10-2011

Secure Network Provenance

Wenchao Zhuo
University of Pennsylvania

Qiong Fei
University of Pennsylvania

Arjun Narayan
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania, ahae@cis.upenn.edu

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

See next page for additional authors

Recommended Citation

Wenchao Zhuo, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr, "Secure Network Provenance", . October 2011.

Zhuo, W., Fei, Q., Narayan, A., Haeberlen, A., Loo, B., & Sherr, M., Secure Network Provenance, *23rd ACM Symposium on Operating Systems Principles*, Oct. 2011, doi: 10.1145/2043556.2043584

Secure Network Provenance

Abstract

This paper introduces secure network provenance (SNP), a novel technique that enables networked systems to explain to their operators why they are in a certain state – e.g., why a suspicious routing table entry is present on a certain router, or where a given cache entry originated. SNP provides network forensics capabilities by permitting operators to track down faulty or misbehaving nodes, and to assess the damage such nodes may have caused to the rest of the system. SNP is designed for adversarial settings and is robust to manipulation; its tamper-evident properties ensure that operators can detect when compromised nodes lie or falsely implicate correct nodes. We also present the design of SNooPy, a general-purpose SNP system. To demonstrate that SNooPy is practical, we apply it to three example applications: the Quagga BGP daemon, a declarative implementation of Chord, and Hadoop MapReduce. Our results indicate that SNooPy can efficiently explain state in an adversarial setting, that it can be applied with minimal effort, and that its costs are low enough to be practical.

Disciplines

Computer Sciences

Comments

Zhuo, W., Fei, Q., Narayan, A., Haeberlen, A., Loo, B., & Sherr, M., Secure Network Provenance, *23rd ACM Symposium on Operating Systems Principles*, Oct. 2011, doi: 10.1145/2043556.2043584

Author(s)

Wenchao Zhuo, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr

Secure Network Provenance

Wenchao Zhou
University of Pennsylvania

Qiong Fei
University of Pennsylvania

Arjun Narayan
University of Pennsylvania

Andreas Haeberlen
University of Pennsylvania

Boon Thau Loo
University of Pennsylvania

Micah Sherr
Georgetown University

ABSTRACT

This paper introduces *secure network provenance (SNP)*, a novel technique that enables networked systems to explain to their operators *why* they are in a certain state – e.g., why a suspicious routing table entry is present on a certain router, or where a given cache entry originated. SNP provides network forensics capabilities by permitting operators to track down faulty or misbehaving nodes, and to assess the damage such nodes may have caused to the rest of the system. SNP is designed for adversarial settings and is robust to manipulation; its tamper-evident properties ensure that operators can detect when compromised nodes lie or falsely implicate correct nodes.

We also present the design of SNooPy, a general-purpose SNP system. To demonstrate that SNooPy is practical, we apply it to three example applications: the Quagga BGP daemon, a declarative implementation of Chord, and Hadoop MapReduce. Our results indicate that SNooPy can efficiently explain state in an adversarial setting, that it can be applied with minimal effort, and that its costs are low enough to be practical.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*; D.4.5 [**Software**]: Operating Systems—*Reliability*

General Terms

Algorithms, Design, Reliability, Security

Keywords

Accountability, Byzantine faults, Distributed systems, Evidence, Provenance, Security

1. INTRODUCTION

Operators of 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. The operators must determine the *causes* of this state before they can decide on an appropriate response. On the one hand, there may be an innocent explanation: the routing table entry could be the result of a misconfiguration, and the cache entries could have appeared due to a workload change. On the other hand, the unexpected state may be the symptom of an ongoing attack: the routing table entry could be the result of route hijacking, and the cache entries could be a side-effect of a malware infection. If an attack or misconfiguration is discovered, the operators must determine its *effects*, such as corrupted state or configuration changes on other nodes, so that these nodes can be repaired and the system brought back to a correct state.

In this paper, we consider forensics in an *adversarial* setting, that is, we assume that a faulty node does not necessarily crash but can also change its behavior and continue operating. To be conservative, we assume that faults can be Byzantine [24], i.e., a faulty node can behave arbitrarily. This covers a wide range of faults and misbehavior, e.g., cases where a malicious adversary has compromised some of the nodes, but also more benign faults, such as hardware failures or misconfigurations. Getting correct answers to forensic queries in an adversarial setting is difficult because the misbehaving nodes can lie to the querier. For example, the adversary can attempt to conceal his actions by causing his nodes to fabricate plausible (but incorrect) responses to forensic queries, or he can attempt to frame correct nodes by returning responses that blame his own misbehavior on them. Thus, the adversary can gain valuable time by misdirecting the operators and/or causing them to suspect a problem with the forensic system itself.

Existing forensic systems are either designed for non-adversarial settings [43, 50] or require some trusted components, e.g., a trusted virtual-machine monitor [3, 21], a trusted host-level monitor [27], a trusted OS [29], or trusted hardware [7]. However, most components that are available today are not fully trustworthy; OSes and virtual machine monitors have bugs, which a powerful adversary could exploit, and even trusted hardware is sometimes compromised [20]. We argue that it is useful to have alternative techniques available that do not require this type of trust.

We introduce *secure network provenance (SNP)*, a technique for building forensic systems that can operate in a *completely untrusted* environment. We assume that the adversary may have compromised an arbitrary subset of the nodes, and that he may have complete control over these nodes. On the one hand, this very conservative threat model requires some compromises: an SNP system can only answer queries about observable network state—i.e., state that has directly or indirectly affected at least one correct node—and its responses can be incomplete, although the missing parts are always clearly identified. On the other hand, an SNP system provides strong, provable guarantees: it ensures that an observable symptom of a fault or an attack can always be traced to a specific event—passive evasion or active misbehavior—on at least one faulty node, even when an adversary attempts to prevent this.

Two existing concepts, data provenance and tamper-evident logging, can provide a starting point for building SNP systems. Data provenance [4, 50] tracks and records data dependencies as data flows through the system. In the context of distributed systems, network provenance [50] is captured as a global dependency graph, where vertices are data items that represent state at a particular node, and edges represent local processing or message transmissions across nodes. This graph can then be used to answer forensic queries. Tamper-evident logging [17] can record data in such a way that forgeries, omissions, and other forms of tampering can be detected and proven to a third party.

However, as is often the case in computer security, a simple layering of these two concepts fails to achieve the desired goal. If an existing network provenance system, say ExSPAN [50], were combined with a system like PeerReview [17] that supports tamper-evident logging, an adversary could potentially subvert the resulting system by attacking it twice. The first attack would corrupt the system's internal data structures; this would require a protocol violation that PeerReview could detect, but not diagnose or repair. With the data structures suitably damaged, the adversary could then carry out the second attack without further protocol violations, and without leaving visible traces in the provenance system. Thus, the second attack would be invisible.

We have designed SNooPy, a system that provides secure network provenance by combining evidence-based distributed query processing with a novel provenance model that is specially designed with fault detection in mind. We have formalized SNP's security properties, and we have proven that SNooPy satisfies them. To demonstrate SNooPy's practicality and generality, we have implemented a prototype, and we have applied it to three example applications: the Quagga BGP daemon [35], a declarative implementation of Chord [26], and Hadoop MapReduce [12]. Our evaluation demonstrates SNooPy's ability to solve real-world forensic problems, such as finding the causes and effects of BGP misconfigurations, DHT routing attacks, and corrupt Hadoop mappers; our results also show that SNooPy's costs (additional bandwidth, storage, and computation) vary with the application but are low enough to be practical. In summary, we make the following contributions:

- A provenance graph for causal, dynamic, and historical provenance queries that is suitable for SNP (Section 3);
- SNP, a method to securely construct network provenance graphs in untrusted environments (Section 4);
- The design of SNooPy, a system that implements SNP for the provenance graph presented earlier (Section 5);
- A proof of correctness for SNooPy (included in the appendix);
- An application of SNooPy to Quagga, Chord, and Hadoop MapReduce (Section 6); and
- A quantitative evaluation (Section 7).

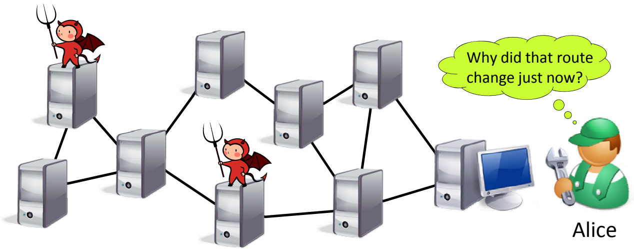


Figure 1: Motivating scenario. Alice is running a distributed system and observes some unexpected behavior that may indicate a fault or an attack.

2. OVERVIEW

Figure 1 illustrates the scenario that we are addressing in this paper. An administrator, here called Alice, is operating a distributed system – perhaps a cluster, a corporate network, or a content distribution system. At some point, Alice observes some unexpected behavior in the system and decides to investigate whether the behavior is legitimate or perhaps a symptom of a fault or an attack. Our goal is to enable Alice to query the system about the causes and effects of the unexpected behavior, and to obtain reliable results.

To achieve this goal, we extend each node in the system with a monitoring component that maintains some forensic information. We refer to the system that is being monitored as the *primary system* and to our additional components as the *provenance system*. To be useful to Alice, the provenance system should have the following two high-level properties:

- When the queried behavior is legitimate, the system should return a complete and correct explanation.
- When the queried behavior is a symptom of a fault or misbehavior, the explanation should tie it to a specific event on a faulty or misbehaving node.

By behavior, we mean a state change or a message transmission on any node. We assume that Alice knows what behavior is legitimate, e.g., because she knows which software the system was expected to run.

2.1 Threat model

Since we would like to enable Alice to investigate a wide range of problems, ranging from simple misconfigurations to hardware faults and even clandestine attacks, we conservatively assume Byzantine faults [24], i.e., that an adversary may have compromised an unknown subset of the nodes, and that he has complete control over them. Thus, the non-malicious problems are covered as a special case. We assume that the adversary can change both the primary system and the provenance system on these nodes, and he can read, forge, tamper with, or destroy any information they are holding. We also assume that no nodes or components of the system are inherently safe, i.e., Alice does not a priori trust any node other than her own local machine.

Handling such a broad range of faults is challenging because Alice cannot be sure that any data she is receiving is actually correct. When she queries a compromised node, the adversary can cause that node to lie or equivocate. In particular, he can try to forge a plausible explanation for the symptoms Alice has observed, or he can try to make it appear as if the symptoms were caused by a different node. If this is not prevented, Alice could overlook the attack entirely or waste time trying to repair the wrong nodes.

2.2 Approach

Our approach to this challenge is to construct a distributed data structure called the *provenance graph* which, at a high level, tracks how data flows through the system. Data provenance itself is not a new concept—it has been explored by the database and the system community [4, 10, 19, 29, 47, 50]—but most existing provenance systems are designed for non-adversarial settings and lack features that are necessary for forensics. For example, existing systems focus on explaining state that exists at query time ("Why does τ exist?"), which would allow an adversary to thwart Alice's investigation simply by deleting data that implicates him. To support forensics, we additionally provide *historical queries* ("Why did τ exist at time t?") and *dynamic queries* ("Why did τ (dis)appear?'); to assist with recovery, we also provide *causal queries* ("What state on other nodes was derived from τ?"), which can be used to determine which parts of the system have been affected and require repair.

