements, our system can track packets as they traverse through the network, reason about suspicious routing table updates, and detect policy conflicts across ASes.

In the absence of explicit rules to capture data dependencies, additional mechanisms (such as taint analysis [14] or programmer annotations) could be used to correlate changes in the network state to incoming/outgoing network messages.

Extensible query language: An important benefit of using a database approach towards maintaining and querying network provenance is the potential to utilize declarative languages for customizing the specific provenance of interest. Zhou *et al.* [20, 21] demonstrate such capabilities using *NDlog* as the query language, typically by issuing queries asking "why" questions on how a particular network state is derived based on its origins.

We intend to further extend the types of queries supported in SNP by providing mechanisms for querying provenance in the forward direction (in addition to the traditional reverse direction), and to enable *time-traveling* for querying the history of a particular network state (i.e., to retrieve a consistent provenance state at a particular period of time, or to discover how state changed over time). Time-traveling in databases have been extensively explored in the past as *temporal databases* [19]. However, it is still an open challenge how provenance can be stored and queried efficiently in traditional databases with versioning capabilities, and we believe the same challenges also apply to SNP.

Finally, given that provenance is essentially a graph, the database community has recently proposed the use of graph-based languages [7] for formulating queries and transformations over provenance data. We plan to explore distributed variants of these graph query languages, which we believe can be transformed into *NDlog* queries for execution.

Confidentiality: In some instances, it may be appropriate to treat provenance data as confidential. This is particularly important when SNP is applied to a multi-domain setting, where each autonomous system may limit the provenance information that can be seen by other nodes further downstream. Provenance information can be hidden (encrypted) based on roles, security levels or customized requirements.

A naïve implementation of provenance may result in information leakage by exposing sensitive information to the recipients of the tuples, some of which may be unauthorized to access the leaked data. We conjecture that one can utilize *information hiding* techniques [12] to support fine-grained confidentiality controls, by *partially* encrypting provenance in a key-based tree-structured architecture.

7. REFERENCES

[1] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. OSDI*, 2004.

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

[3] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking*, 10(2):232–243, April 2002.

[4] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proc. NSDI*, Apr 2009.

[5] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. SOSP*, Oct 2007.

[6] R. Hasan, R. Sion, and M. Winslett. Preventing history forgery with secure provenance. *ACM Transactions on Storage*, 5(4):1–43, 2009.

[7] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *Proc. SIGMOD*, 2010.

[8] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *Proc. NDSS*, 2005.

[9] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[10] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: debugging deployed distributed systems. In *Proc. NSDI*, pages 423–437, 2008.

[11] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. *Communications of the ACM*, 52(11):87–95, 2009.

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

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

[14] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. NDSS*, Feb 2005.

[15] Quagga Routing Suite. `http://www.quagga.net/`.

[16] A. Ramachandran, K. Bhandankar, M. Bin Tariq, and N. Feamster. Packets with provenance. Technical Report GT-CS-08-02, Georgia Tech, 2008.

[17] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. NSDI*, May 2006.

[18] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *Proc. EuroSys*, Apr. 2006.

[19] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communication of the ACM*, 34(10):78–92, 1991.

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

[21] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *Proc. SIGMOD*, 2010.