Our key contribution, however, is to *secure* the provenance graph. Ideally, we would like to correctly answer Alice's queries even when the system is under attack. However, given our conservative threat model, this is not always possible. Hence, we make the following two compromises: first, we only demand that the system answer provenance queries about behavior that is *observable by at least one correct node* [15]; in other words, if some of the adversary's actions never affect the state of any correct node, the system is allowed to omit them. Second, we accept that the system may sometimes return an answer that is incorrect or incomplete, as long as Alice can a) tell which parts of the answer are affected, and she can b) learn the identity of at least one faulty node. In a forensic setting, this seems like a useful compromise: any unexpected behavior that can be noticed by Alice is observable by definition, and even a partial answer can help Alice to determine whether a fault or misbehavior has occurred, and which parts of the system have been affected.

2.3 Provenance and confidentiality

If Alice can query any datum on any node, she can potentially learn the full state of the entire system. Throughout this paper, we will assume that Alice is authorized to have this information. In centrally managed systems, there are typically at least some individuals (e.g., the system administrators) who have that authority. Examples of such systems include academic or corporate networks as well as infrastructure services—such as Akamai's CDN—that are physically distributed but controlled by a single entity.

In systems without central management, it is sometimes possible to partition the state among different managers. For example, in Amazon's Elastic MapReduce service, the owner of a given MapReduce job could be authorized to issue queries about that specific job while being prevented from querying jobs that belong to other customers. In other cases, abstractions can be used to hide confidential details from unauthorized queriers. SNP includes extensions to the provenance graph that can selectively conceal how certain parts of a node's state were derived. As discussed in Section 3.4, the resulting graph can be queried without disclosing the node's actual computation.

2.4 Strawman solutions

It is natural to ask whether our goals could be achieved by using some combination of an existing fault detection system, such as PeerReview [17], and/or an existing network provenance system, such as ExSPAN [50]. However, a simple combination of these two systems is insufficient for the following reasons.

Individually. In isolation, neither of the two systems can achieve our goals. PeerReview can detect when nodes deviate from the algorithm they are expected to run, but it provides no mechanisms for detecting or diagnosing problems that result from interactions between multiple nodes (such as an instance of BadGadget [11] in interdomain routing), or problems that are related to nodes lying about their local inputs or deliberately slowing down their execution. ExSPAN captures the interactions among nodes via provenance, but cannot detect when compromised nodes lie about provenance.

Layering. A natural approach to addressing ExSPAN's security vulnerabilities is simply to layer ExSPAN over PeerReview. However, this approach also fails to achieve the desired security guarantees. First, PeerReview reports faults with a certain delay; thus, a compromised node has a window of opportunity in which it can corrupt the provenance graph. Even if detection is nearly instantaneous, simply identifying the faulty node is not sufficient: since the graph is itself distributed, effects of the corruption can manifest in parts of the provenance graph that are stored on other nodes, and there is no way for the layered approach to detect this easily. This means that once a fault is detected by PeerReview, the results of further provenance queries (e.g., to find other compromised nodes, or to locate corrupted state) can no longer be trusted, and the entire provenance system is rendered unusable.

Our integrated solution. Achieving hard guarantees for secure provenance requires rethinking both ExSPAN and PeerReview. Instead of layering one system over the other, we tightly integrate the process of provenance generation and querying with the underlying fault detection system. Providing secure network provenance involves a fundamental redesign of ExSPAN's query and provenance model to enable tamper-evident query processing and the generation of evidence against faulty nodes, which can be used for further investigations.

In the following sections, we not only demonstrate that our integrated approach achieves the desired high-level properties introduced earlier at a cost that is low enough to be practical, we also experimentally validate its usefulness by performing forensic analysis on several existing applications. An additional benefit of this tight integration and our richer provenance model is that we can naturally support richer forensic queries, such as historical, dynamic, and causal provenance queries.

3. PROVENANCE GRAPH

In this section, we introduce our system model, and we define an 'ideal' provenance graph G, based on the true actions of each node. Of course, if faulty nodes can lie about their actions or suppress information, a correct node that is processing a provenance query may not be able to reconstruct G entirely. However, as we will show in the following sections, SNP can reconstruct a close approximation G_ν of G.

3.1 System model

For ease of exposition, we adopt a system model that is commonly used in database systems to reason about data provenance. In this model, the state of the primary system is represented as *tuples*, and its algorithm is represented as *derivation rules* [50], which describe how tuples are derived from the system's inputs. Few practical systems are explicitly built in terms of tuples and derivation rules, but this is not required to apply SNP: in Section 5.3, we describe three general techniques for extracting tuples and derivations from existing systems, and in Section 6 we report how we applied these techniques to Quagga, Chord, and Hadoop MapReduce.

Each node in a distributed system has its own set of tuples, and derivation rules can span multiple nodes. For example, the state of a router r might consist of tuples such as `link(@r,a)` to show that r has a link to a, or `route(@r,b,c)` to show that r knows a route to b on which the next hop is c. Here, `link` and `route` are the names of specific relations, and `@r` indicates that the tuple is maintained on r. The lower-case letters are constants; we later use upper-case letters for variables. Where the specific relation does not matter, we simply write $\tau@n$ to denote a tuple τ on a node n.

Tuples can either be *base tuples* or *derived tuples*. Base tuples correspond to local inputs that are assumed to be true without derivations, e.g., a list of physical links that is input to a routing protocol. Derived tuples are obtained from other tuples through a derivation rule of the form $\tau@n \leftarrow \tau_1@n_1 \wedge \tau_2@n_2 \wedge \cdots \wedge \tau_k@n_k$. This is interpreted as a conjunction: tuple τ should be derived on n whenever all τ_i exist on their respective nodes n_i, and τ should then continue to exist until at least one of the τ_i disappears. (If a tuple has more than one derivation, we can distinguish between them using a logical reference counter.) When a derivation rule spans multiple nodes, the nodes must notify each other of relevant tuple changes: if a node i has a rule that depends on a tuple $\tau@j$, j must send a message $+\tau$ to i whenever τ is derived or inserted as a base tuple, and j must send $-\tau$ to i whenever τ is underived or removed. We require that all derivations are finite and have no cyclic dependencies. This can be achieved by carefully writing the derivation rules, and it holds for our three example applications.

We assume that each node applies its rules deterministically. Thus, we can model the expected behavior of a node i as a state machine A_i, whose inputs are incoming messages and changes to base tuples, and whose outputs are messages that need to be sent to other nodes. An *execution* of the system can then be represented as a sequence of message transmissions, message arrivals, base tuple insertions, and base tuple deletions. We say that a node i is *correct* in an execution e if i's outputs in e are legal, given A_i and i's inputs in e. Otherwise we say that i is *faulty* in e.

Routing example. The derivation rule `route(@R,C,B)` $\leftarrow$ `link(@R,B)` $\wedge$ `route(@B,C,D)` expresses network reachability in a router: a router R has route to C via B (`route(@R,C,B)`) whenever it has a link to another router B (`link(@R,B)`) that already has a route to C via some third router D (`route(@B,C,D)`). Here, R, B, C, and D are variables that can refer to any router. If we declare the `link` tuples to be base tuples and add another rule to say that each router has a route to its immediate neighbors, the resulting system implements a simplified form of path-vector routing [26].

3.2 Vertices and edges

Having explicit derivation rules makes it very easy to see the provenance of a tuple: if a tuple τ was derived from other tuples $\tau_1, \ldots, \tau_k$, then τ's immediate provenance simply consists of all the τ_i taken together. To capture transitive provenance, we can define, for any execution e, a *provenance graph* $G(e) = (V(e), E(e))$, in which each vertex $v \in V(e)$ represents a state or state change, and each edge (v_1, v_2) indicates that v_1 is part of the provenance of v_2. The complete explanation for the existence of a tuple τ in e would then be a subtree that is embedded in $G(e)$ and rooted at the vertex that corresponds to τ. The leaves of this subtree consist of base tuple insertions or deletions, which require no further explanation.

$V(e)$ consists of twelve vertex types. The following seven types are used to represent local states and state changes:

- INSERT(n, τ, t) and DELETE(n, τ, t): Base tuple τ was inserted/deleted on node n at time t;
- APPEAR(n, τ, t) and DISAPPEAR(n, τ, t): Tuple τ appeared/disappeared on node n at time t;
- EXIST$(n, \tau, [t_1, t_2])$: Tuple τ existed on node n during interval $[t_1, t_2]$; and
- DERIVE(n, τ, R, t) and UNDERIVE(n, τ, R, t): Tuple τ was derived/underived on n via rule R at time t.

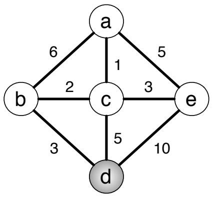
In contrast to other provenance graphs, such as the one in [50], the graph G we present here has an explicit representation for state changes, which is useful to support dynamic queries. G also retains information about tuples that no longer exist, which is necessary for historic queries; note particularly that vertices such as DELETE, UNDERIVE, and DISAPPEAR would not be necessary in a provenance graph that contains only extant tuples. The timestamps t should be interpreted relative to node n.

The remaining five vertex types are used to represent interactions between nodes. For the purposes of SNP, it is important that each vertex v has a specific node that is 'responsible' for it. (We will refer to this node as HOST(v).) To achieve this property, derivations and underivations from remote tuples must be broken up into a sequence of smaller steps that can each be attributed to a specific node. For example, when a rule $\tau_1@i \leftarrow \tau_2@j$ is triggered, we do not simply connect τ_1's DERIVE vertex to τ_2's APPEAR vertex; rather, we say that the provenance of τ_1's derivation was i's *belief* that τ_2 had appeared on j, which was caused by the arrival of $+\tau_2$ on i, the transmission of $+\tau_2$ by j, and finally the appearance of τ_2 on j. Thus, if j's message is later found to be erroneous, i's belief—and thus its derivation—is still legitimate, and the error can be attributed to j. The specific vertex types are the following:

- SEND$(n, n', \pm\tau, t)$: At time t, node n sent a notification to node n' that tuple τ has appeared/disappeared; and
- RECEIVE$(n, n', \pm\tau, t)$: At time t, node n received a message from node n' that tuple τ has appeared/disappeared.
- BELIEVE-APPEAR(n, n', τ, t) and BELIEVE-DISAPPEAR (n, n', τ, t): At time t, node n learned of the (dis)appearance of tuple τ on node n';
- BELIEVE$(n, n', \tau, [t_1, t_2])$: During $[t_1, t_2]$, node n believed that tuple τ existed on node n';

Finally, we introduce a *color* for each vertex $v \in V(e)$. Colors are used to indicate whether a vertex is legitimate: correct vertices are black, and faulty vertices are red. For example, if a faulty node i has no tuple τ but nevertheless sends a message $+\tau$ to another node, $\tau@i$ has no legitimate provenance, so we use a red SEND vertex to represent the transmission of $+\tau$. In Section 4.2, we will introduce a third color, yellow, for vertices whose true color is not yet known.

3.3 Example: Minimum cost routing



As a simple example, consider the network depicted on the right, which consists of five routers that are connected by links of different costs. Each router attempts to find the lowest-cost path to router d using a MinCost protocol. There are 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 to `Y` via `Z` with total cost `K`; and `bestCost(@X,Y,K)` indicates that the cheapest path known by `X` to `Y` has cost `K`. 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, and that links are symmetric), whereas `cost` and `bestCost` tuples are derived from other tuples according to one of three derivation 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 (R2); and it chooses its own `bestCost` tuple according to the lowest-cost path it currently knows (R3).

Figure 2 shows an example of a provenance tree for the 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 `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,d,b,5)` and communicates it to `c`.

3.4 Constraints and 'maybe' rules

We now introduce two extensions to the provenance graph. The first extension is a second type of rule, called a *'maybe' rule* and written $\tau@n \xleftarrow{maybe} \tau_1@n_1 \wedge \ldots \wedge \tau_k@n_k$, which stipulates that the tuple τ on node n *may* be derived from tuples $\tau_1@n_1, \ldots, \tau_k@n_k$, but that the derivation is optional. In other words, as long as all of the underlying tuples are present, node n is free to decide whether or not to derive τ, and it is free to change its decision while the underlying tuples still exist. The rule merely describes τ's provenance if and when it exists.

There are at least two situations in which 'maybe' rules are useful. The first involves a node on which some rules or tuples are confidential. In this case, the node can be assigned *two* sets of rules: one full set for the actual computation (without 'maybe' rules) and another to define provenance, in which the confidential computation is replaced by 'maybe' rules. The second set can then be safely revealed to queriers. Another situation involves a node with a black-box computation, for which only the general dependencies are known. For example, a node n might choose a tuple τ from a set of other tuples, but the details of the decision process might not be known (e.g., because it is performed by a third-party binary). In this case, 'maybe' rules can be used to infer provenance by observing the set of tuples: if all the τ_i exist, we cannot predict whether τ will appear, but if τ *does* appear, it must have been derived from the τ_i.

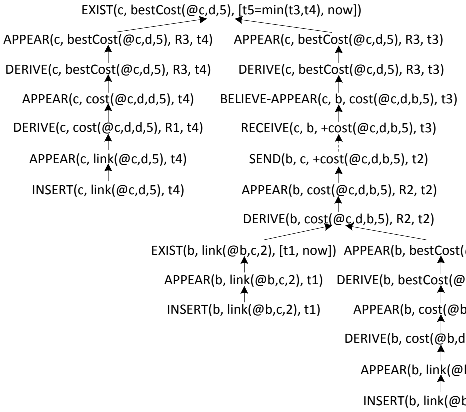

EXIST(c, bestCost(@c,d,5), [t5=min(t3,t4), now])
APPEAR(c, bestCost(@c,d,5), R3, t4)
APPEAR(c, bestCost(@c,d,5), R3, t3)
DERIVE(c, bestCost(@c,d,5), R3, t4)
DERIVE(c, bestCost(@c,d,5), R3, t3)
APPEAR(c, cost(@c,d,d,5), t4)
BELIEVE-APPEAR(c, b, cost(@c,d,b,5), t3)
DERIVE(c, cost(@c,d,d,5), R1, t4)
RECEIVE(c, b, +cost(@c,d,b,5), t3)
APPEAR(c, link(@c,d,5), t4)
SEND(b, c, +cost(@c,d,b,5), t2)
INSERT(c, link(@c,d,5), t4)
APPEAR(b, cost(@c,d,b,5), R2, t2)
DERIVE(b, cost(@c,d,b,5), R2, t2)
EXIST(b, link(@b,c,2), [t1, now])
APPEAR(b, bestCost(@b,d,3), t2)
APPEAR(b, link(@b,c,2), t1)
DERIVE(b, bestCost(@b,d,3), R3, t2)
INSERT(b, link(@b,c,2), t1)
APPEAR(b, cost(@b,d,d,3), t2)
DERIVE(b, cost(@b,d,d,3), R1, t2)
APPEAR(b, link(@b,d,3), t2)
INSERT(b, link(@b,d,3), t2)

Figure 2: Provenance of `bestCost(@c,d,5)` at `c`

The second extension is intended for applications where the presence of *constraints* prevents us from modeling the state as completely independent tuples. For example, given tuples α and β, an application might derive *either* a tuple γ *or* a tuple δ, but not both. Modeling this with disjunctive rules would lose important information: if tuple δ replaces tuple γ, the appearance of δ and the disappearance of γ are not merely independent events, they are causally related. Thus, the explanation of δ's appearance should include the disappearance of γ. In G, we represent this by a direct edge between the corresponding APPEAR and DISAPPEAR vertices.

3.5 Graph construction

Conceptually, we can think of the provenance graph $G(e)$ as being constructed incrementally as the execution e unfolds – each new derivation, tuple insertion or deletion, or message transmission/arrival causes some new vertices to be added and/or existing BELIEVE and EXIST vertices to be updated. In practice, our implementation does not store the vertices and edges themselves; rather, it records only enough information to securely construct the subgraphs of $G(e)$ that are relevant to a given query.

Appendix B specifies the full algortihm that computes $G(e)$ for any execution e. We do not present the details of this algorithm here, but we briefly anticipate three of its key properties. The first property says that the graph can be constructed incrementally:

Theorem 1 *If an execution e_1 is a prefix of an execution e_2, then $G(e_1)$ is a subgraph of $G(e_2)$.*

This holds because, at least conceptually, $G(e)$ contains vertices and edges for all tuples that have ever existed; vertices can be added but not removed. (Of course, our practical implementation has only limited storage and must eventually 'forget' about old vertices and edges.) Theorem 1 makes it possible to answer queries while the system is still running, without risking an inaccurate result. Graph construction is also compositional:

Theorem 2 *To construct the vertices $v \in V(e)$ with* HOST$(v) = i$*, it is sufficient to run the algorithm on the events that have occurred on i.*

Briefly, this holds because G has been carefully designed to be partitionable by nodes, and because derivations from remote tuples (which span multiple nodes) have been split into several steps that can each be attributed to a specific node. Compositionality is crucial for a scalable implementation because it implies that each node's subgraph of G can be reconstructed independently. Thus, we need only reconstruct those subgraphs that are relevant for a given query.

Finally, the graph construction algorithm uses the colors appropriately:

Theorem 3 *All the vertices v in $G(e)$ with $\textsc{host}(v) = i$ are black if, and only if, i is correct in e.*

Thus, if we encounter a red vertex in the provenance graph, we know that the corresponding node is faulty or has misbehaved. The proofs for these theorems are included in Appendix B.

4. SECURE NETWORK PROVENANCE

The definition of the provenance graph G in the previous section assumes that, at least conceptually, the entire execution e of the primary system is known. However, in a distributed system without trusted components, no single node can have this information, especially when nodes are faulty and can tell lies. In this section, we define SNP, which constructs an approximation G_ν of the 'true' provenance graph G that is based on information available to correct nodes.

4.1 Using evidence to approximate G

Although each node can observe only its own local events, nodes can use messages from other nodes as *evidence* to reason about events on these nodes. Since we have assumed that messages can be authenticated, each received message m is evidence of its own transmission. In addition, we can demand that nodes attach some additional information $\varphi(m)$, such as an explanation for the transmission of m. Thus, when a provenance query is issued on a correct node, that node can collect some evidence ϵ, such as messages it has locally received, and/or messages collected from other nodes. It can then use this evidence to construct an approximation $G_\nu(\epsilon)$ of $G(e)$, from which the query can be answered. For the purposes of this section, we will assume that $\varphi(m)$ describes the sender's entire execution prefix, i.e., all of its local events up to and including the transmission of m. Of course, this would be completely impractical; our implementation in Section 5 achieves a similar effect in a more efficient way.

4.2 Limitations

When faulty nodes are present, we cannot always guarantee that $G_\nu(\epsilon) = G(e)$. There are four fundamental reasons for this. First, $\varphi(m)$ can be incorrect; for example, a faulty node can tell lies about its local inputs. As a human investigator, Alice may be able to recognize such lies (so there is still value in displaying all the available information), but it is not possible to detect them automatically, since nodes cannot observe each other's inputs. Thus, the corresponding vertices do not appear red in G_ν. Note, however, that a faulty node cannot lie arbitrarily; for example, it cannot forge messages from other nodes.

Second, $\varphi(m)$ can be incomplete. For example, if two faulty nodes secretly exchange messages but otherwise act normally, we cannot guarantee that these messages will appear in G_ν because the correct nodes cannot necessarily obtain any evidence about them. We *can*, however, be sure that detectable faults [16] are represented in the graph. Briefly, a detectable fault is one that directly or indirectly affects a correct node through a message, or a chain of messages. Recall that, in our motivating scenario, we have assumed that Alice has observed some symptom of the fault; any fault of this type is detectable by definition.

Third, faulty nodes can equivocate, i.e., there can be two messages m_1 and m_2 such that $\varphi(m_1)$ is inconsistent with $\varphi(m_2)$. If a correct node encounters both m_1 and m_2, it can detect the inconsistency, but it is not clear which of them (if any) is correct and should appear in G_ν. One approach is to liberally use the color red for each vertex that is involved in an inconsistency. However, this can lead to an excessive amount of red coloring on equivocating nodes, which limits the usefulness of G_ν. Another approach, which we adopt here, is to arbitrarily accept one of the explanations as true, e.g., the one that appears first in ϵ, and to allow black for the corresponding vertices. Alice can influence this choice by reordering the messages in ϵ.

Finally, if φ is evaluated on demand, $\varphi(m)$ can be unavailable. For example, a correct node that is trying to evaluate a provenance query on ϵ might ask the sender of some $m \in \epsilon$ for $\varphi(m)$ but might not receive a response. This situation is ambiguous and does not necessarily indicate a fault – for example, the queried node could be slow, or the response could be delayed in the network – so it is not a good basis on which to color a vertex red. However, the only way to avoid it reliably would be to proactively attach $\varphi(m)$ to every message, which would be prohibitively expensive. Instead, SNP uses a third color (yellow) for vertices whose color is not yet known. Yellow vertices turn black or red when the response arrives. If a vertex v remains yellow, this is a sign that $\textsc{host}(v)$ is refusing to respond and is therefore faulty.

4.3 Definition: SNP

We say that an approximation G_ν of G is *monotonic* if $G_\nu(\epsilon)$ is a subgraph of $G_\nu(\epsilon+\epsilon')$ for additional evidence ϵ'. This is an important property because it prevents G_ν from changing fundamentally once additional evidence becomes available, which could invalidate responses to earlier queries.

We define *secure network provenance (SNP)* to be a monotonic approximation G_ν of a provenance graph G that has the following two properties in an untrusted setting. G_ν is *accurate* if it faithfully reproduces all the vertices on correct nodes; in other words, if a vertex v on a correct node appears in $G_\nu(\epsilon)$ then v must also exist in G, be colored black, and have the same predecessors and successors. G_ν is *complete* if, given sufficient evidence ϵ from the correct nodes, a) each vertex in G on a correct node also appears in $G_\nu(\epsilon)$, and b) for each detectably faulty node, $G_\nu(\epsilon)$ contains at least one red or yellow vertex.

We also define a primitive called MICROQUERY that can be used to navigate a SNP graph.[1] MICROQUERY has two arguments: a vertex v, and evidence ϵ such that $v \in G_\nu(\epsilon)$. MICROQUERY returns one or two *color notifications* of the form $\textsc{black}(v)$, $\textsc{yellow}(v)$, or $\textsc{red}(v)$. If two notifications are returned, the first one must be $\textsc{yellow}(v)$. MICROQUERY can also return two sets P_v and S_v that contain the predecessors and successors of v in $G_\nu(\epsilon)$, respectively. Each set consists of elements (v_i, ϵ_i), where ϵ_i is additional evidence such that v_i and the edge between v_i and v appear

[1] MICROQUERY returns a single vertex; provenance queries must invoke it repeatedly to explore G_ν. Hence the name.

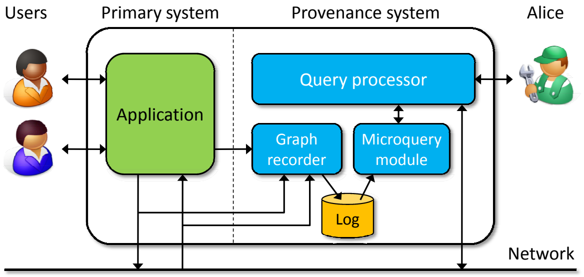


Figure 3: Architecture of a single SNooPy node

in $G_\nu(\epsilon + \epsilon_i)$; this makes it possible to explore all of G_ν by invoking MICROQUERY recursively. We also require that MICROQUERY preserve accuracy, that is, if HOST(v) is correct, it must return BLACK(v), as well as P_v and S_v.

4.4 Discussion

MICROQUERY is sufficient to achieve the goals we stated in Section 2. Any system behavior that Alice can observe (such as derivations, messages, or extant tuples) corresponds to some vertex v in the provenance graph. Alice can then recursively invoke MICROQUERY to learn the causes or effects of v. To learn the causes of v, Alice can start at v and navigate the graph backwards until she arrives at the legitimate root causes (i.e., base tuples) or at some vertex that is colored red. To learn the effects of v, Alice can navigate the graph in the forward direction. The completeness of SNP ensures that, when a detectable fault has occurred, even an adversary cannot prevent Alice from discovering it. The accuracy of SNP ensures that the adversary cannot cause Alice to believe that a correct node is faulty.

Note that, if v is a vertex on a faulty node, it is possible that MICROQUERY returns only YELLOW(v), and nothing else. This is a consequence of the final limitation from Section 4.2, and it can prevent Alice from identifying *all* faulty nodes, since she may not be able to navigate 'past' a yellow vertex. However, Alice can still discover that a fault exists, and she can identify at least one faulty or misbehaving node. At worst, this provides a starting point for a more detailed investigation by supplying evidence against the faulty node. If the faulty node is able to be repaired and its prior observable actions can be verified to conform to its expected behavior, then the node can be recolored black, and subsequent MICROQUERYs will identify whether faults exist(ed) on other nodes.

5. THE SNOOPY SYSTEM

We next present the design of SNooPy, a system that implements secure network provenance for the provenance graph G that was defined earlier in Section 3.

5.1 Architecture

SNooPy consists of three major building blocks: a *graph recorder*, a *microquery module*, and a *query processor* (Figure 3). The graph recorder extracts provenance information from the actions of the primary system (Section 5.3) and stores it in a tamper-evident log (Section 5.4). The microquery module (Section 5.5) uses the information in this log to implement MICROQUERY; it uses authenticators as a specific form of evidence.

The query processor accepts higher-level (macro) queries, such as simple provenance queries, but also causal, historical, or dynamic queries, and answers them by repeatedly invoking MICROQUERY to retrieve the relevant part of the provenance graph. In some primary systems, this graph can be very large; therefore, queries can be parametrized with a *scope* k, which causes the query processor to return only vertices that are within distance k of the queried vertex. For a discussion of scope in an actual usage scenario, see Section 7.3.

5.2 Assumptions and requirements

SNooPy makes the following assumptions:

1. A message sent from one correct node to another is eventually received, if retransmitted sufficiently often;
2. Each node i has a certificate that securely binds a keypair to the node's identity;
3. Nodes have access to a cryptographic hash function, and the signature of a correct node cannot be forged;
4. In the absence of an attack, messages are typically received within at most time T_{prop};
5. Each node has a local clock, and clocks are synchronized to within Δ_{clock};
6. Apart from the 'maybe' rules, the computation on each node is deterministic; and
7. Queriers are allowed to see any vertex and any edge in the provenance graph.

The first three assumptions are needed for the tamper-evident log. Assumption 2 prevents faulty nodes from changing their identity and from creating fictitious nodes; it could be satisfied by installing each node with a certificate that is signed by an offline CA. Assumption 3 is commonly assumed to hold for algorithms like RSA and SHA-1. The next two assumptions are for simplicity; there are ways to build tamper-evident logs without them [17]. Both T_{prop} and Δ_{clock} can be large, e.g., on the order of seconds. The sixth assumption is needed to efficiently store and verify the provenance graph; it is also required for certain BFT systems [6], and it can be enforced for different types of applications [17], including legacy binaries [13]. The final assumption was already discussed in Section 2.3.

5.3 Extracting provenance

To generate the provenance graph, SNooPy must extract information about events from the application to which it is applied. Provenance extraction (or the more general problem of correlating changes to network state based on incoming/outgoing messages) is an ongoing area of active research [29, 30] that is largely orthogonal to the main focus of this paper. In SNooPy, we have found the following three techniques useful for extracting provenance for the target applications that we have examined:

Method #1: Inferred provenance. SNooPy can infer provenance by transparently tracking data dependencies as inputs flow through the system. Inferred provenance can be applied when the dependencies are already explicitly captured in the programming language. We have applied this method to a version of the Chord DHT written in a declarative language (Section 6.1).

Method #2: Reported provenance. Following the approach from [29], applications can explicitly call methods in

SNooPy to report data dependencies. This requires modifications to the source code; also, key parts of the application must be deterministic to enable the querier to verify that provenance was reported correctly. We have applied this method to the Hadoop MapReduce system (Section 6.2).

Method #3: External specification. When black-box applications cannot use either of the previous two approaches, SNooPy can rely on an external specification of how the application's outputs are derived from its inputs. SNooPy can then generate the provenance graph by observing the inputs and outputs. We have applied this method to the Quagga BGP daemon (Section 6.3).

5.4 Graph recorder

The graph recorder stores the extracted provenance information securely at runtime, so that it can later be used by the microquery module when a query is issued.

Recall from Section 3 that our provenance graph $G = (V, E)$ is designed so that each vertex $v \in V$ can be attributed to a specific node $\textsc{host}(v)$. Thus, we can partition the graph so that each $v \in V$ is stored on $\textsc{host}(v)$. To ensure accuracy, we must additionally keep evidence for each cross-node edge, i.e., $(v_1, v_2) \in E$ with $\textsc{host}(v_1) \neq \textsc{host}(v_2)$. Specifically, $\textsc{host}(v_1)$ must be able to prove that $\textsc{host}(v_2)$ has committed to v_2, and vice versa, so that each node can prove that its own vertex is legitimate, even if the other node is compromised. Finally, according to assumption 6, each node's subgraph of G is completely determined by its inputs, its outputs, and the behavior of its 'maybe' rules; hence, it is sufficient to store messages, changes to base tuples, and any (un)derivations that directly involve a 'maybe' rule. When necessary, the microquery module can reconstruct the node's subgraph of G from this information.

In the following, we will write $\sigma_i(x)$ to indicate a signature on x with i's private key, and $\pi_i(x, y)$ to indicate a check whether x is a valid signature on y with i's private key. $H(\cdot)$ stands for the hash function, and $||$ for concatenation.

Logs and authenticator sets: SNooPy's log is a simplified version of the log from PeerReview [17]. The log λ_i of a node i consists of entries of the form $e_k := (t_k, y_k, c_k)$, where t_k is a timestamp, y_k is an entry type, and c_k is some type-specific content. There are five types of entries: SND and RCV record messages, ACK records acknowledgments, and INS and DEL record insertions and deletions of base tuples and, where applicable, tuples derived from 'maybe' rules. Note that log entries are different from vertex types. Each entry is associated with a hash value $h_k = H(h_{k-1} \,||\, t_k \,||\, y_k \,||\, c_k)$ with $h_0 := 0$. Together, the h_k form a hash chain. A node i can issue an *authenticator* $a_k := (t_k, h_k, \sigma_i(t_k \,||\, h_k))$, which is a signed commitment that e_k (and, through the hash chain, $e_1, \ldots, e_{k-1}$) must exist in λ_i. Each node i stores the authenticators it receives from another node j in its *authenticator set* $U_{i,j}$.

Commitment: When a node i needs to send a message m ($+\tau$ or $-\tau$) to another node j, it first appends a new entry $e_x := (t_x, \textsc{snd}, (m, j))$ to its local log. Then it sends $(m, h_{x-1}, t_x, \sigma_i(t_x \,||\, h_x))$ to j. When a node j receives a message (m, a, b, c), j calculates $h'_x := H(a \,||\, b \,||\, \textsc{snd} \,||\, (m, j))$ and then checks whether the authenticator is properly signed, i.e., $\pi_i(c, (b \,||\, h'_x))$, and whether t_x is within $\Delta_{\text{clock}}+\text{T}_{\text{prop}}$ of its local time. If not, j discards the message. Otherwise, j adds (t_x, h'_x, c) to its authenticator set $U_{j,i}$, appends an entry $e_y := (k, \textsc{rcv}, (m, i, a, b, c))$ to λ_j, and sends $(\textsc{ack}, t_x, h_{y-1}, t_y, \sigma_j(t_y \,||\, h_y))$ back to i.

Once i receives $(\textsc{ack}, a, b, c, d)$ from j, it first checks its log to see whether there is an entry $e_x = (a, \textsc{snd}, (m, j))$ in its log that has not been acknowledged yet. If not, it discards the message. i then calculates $h'_y := H(b \,||\, c \,||\, \textsc{rcv} \,||\, (m, i, h_{x-1}, t_x,\, \sigma_i(t_x \,||\, h_x)))$, and checks $\pi_j(d, (c \,||\, h'_y))$ and t_y is within $\Delta_{\text{clock}}+\text{T}_{\text{prop}}$ of its local time. If not, i discards the message. Otherwise, i adds (c, h'_y, d) to its authenticator set $U_{i,j}$ and appends an entry $e_z := (t, \textsc{ack}, a, b, c, d)$ to its log.

If i does not receive a valid acknowledgment within $2{\cdot}\text{T}_{\text{prop}}$, it immediately notifies the maintainer of the distributed system. Any such notification is a clear indication of a fault: at least one of i, j, or the connection between them must be faulty. Once the maintainer acknowledges the notification, the problem is known and can be ignored for the purposes of forensics. However, if the maintainer has *not* received a notification and a query later uncovers a SND without a matching ACK, SNooPy can color the corresponding SEND vertex red because the sender is clearly faulty. Without the notification mechanism, this situation would be ambiguous and could not be reliably attributed to i or j.

Retrieval: The graph recorder implements a primitive $\textsc{retrieve}(v, a^i_k)$ which, when invoked on $i := \textsc{host}(v)$ with a vertex v and an authenticator a^i_k of i, returns the prefix[2] of the log in which v was generated. In essence, RETRIEVE implements the function φ from Section 4 but evaluates it on demand. Typically, the prefix RETRIEVE returns is the prefix authenticated by a^i_k, but if v is an EXIST or BELIEVE vertex that exists at e_k, the prefix is extended to either a) the point where v ceases to exist, or b) the current time. (The special case is necessary because an existing or believed tuple can be involved in further derivations between e_k and the time it disappears, so its vertex may acquire additional outbound edges.) If the prefix extends beyond e_k, i must also return a new authenticator that covers the entire prefix. A correct node can always comply with such a request.

5.5 Microquery module

The microquery module implements $\textsc{microquery}(v, \epsilon)$. At a high level, this works by 1) using ϵ to retrieve a log prefix from $\textsc{host}(v)$, 2) replaying the log to regenerate $\textsc{host}(v)$'s partition of the provenance graph G, and 3) checking whether v exists in it. If v exists and was derived correctly, its predecessors and successors are returned, and v is colored black; otherwise v is colored red.

More formally, the evidence for a vertex v is an authenticator from $\textsc{host}(v)$ that covers a log prefix in which v existed. When $\textsc{microquery}(v, \epsilon)$ is invoked on a node i, i first outputs $\textsc{yellow}(v)$ to indicate that v's real color is not yet known, and then invokes $\textsc{retrieve}(v, \epsilon)$ on $j := \textsc{host}(v)$. If j returns a log prefix that matches ϵ, i replays the prefix to regenerate j's partial provenance subgraph $G_\nu(\epsilon) \,|\, j$. This is possible because we have assumed that the computation is deterministic. If $G_\nu(\epsilon) \,|\, j$ does not contain v or replay fails (i.e., the sent messages do not match the SEND entries in the log, a SEND does not have a matching ACK, or the authenticators in the RECV and ACK entries do not satisfy the conditions from Section 5.4), i outputs $\textsc{red}(v)$; otherwise it outputs $\textsc{black}(v)$ and returns the predecessors

[2] In practice, SNooPy usually does not return an entire prefix; see Section 5.6 for a list of optimizations.

and successors of v in $G_\nu(\epsilon)$. The additional evidence that is returned for a SEND predecessor and a RECEIVE successor consists of the authenticator from the RCV and ACK entries, respectively; the additional evidence for all other vertices is the authenticator returned by RETRIEVE, if any.

Consistency check: As described so far, the algorithm colors a vertex v red when HOST(v) does not have a correct 'explanation' (in the form of a log prefix), and it colors v yellow if HOST(v) does not return any explanation at all. The only remaining case is the one in which v's explanation is inconsistent with the explanation for one of its other vertices. To detect this, i performs the following check: it determines the interval I during which v existed during replay, and asks all nodes with which j could have communicated during I (or simply all other nodes) to return any authenticators that were a) signed by j, and b) have timestamps in I. If such authenticators are returned, i checks whether they are consistent with the log prefix it has retrieved earlier; if not, i outputs RED(v).

5.6 Optimizations

As described so far, each SNooPy node cryptographically signs every single message and keeps its entire log forever, and each microquery retrieves and replays an entire log prefix. Most of the corresponding overhead can be avoided with a few simple optimizations. First, nodes can periodically record a checkpoint of their state in the log, which must include a) all currently extant or believed tuples and b) for each tuple, the time when it appeared. Thus, it is sufficient for MICROQUERY(v, ϵ) to retrieve the log segment that starts at the last checkpoint before v appeared, and start replay from there. Note that this does not affect correctness because, if a faulty node adds a nonexistent tuple τ to its checkpoint, this will be discovered when the corresponding EXIST or BELIEVE vertex is queried, since replay will then begin before the checkpoint and end after it. If the node omits an extant or believed tuple that affects a queried tuple, this will cause replay to fail.

Second, nodes can be required to keep only the log segment that covers the most recent T_{hist} hours in order to decrease storage costs. To speed up queries, the querier can cache previously retrieved log segments, authenticators, and even previously regenerated provenance graphs. As we show in Section 7, this reduces the overhead to a practical level.

Third, the overhead of the commitment protocol can be reduced by sending messages in batches. This can be done using a variant of Nagle's algorithm that was previously used in NetReview [14]: each outgoing message is delayed by a short time T_{batch}, and then processed together with any other messages that may have been sent to the same destination within this time window. Thus, the rate of signature generations/verifications is limited to $1/\text{T}_{\text{batch}}$ per destination, regardless of the number of messages. The cost is an increase in message latency by up to T_{batch}.

5.7 Correctness

Next, we argue that, given our assumptions from Section 5.2, SNooPy provides secure network provenance as defined in Section 4.3—that is, monotonicity, accuracy, and completeness. Here we present only informal theorems and proof sketches; the formal theorems and the proofs can be found in Appendix C.

Theorem 4 SNooPy *is monotonic: if ϵ is a set of valid authenticators and a_k^i a valid authenticator, $G_\nu(\epsilon)$ is a subgraph of $G_\nu(\epsilon + a_k^i)$.*

Proof sketch: There are four cases we must consider. First, the new authenticator a_k^i can be the first authenticator from node i that the querying node has seen. In this case, the querying node will RETRIEVE the corresponding log segment, replay it, and add the resulting vertices to G_ν. Since the graph construction is compositional, this can only add to the graph, and the claim holds. Second, a can belong to a log segment SNooPy has previously retrieved; in this case, G_ν already contains the corresponding vertices and remains unchanged. Third, a can correspond to an extension of an existing log segment. In this case, the additional events are replayed and the corresponding vertices added, and the claim follows because the graph construction is compositional and incremental. Finally, a's log segment can be inconsistent with an existing segment; in this case, the consistency check will add a red SEND vertex to G_ν. □

Theorem 5 SNooPy *is accurate: any vertex v on a correct node that appears in $G_\nu(\epsilon)$ must a) also appear in G, with the same predecessors and successors, and b) be colored black.*

Proof sketch: Claim a) follows fairly directly from the fact that i := HOST(v) is correct and will cooperate with the querier. In particular, i will return the relevant segment of its log, and since the graph construction is deterministic, the querier's replay of this log will faithfully reproduce a subgraph of G that contains v. Any predecessors or successors v' of v with HOST(v') = i can be taken from this subgraph. This leaves the case where HOST(v') $\neq v$. If v' is a predecessor, then it must be a SEND vertex, and its existence can be proven with the authenticator from the corresponding SND entry in λ. Similarly, if v' is a successor, then it must be a RECV vertex, and the evidence is the authenticator in the corresponding ACK entry in λ.

Now consider claim b). Like all vertices, v is initially yellow, but it must turn red or black as soon as i := HOST(v) responds to the querier's invocation of RETRIEVE, which will happen eventually because i is correct. However, v can only turn red for a limited number of reasons—e.g., because replay fails, or because i is found to have tampered with its log—but each of these is related to some form of misbehavior and cannot have occurred because i is correct. Thus, since v cannot turn red and cannot remain yellow, it must eventually turn (and remain) black. □

Theorem 6 SNooPy *is complete: given sufficient evidence ϵ from the correct nodes, a) each vertex in G on a correct node also appears in $G_\nu(\epsilon)$, and b) when some node is detectably faulty, recursive invocations of* MICROQUERY *will eventually yield a red or yellow vertex on a faulty node.*

Proof sketch: Claim a) follows if we simply choose ϵ to include the most recent authenticator from each correct node, which the querying node can easily obtain. Regarding claim b), the definition of a detectable fault implies the existence of a chain of causally related messages such that the fault is apparent from the first message and the last message m is received by a correct node j. We can choose v' to be the RECV vertex that represents m's arrival. Since causal relationships correspond to edges in G, G_ν must contain a path $v' \rightarrow^* v$. By recursively invoking MICROQUERY on v' and its predecessors, we retrieve a subgraph of G_ν that contains this path, so the vertices on the path are queried in turn. Now consider some vertex v'' along the path. When v'' is queried, we either obtain the next vertex on the path, along with valid evidence, or v'' must turn red or yellow. Thus, either this color appears before we reach v, or we eventually obtain evidence of v. □

5.8 Limitations

By design, SNooPy is a forensic system; it cannot actively detect faults, but rather relies on a human operator to spot the initial symptom of an attack, which can then be investigated using SNooPy. Investigations are limited to the part of the system that is being monitored by SNooPy. We do not currently have a solution for partial deployments, although it may be possible to use the approach adopted by NetReview [14] at the expense of slightly weaker guarantees. SNooPy also does not have any built-in redundancy; if the adversary sacrifices one of his nodes and destroys all the provenance state on it, some parts of the provenance graph may no longer be reachable via queries (though any disconnection points will be marked yellow in the responses). This could be mitigated by replicating each log on some other nodes, although, under our threat model, the problem cannot be avoided entirely because we have assumed that any set of nodes—and thus any replica set we may choose—could be compromised by the adversary. Finally, SNooPy does not provide negative provenance, i.e., it can only explain the existence of a tuple (or its appearance or disappearance), but not its absence. Negative provenance is known to be a very difficult problem that is actively being researched in the database community [28]. We expect that SNooPy can be enhanced to support negative provenance by incorporating recent results from this community.

5.9 Prototype implementation

We have built a SNooPy prototype based on components from ExSPAN [50] and PeerReview [17], with several modifications. We completely redesigned ExSPAN's provenance graph according to Section 3, added support for constraints and 'maybe' rules, and implemented the graph recorder and the microquery module. Unlike ExSPAN, the provenance graph is not maintained at runtime; rather, the prototype records just enough information to reconstruct relevant parts of the graph on demand when a query is issued. This is done using deterministic replay, but with additional instrumentation to capture provenance. Since auditing in SNooPy is driven by the forensic investigator, PeerReview's witnesses are not required, so we disabled this feature. It would not be difficult to connect the prototype to a visualizer for provenance graphs, e.g., VisTrails [45].

Macroqueries are currently expressed in *Distributed Datalog* (*DDlog*), a distributed query language for maintaining and querying network provenance graphs. All three methods from Section 5.3 for extracting provenance are supported: since the prototype is internally based on *DDlog*, it can directly infer provenance from any *DDlog* program, but it also contains hooks for reporting provenance, as well as an API for application-specific proxies.

6. APPLICATIONS

To demonstrate that SNooPy is practical, we have applied our prototype implementation to three existing applications, using a different provenance extraction method each time.

6.1 Application #1: Chord

To test SNooPy's support for native *DDlog* programs, we applied it to a declarative implementation [26] of the Chord distributed hash table that uses RapidNet [37]. There are several known attacks against DHTs, so this seems like an attractive test case for a forensic system. Since ExSPAN can automatically transform any *DDlog* program into an equivalent one that automatically reports provenance, and since RapidNet is already deterministic, no modifications were required to the Chord source code.

6.2 Application #2: Hadoop MapReduce

To test SNooPy's support for reported provenance, we applied it to Hadoop MapReduce [12]. We manually instrumented Hadoop to report provenance to SNooPy at the level of individual key-value pairs.

Our prototype considers input files to be base tuples. The provenance of an intermediate key-value pair consists of the arguments of the corresponding `map` invocation, and the provenance of an output consists of the arguments of the corresponding `reduce` invocation. The set of intermediate key-value pairs sent from a map task to a reduce task constitutes a message that must be logged; thus, if there are m map tasks and r reduce tasks, our prototype sends up to $2mr$ messages (a request and a response for each pair). To avoid duplication of the large data files, we apply a trivial optimization: rather than copying the files in their entirety into the log, we log their hash values, which is sufficient to authenticate them later during replay. Since we are mainly interested in tracking the provenance of key-value pairs, we treat inputs from the JobTracker as base tuples. It would not be difficult to extend our prototype to the JobTracker as well.

Individual `map` and `reduce` tasks are already deterministic in Hadoop, so replay required no special modifications. We did, however, add code to replay `map` and `reduce` tasks separately, as well as a switch for enabling provenance reporting (recall that this is only needed during replay). More specifically, we assign a unique identifier (UID) [19] to each of the input, output and intermediate tuples, based on its content and execution context (which indicates, for example, a tuple τ is an input of `map` task m). The Hadoop implementation is instrumented to automatically track *cross-stage* causalities. This is achieved by adding edges between corresponding vertices when tuples are communicated across stages (e.g. from a `map` output file to a reducer). For the causalities *within* a stage, users need to *report* them using a provided API, which takes as arguments the UID of the output tuple, the UIDs of the input tuples that contribute to the output, and the execution context. The reported provenance information is then passed to and maintained in the graph recorder.

Altogether, we added or modified less than 100 lines of Java code in Hadoop itself, and we added another 550 lines for the interface to SNooPy.

6.3 Application #3: Quagga

To test SNooPy's support for application-specific proxies, we applied it to the Quagga BGP daemon. BGP interdomain routing is plagued by a variety of attacks and malfunctions [32], so a secure provenance system seems useful for diagnostics and forensics. SNooPy could complement secure routing protocols such as S-BGP [40]: it cannot actively prevent routing problems from manifesting themselves, but it can investigate a wider range of problems, including route equivocation (i.e., sending conflicting route announcements to different neighbors), replaying of stale routes, and failure to withdraw a route, which are not addressed by S-BGP.

Rather than instrumenting Quagga for provenance and deterministic replay, we treated it as a 'black box' and implemented a small proxy that a) transparently intercepts Quagga's BGP messages and converts them into SNooPy tuples, and b) converts incoming tuples back to BGP messages. The proxy uses a small *DDlog* specification of only four rules. The first rule specifies how announcements propagate between networks, and the next two express the constraint that a network can export at most one route to each prefix at any given time, as required by BGP. The fourth rule is a 'maybe' rule (Section 3.4); it stipulates that each route must either be originated by the network itself, or extend the path of a route that was previously advertised to it. Due to the 'maybe' rule, we did not need to model the details of Quagga's routing policy (which may be confidential); rather, the proxy can infer the essential dependencies between routes from the incoming and outgoing BGP messages it observes.

In addition to the four rules, we wrote 626 lines of code for the proxy. Much of this code is generic and could be reused for other black-box applications. We did not modify any code in Quagga.

6.4 Summary

Our three application prototypes demonstrate that SNooPy can be applied to different types of applications with relatively little effort. Our prototypes cover all three provenance extraction methods described in Section 5.3. Moreover, the three applications generate different amounts of communication, process different amounts of data, have different scalability requirements, etc., so they enable us to evaluate SNooPy across a range of scenarios.

7. EVALUATION

In this section, we evaluate SNooPy using our three applications in five different scenarios. Since we have already proven that SNooPy correctly provides secure network provenance, we focus mostly on overheads and performance. Specifically, our goal is to answer the following high-level questions: a) can SNooPy answer useful forensic queries? b) how much overhead does SNooPy incur at runtime? and c) how expensive is it to ask a query?

7.1 Experimental setup

We examine SNooPy's performance across five application configurations: a Quagga routing daemon (version 0.99.16) deployment, two Chord installations (derived from RapidNet [37] v0.3), and two Hadoop clusters (version 0.20.2).

Our **Quagga** experiment is modeled after the setup used for NetReview [14]. We instantiated 35 unmodified Quagga daemons, each with the SNooPy proxy from Section 6.3, on an Intel machine running Linux 2.6. The daemons formed a topology of 10 ASes with a mix of tier-1 and small stub ASes, and both customer/provider and peering relationships. The internal topology was a full iBGP mesh. To ensure that both the BGP traffic and the routing table sizes were realistic, we injected approximately 15,000 updates from a RouteViews [39] trace. The length of the trace, and thus the duration of the experiment, was 15 minutes. In all experiments, each node was configured with a 1,024-bit RSA key.

We evaluated the Chord prototype (Section 6.1) in two different configurations: **Chord-Small** contains 50 nodes and **Chord-Large** contains 250 nodes. The experiments were performed in simulation, with stabilization occurring every 50 seconds, optimized finger fixing every 50 seconds, and keep-alive messages every 10 seconds. Each simulation ran for 15 minutes of simulated time.

In the **Hadoop-Small** experiment, we ran the prototype described in Section 6.2 on 20 `c1.medium` instances on Amazon EC2 (in the `us-east-1c` region). The program we used (WordCount) counts the number of occurrences of each word in a 1.2 GB Wikipedia subgraph from WebBase [46]. We used 20 mappers and 10 reducers; the total runtime was about 79 seconds. Our final experiment, **Hadoop-Large**, used 20 c1.medium instances with 165 mappers, 10 reducers, and a 10.3 GB data set that consisted of the same Wikipedia data plus the 12/2010 Newspapers crawl from WebBase [46]; the runtime for this was about 255 seconds.

Quagga, Chord, and Hadoop have different characteristics that enable us to study SNooPy under varying conditions. For instance, Quagga and Chord have small messages compared to Hadoop, while Quagga has a large number of messages. In terms of rate of system change, Quagga has the highest, with approximately 1,350 route updates per minute. In all experiments, the actual replay during query evaluation was carried out on an Intel 2.66GHz machine running Linux with 8GB of memory.

7.2 Example queries

To evaluate SNooPy's ability to perform a variety of forensic tasks as well as to measure its query performance, we tested SNooPy using the following provenance queries, each of which is motivated by a problem or an attack that has been previously reported in the literature:

Quagga-Disappear is a dynamic query that asks why an entry from a routing table has disappeared. In our scenario, the cause is the appearance of an alternative route in another AS j, which replaces the original route in j but, unlike the original route, is filtered out by j's export policy. This is modeled after a query motivated in Teixeira *et al.* [44]; note that, unlike Omni, SNooPy works even when nodes are compromised. **Quagga-BadGadget** query asks for the provenance of a 'fluttering' route; the cause is an instance of BadGadget [11], a type of BGP configuration problem.

Chord-Lookup is a historical query that asks which nodes and finger entries were involved in a given DHT lookup, and **Chord-Finger** returns the provenance of a given finger table entry. Together, these two queries can detect an Eclipse attack [42], in which the attacker gains control over a large fraction of the neighbors of a correct node, and is then able to drop or reroute messages to this node and prevent correct overlay operation.

Hadoop-Squirrel asks for the provenance of a given key-value pair in the output; for example, if WordCount produces the (unlikely) output `(squirrel,10000)` to indicate that the word 'squirrel' appeared 10,000 times in the input, this could be due to a faulty or compromised mapper. Such queries are useful to investigate computation results on outsourced Cloud databases [34].

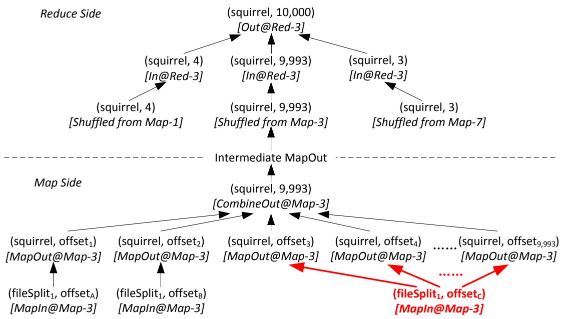


Figure 4: Example result of the Hadoop-Squirrel macroquery (in a simplified notation).

7.3 Usability

In addition to the formal guarantees in Section 4, we also need to demonstrate that SNooPy is a useful forensic tool *in practice*. For this purpose, we executed each of the above queries twice – once on a correct system and once on a system into which we had injected a corresponding fault. Specifically, we created an instance of BadGadget in our Quagga setup, we modified a Chord node to mount an Eclipse attack by always returning its own ID in response to lookups, and we tampered with a Hadoop map worker to make it return inaccurate results. For all queries, SNooPy clearly identified the source of the fault.

To illustrate this, we examine one specific example in more detail, Figure 4 shows the output of the Hadoop-Squirrel macroquery in which one of the mappers (Map-3) was configured to misbehave: in addition to emitting `(word, offset)` tuples for each word in the text, it injected 9,991 additional `(squirrel, offset)` tuples (shown in red). A forensic analyst who is suspicious of the enormous prevalence of squirrels in this dataset can use SNooPy to query the provenance of the `(squirrel, 10000)` output tuple. To answer this query, SNooPy selectively reconstructs the provenance subgraph of the corresponding reduce task by issuing a series of microqueries, one for each immediate predecessor of the `(squirrel, 10000)` tuple, and then assembles the results into a response to the analyst's macroquery. Seeing that one mapper output 9,993 squirrels while the others only reported 3 or 4, she can 'zoom in' further by requesting the provenance of the `(squirrel, 9993)` tuple, at which point SNooPy reconstructs the provenance subgraph of the corresponding map task. This reveals two legitimate occurrences and lots of additional bogus tuples, which are colored red.

Once the faulty tuples are identified, SNooPy can be used to determine their effects on the rest of the system, e.g., to identify other outputs that may have been affected by key-value pairs from the corrupted `map` worker.

In this example, the analyst repeatedly issues queries with a small scope and inspects the results before deciding which query to issue next. This matches the usage pattern of provenance visualization tools, such as VisTrails [5], which allow the analyst to navigate the provenance graph by expanding and collapsing vertices. The analyst could also use a larger scope directly, but this would cause more subgraphs to be reconstructed, and most of the corresponding work would be wasted because the analyst subsequently decides to investigate a different subtree.

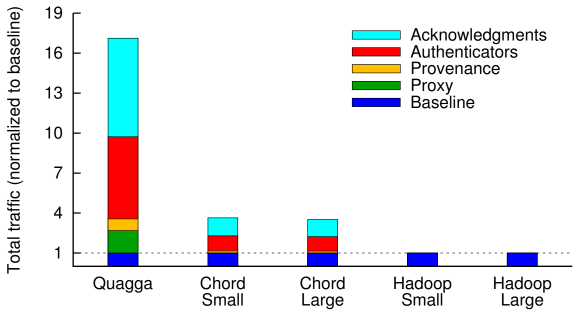


Figure 5: Network traffic with SNooPy, normalized to a baseline system without provenance.

7.4 Network traffic at runtime

SNooPy increases the network traffic of the primary system because messages must contain an authenticator and be acknowledged by the recipient. To quantify this overhead, we ran all five experiments in two configurations. In the *baseline* configuration, we ran the original Hadoop, Quagga, or declarative Chord in RapidNet with no support for provenance. In the SNooPy-enabled prototype, we measured the additional communication overhead that SNooPy adds to the baseline, broken down by cause, i.e., authenticators, acknowledgments, provenance, and proxy.

Figure 5 shows the SNooPy results, normalized to the baseline results. The overhead ranges between a factor of 16.1 for Quagga and 0.2% for Hadoop. The differences are large because SNooPy adds a fixed number of bytes for each message – 22 bytes for a timestamp and a reference count, 156 bytes for an authenticator, and 187 bytes for an acknowledgment. Since the average message size is small for Quagga (68 bytes) and very large for Hadoop (1.08 MB), the relative overhead for Quagga is higher, although in absolute terms, the Quagga traffic is still low (78.2 Kbps with SNooPy). Chord messages are 145 bytes on average, and hence its overhead factor is in between Quagga and Hadoop.

The relative overhead of the Quagga proxy is high in part because, unlike the original BGP implementation in Quagga, the proxy does not combine BGP announcements and (potentially multiple) withdrawals into a single message. However, the overhead can be reduced by enabling the message batching optimization from Section 5.6. With a window size of T_{batch}= 100 ms, the number of messages decreases by more than 80%, and the normalized overhead drops from 16.1 to 4.8, at the expense of delaying messages by up to T_{batch}.

In summary, SNooPy adds a constant number of bytes to each message. Thus, the absolute overhead depends on how many messages the primary system sends. The relative increase in network traffic depends on the primary system's average message size.

7.5 Storage

Each SNooPy node requires some local storage for the graph recorder's log. Since the microquery module uses deterministic replay to partially reconstruct the provenance graph on demand, we should generally expect the log to be at least as large as a replay log, although SNooPy can sometimes save space by referencing data that is already kept for other

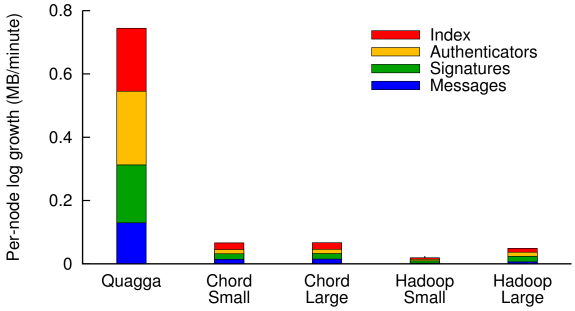


Figure 6: Per-node log growth in SNooPy, excluding checkpoints.

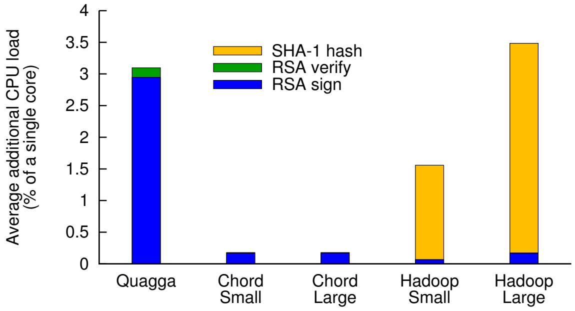


Figure 7: Additional CPU load for generating and verifying signatures, and for hashing.

reasons. To quantify the incremental storage cost, we ran our five experiments in the SNooPy configuration, and we measured the size of the resulting logs.

Figure 6 shows the average amount of log data that each node produced per minute, excluding checkpoints. In absolute terms, the numbers are relatively low; they range from 0.066 MB/min (Chord-Small) to 0.74 MB/min (Quagga). We expect that most forensic queries will be about fairly recent events, e.g., within one week. To store one week's worth of data, each node would need between 7.3 GB (Quagga) and 665 MB (Chord-Small). Note that, in contrast to proactive detection systems like PeerReview [17], this data is merely archived locally at each node and is only sent over the network when a query is issued. Also, it should be possible to combine SNooPy with state-of-the-art replay techniques such as ODR [1], which produce very small logs.

The log contains copies of all received messages (for Hadoop, references to files), authenticators for each sent and received message, and acknowledgments. Thus, log growth depends both on the number of messages and their size distribution. As a result, Figure 6 shows that log growth was fastest for Quagga, given that its baseline system generates the largest number of messages. In the case of Hadoop, our proxy benefits from the fact that Hadoop already retains copies of the input files unless the user explicitly deletes them. Thus, the proxy can save space by merely referencing these files from the log, and the *incremental* storage cost is extremely low (less than 0.1 MB/minute). The size of the input files was 1.2 GB for Small and 10.3 GB for Large. If these files were not retained by Hadoop, they would have to be copied to the log.

As described in Section 5.6, SNooPy can additionally keep checkpoints of the system state. The size of a typical checkpoint is 25 kB for Chord and 64 MB for Quagga. Since replay starts at checkpoint, more checkpoints result in faster queries but consume more space. For Hadoop, the equivalent of a checkpoint is to keep the intermediate files that are produced by the Map tasks, which requires 207 MB for Small and 682 MB for Large.

7.6 Computation

We next measured the computation cost imposed by SNooPy. We expect the cost to be dominated by signature generation and verification and, in the case of Hadoop, hashing the input and output files (see Section 6.2). To verify this, we used `dstat` to measure the overall CPU utilization of a Quagga node with and without the SNooPy proxy; the log and the checkpoints were written to a RAM disk to isolate computation from I/O overhead. Our results show an average utilization of 5.4% of one core with SNooPy, and 0.9% without. As expected, more than 70% of the overhead can be explained by the cost of signature generation and verification alone (in our setup, 1.3 ms and 66 μs per 1,024-bit signature); the rest is due to the proxy logic.

To get a more detailed breakdown of the crypto overhead in our three applications, we counted the number of crypto operations in each configuration, and we multiplied the counts with the measured cost per operation to estimate the average additional CPU load they cause. As our results in Figure 7 show, the average additional CPU load is below 4% for all three applications. For Quagga and Chord, the increase is dominated by the signatures, of which two are required for each message – one for the authenticator and the other for the acknowledgment. Hadoop sends very few messages (one from each mapper to each reducer) but handles large amounts of data, which for SNooPy must be hashed for commitment. Note that we do not include I/O cost for the hashed data because the data would have been written by the unmodified Hadoop as well; SNooPy merely adds a SHA-1 hash operation, which can be performed on-the-fly as the data is written.

The message batching optimization from Section 5.6 can be used to reduce the CPU load. To evaluate this, we performed an additional experiment with Quagga, and we found that a window size of T_{batch}= 100 ms reduced the total number of signatures by a factor of six. Message batching also prevents the CPU load from spiking during message bursts, since it limits the rate at which signatures are generated to at most $1/T_{batch}$ per destination.

7.7 Query performance

Next, we evaluate how quickly SNooPy can answer queries, and how much data needs to be downloaded. Since the answer depends on the query, we performed several different queries in different systems. For each query, we measured a) how much data (log segments, authenticators, and checkpoints) was downloaded, b) how long it took to verify the log against the authenticators, and c) how much time was needed to replay the log and to extract the relevant provenance subgraph. Figure 8 shows our results. Note that the query turnaround time includes an estimated download time, based on an assumed download speed of 10 Mbps.

The results show that both the query turnaround times and the amount of data downloaded can vary consider-

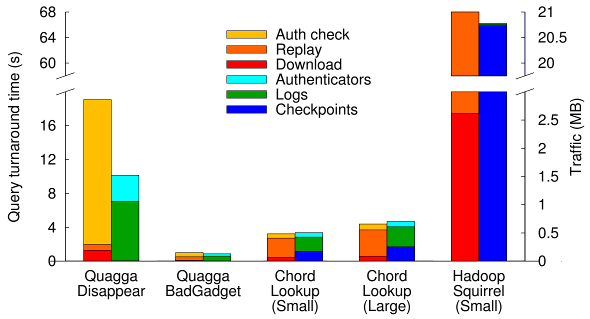


Figure 8: Query turnaround time (left bar) and data downloaded to answer the query (right bar).

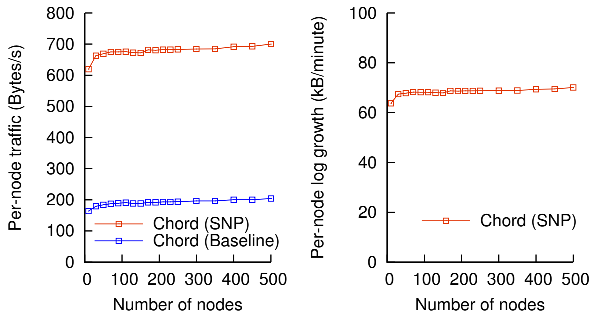


Figure 9: Scalability for Chord: Total traffic (left) and log size (right)

ably with the query. The Chord and Quagga-BadGadget queries were completed in less than five seconds; the Quagga-Disappear query took 19 seconds, of which 14 were spent verifying partial checkpoints using a Merkle Hash Tree; and the Hadoop-Squirrel query required 68 seconds, including 51 for replay. The download varied between 133 kB for Quagga-BadGadget and 20.8 MB for Hadoop-Squirrel. The numbers for Hadoop are larger because our prototype does not create checkpoints *within* map or reduce tasks, and so must replay a node's entire task to reconstruct a vertex on that node. Fine-grained checkpoints could be added but would require more changes to Hadoop. Generally, there is a tradeoff between storage and query performance: finer-grained checkpoints require more storage but reduce the size of the log segments that need to be downloaded and replayed.

In summary, the downloads and query turnaround times vary between queries but generally seem low enough to be practical for interactive forensics.

7.8 Scalability

In our final experiment, we examine how SNooPy's overhead scales with the number of nodes N. We ran our Chord experiment with a range of different system sizes between $N = 10$ and $N = 500$ nodes, and we measured two of the main overheads, traffic and log size, for each N. Figure 9 shows our results, plus the baseline traffic for comparison.

The results show that both overheads increase only slowly with the system size. This is expected because, as discussed in Sections 7.4 and 7.5, the overhead is a function of the number and size of the messages sent. If the per-node traffic of the application did not depend on N, the runtime overhead would not depend on N either; however, recall that Chord's traffic increases with $O(\log N)$, as illustrated here by the baseline traffic results, so the SNooPy overheads in this experiment similarly grow with $O(\log N)$.

Note the contrast to accountability systems like PeerReview [17] where the overhead itself grows with the system size. This is because PeerReview uses witnesses to ensure that each pair of authenticators from a given node is seen by at least one correct node. SNooPy relies on the querier's node for this property (see Section 5.5) and, as a forensic system, it does not audit proactively.

In summary, SNooPy does not reduce the scalability of the primary system; its per-node overheads mainly depend upon the number of messages sent.

7.9 Summary

SNooPy's runtime costs include a fixed-size authenticator and acknowledgment for each message, processing power to generate and verify the corresponding signatures, and storage space for a per-node log with enough information to reconstruct that node's recent execution. Some part of the log needs to be downloaded and replayed when a query is issued. In the three different applications we evaluated, these costs are low enough to be practical. We have also described several example queries that can be used to investigate attacks previously reported in the literature, and we have demonstrated that SNooPy can answer them within a few seconds.

8. RELATED WORK

Debugging and forensics: The main difference between SNP and existing forensic systems is that SNP does not require trust in any components on the compromised nodes. For example, Backtracker [21, 22] and PASS [29] require a trusted kernel, cooperative ReVirt [3] a trusted VMM, and A2M [7] trusted hardware. ForNet [41] and NFA [48] assume a trusted infrastructure and collaboration across domains. Other systems, such as the P2 debugger [43], ExSPAN [50], Magpie [2], D3S [25], QI [33], Friday [9], and Pip [38] are designed to diagnose non-malicious faults, such as bugs or race conditions. When nodes have been compromised by an adversary, these systems can return incorrect results.

Accountability: Systems like PeerReview [17] and NetReview [14] can automatically detect when a node deviates from the algorithm it is expected to run. Unlike SNP, these systems cannot detect problems that arise from interactions between multiple nodes, such as BadGadget [11] in interdomain routing, or problems that are related to inputs or unspecified aspects of the algorithm. Also, accountability systems merely report that a node is faulty, whereas SNP also offers support for diagnosing faults and for assessing their effects on other nodes.

Fault tolerance: An alternative approach to the problem of Byzantine faults is to mask their effects, e.g., using techniques like PBFT [6]. Unlike SNP, these techniques require a high degree of redundancy and a hard bound on the number of faulty nodes, typically one third of the total. The two approaches are largely complementary and could be combined.

Proofs of misbehavior: Many systems that are designed to handle non-crash faults internally use proofs of misbehavior, such as the signed confessions in Ngan *et al.* [31], a set of conflicting tickets in SHARP [8], or the POM message in Zyzzyva [23]. In SNP, any evidence that creates a red vertex in G_ν essentially constitutes a proof of misbehavior, but SNP's evidence is more general because it proves misbehavior with respect to the (arbitrary) primary system, rather than with respect to SNP or its implementation, e.g., SNooPy. Systems like PeerReview [17] can generate protocol-independent evidence as well, but, unlike SNP's evidence, PeerReview's evidence is not diagnostic: it only shows that a node is faulty, but not what went wrong.

Network provenance: Systems like ExSPAN [50] describe the history and derivations of network state that results from the execution of a distributed protocol. SNP extends network provenance to adversarial environments, and enhances the traditional notion of network provenance by adding support for dynamic provenance and historical queries. The support for historical queries includes some features from an earlier workshop paper [49].

Secure provenance: McDaniel *et al.* [27] outlines requirements for secure network provenance, emphasizing the need for provenance to be tamper-proof and non-repudiable. Sprov [18] implements secure chain-structured 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 [36] 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.

9. CONCLUSION

This paper introduces secure network provenance (SNP), a technique for securely constructing network provenance graphs in untrusted environments with Byzantine faults. SNP systems can help forensic analysts by answering questions about the causes and effects of specific system states. Since faulty nodes can tell lies or suppress information, SNP systems cannot always determine the exact provenance of a given system state, but they can approximate it and give strong, provable guarantees on the quality of the approximation.

SNooPy, our implementation of a SNP system, can query not only the provenance of an extant state, but also the provenance of a past state or a state change, which should be useful in a forensic setting. For this, it relies on a novel, SNP-enabled provenance graph that has been augmented with additional vertex types to capture the necessary information. To demonstrate that SNP and SNooPy are general, we have evaluated a SNooPy prototype with three different example applications: the Quagga BGP daemon, a declarative implementation of Chord, and Hadoop MapReduce. Our results show that the costs vary with the application but are low enough to be practical.

Acknowledgments

We thank our shepherd, Petros Maniatis, and the anonymous reviewers for their comments and suggestions. We also thank Joe Hellerstein, Bill Marczak, Clay Shields, and Atul Singh for helpful comments on earlier drafts of this paper. This work was supported by NSF grants IIS-0812270, CNS-0845552, CNS-1040672, CNS-1054229, CNS-1064986, CNS-1065130, AFOSR MURI grant FA9550-08-1-0352, and DARPA SAFER award N66001-11-C-4020. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2004.

[3] M. Basrai and P. M. Chen. Cooperative ReVirt: adapting message logging for intrusion analysis. Technical Report University of Michigan CSE-TR-504-04, Nov 2004.

[4] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *Proc. International Conference on Database Theory (ICDT)*, Jan. 2001.

[5] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo. VisTrails: Visualization meets data management. In *Proc. ACM SIGMOD Conference*, June 2006.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[7] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[8] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[9] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[10] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *Proc. International Conference on Very Large Data Bases (VLDB)*, Sept. 2007.

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

[12] Hadoop. `http://hadoop.apache.org/`.

[13] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proc. USENIX Symposium on Operating System Design and Implementation (OSDI)*, Oct. 2010.

[14] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[15] A. Haeberlen and P. Kuznetsov. The Fault Detection Problem. In *Proc. Intl. Conference on Principles of Distributed Systems (OPODIS)*, Dec. 2009.

[16] A. Haeberlen, P. Kuznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proc. Workshop on Hot Topics in System Dependability (HotDep)*, Nov. 2006.

[17] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

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

[19] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, Jan. 2011.

[20] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proc. 16th USENIX Security Symposium*, Aug 2007.

[21] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems (TOCS)*, 23(1):51–76, 2005.

[22] S. T. King, Z. M. Mao, D. Lucchetti, and P. Chen. Enriching intrusion alerts through multi-host causality. In *Proc. Annual Network and Distributed Systems Security Symposium (NDSS)*, Feb. 2005.

[23] R. Kotla, L. A. M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. on Comp. Syst. (TOCS)*, 27(4), Dec. 2009.

[24] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Prog. Lang. and Systems (TOPLAS)*, 4(3):382–401, 1982.

[25] 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. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[26] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. *CACM*, 2009.

[27] P. McDaniel, K. Butler, S. McLaughlin, R. Sion, E. Zadok, and M. Winslett. Towards a Secure and Efficient System for End-to-End Provenance. In *Proc. USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, Feb. 2010.

[28] A. Meliou, W. Gatterbauer, K. M. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. In *Proc. International Conference on Very Large Data Bases (VLDB)*, Aug. 2011.

[29] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX Annual Technical Conference*, 2006.

[30] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. Annual Network and Distributed Systems Security Symposium (NDSS)*, Feb. 2005.

[31] T.-W. Ngan, D. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2003.

[32] O. Nordstroem and C. Dovrolis. Beware of BGP attacks. *ACM Comp. Comm. Rev. (CCR)*, Apr 2004.

[33] A. J. Oliner and A. Aiken. A query language for understanding component interactions in production systems. In *Proc. International Conference on Supercomputing (ICS)*, June 2010.

[34] H. Pang and K.-L. Tan. Verifying Completeness of Relational Query Answers from Online Servers. *ACM Transactions on Information and System Security (TISSEC)*, 11:5:1–5:50, May 2008.

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

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

[37] RapidNet. `http://netdb.cis.upenn.edu/rapidnet/`.

[38] 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. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[39] RouteViews project. `http://www.routeviews.org/`.

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

[41] K. Shanmugasundaram, N. Memon, A. Savant, and H. Bronnimann. ForNet: A distributed forensics network. In *Proc. Intl. Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS)*, Sept. 2003.

[42] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against the Eclipse attack in overlay networks. In *Proc. ACM SIGOPS European Workshop*, Sept. 2004.

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

[44] R. Teixeira and J. Rexford. A measurement framework for pin-pointing routing changes. In *Proc. SIGCOMM Network Troubleshooting Workshop*, Sep 2004.

[45] Vistrails. `http://www.vistrails.org/`.

[46] The Stanford WebBase Project. `http://diglib.stanford.edu/~testbed/doc2/WebBase/`.

[47] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005.

[48] Y. Xie, V. Sekar, M. Reiter, and H. Zhang. Forensic analysis for epidemic attacks in federated networks. In *Proc. IEEE International Conference on Network Protocols (ICNP)*, Nov. 2006.

[49] W. Zhou, L. Ding, A. Haeberlen, Z. Ives, and B. T. Loo. TAP: Time-aware provenance for distributed systems. In *Proc. USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, June 2011.

[50] 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. ACM SIGMOD Conference*, June 2010.

APPENDIX

A. PRELIMINARIES

We begin with a few additional definitions and clarifications that are not included in the main paper.

A.1 Simplifications

The system model was already described in Section 3.1, but for the purposes of the proof, we simplify this model in two ways: we assume that tuples have unique derivations, and we assume that no constraints or maybe rules are used. Neither of these simplifications reduces generality, for the following reason.

If the system contains a tuple τ with $k > 1$ derivations, we can consider an equivalent system that has k additional tuples $\tau_1, \ldots, \tau_k$, one for each possible derivation, and an extra derivation rule that causes τ to be derived whenever at least one of the τ_i exists.

If the system contains a maybe rule $\tau@\mathtt{n} \xleftarrow{maybe} \tau_1@\mathtt{n}_1 \wedge \ldots \wedge \tau_k@\mathtt{n}_k$, we can consider an equivalent system that contains an extra base tuple $\beta@\mathtt{n}$, in which the maybe rule is replaced with $\tau@\mathtt{n} \leftarrow \tau_1@\mathtt{n}_1 \wedge \ldots \wedge \tau_k@\mathtt{n}_k \wedge \beta@\mathtt{n}$, and in which $\mathtt{n}$ inserts or deletes β whenever it chooses to trigger the maybe rule, or to stop it.

A.2 Definitions

In the following, we will write N to denote the set of nodes, and R to denote the set of rules.

We define a *message* m to be $(+\tau, t)$ or $(-\tau, t)$, where t is the sender's local timestamp, or $\textsc{ack}(m', t')$, where m' is another message $(+\tau, t)$ or $(-\tau, t)$. We say that $\textsc{ack}(m')$ is an *acknowledgment* of the message m'. A message $(+\tau, t)$ sent by a node i indicates that a tuple τ now exists on i; a message $(-\tau, t)$ indicates that τ has ceased to exist on i; and an acknowledgment of m' indicates that m' was received by the sender of the acknowledgment. We assume that messages can be authenticated.

In the following, we will simply write $+\tau$ or $-\tau$ when the timestamp is irrelevant. We write $\textsc{src}(m)$ to denote the node that sent message m, $\textsc{dst}(m)$ to denote the node to which m was sent, and $\textsc{txmit}(m)$ to denote the timestamp t.

In Section 3.1, we have assumed that each node i applies its local rules deterministically, and that this process can be modeled as a state machine A_i that maintains i's tuple set and responds to inputs (incoming messages or base tuple changes) by updating the tuple set and by sending messages to other nodes. We now precisely define the inputs and outputs. A_i accepts the inputs $\textsc{ins}(\beta)$, $\textsc{del}(\beta)$, and $\textsc{rcv}(m)$, where β is a base tuple and m is a message with $\textsc{dst}(m) = i$. A_i produces outputs $\textsc{der}(\tau)$, $\textsc{und}(\tau)$, and $\textsc{snd}(m)$, where τ is a tuple (but not a base tuple) and m is a message with $\textsc{src}(m) = i$.

A.3 Histories and executions

We must have a way to talk about what 'has actually happened' in the primary system by the time a query is issued to the provenance system. For this purpose, we define the concept of a *history*, which describes the behavior of the primary system from the perspective of an omniscient observer that can observe all messages and base tuple insertions/deletions. This will serve as the 'ground truth' against which the responses of the provenance system can be evaluated. The following definitions are based on the ones in [15, 16].

An history h of the primary system is a sequence of *events* $e_k = (t_k, i_k, x_k)$, where $k = 1, 2, 3, \ldots$, $t_k > t_{k-1}$, $i_k \in N$ and x_k is one of the following:

1. $\textsc{snd}(m)$, which indicates that i_k has sent message m;
2. $\textsc{rcv}(m)$, which indicates that i_k has received message m;
3. $\textsc{ins}(\tau)$, which indicates that a base tuple τ has been inserted on i_k; and
4. $\textsc{del}(\tau)$, which indicates that a base tuple τ has been removed from i_k.

Each message can be sent at most once (recall the sequence numbers), and each sent message is received at most once. Base tuples are inserted before they can be removed, and no base tuple is removed more than once. We write $h \,|\, i$ to denote the subsequence of h that consists of all the events associated with i in h. We define an *execution* e of the primary system to be a history in which each sent message is sent exactly once.

We say that a node i is *correct* in an execution e iff $e \,|\, i$ conforms to A_i, i.e., if the following holds for $e \,|\, i$:

- $e|i$ starts with a $\textsc{rcv}$, $\textsc{ins}$ or $\textsc{del}$ event; and
- for any prefix p of $e \,|\, i$ that starts with a $\textsc{rcv}$, $\textsc{ins}$, or $\textsc{del}$ event, we can construct another prefix p' of $e \,|\, i$ by appending 1) $\textsc{snd}(\textsc{ack}(m))$ if p ended with $\textsc{rcv}(m)$, and 2) all the $\textsc{snd}$ outputs that A_i produces when fed with all the inputs in p, but excluding $\textsc{der}$ and $\textsc{und}$ outputs, as well as any outputs that are already in p. If $p' \neq e \,|\, i$, p' must not be followed by a $\textsc{snd}$ event in $e \,|\, i$.

If i is not correct in e, we say that i is *faulty* in e.

B. PROVENANCE GRAPH

In this section, we give a definition of the 'true' provenance graph $G(e)$ for an execution e. Ideally, we would like SNP to answer queries about this graph. However, in an actual distributed system, no single node can know the entire execution e, which includes the actual events on *all* nodes, including the faulty ones. In Appendix C, we will give a definition of another provenance graph $G_\nu(\epsilon)$, which relies on evidence ϵ that is available at a single node (or can be collected by a single node). Since faulty nodes can lie or equivocate, we cannot guarantee that G_ν is always identical to G; instead, our goal will be for G_ν to approximate G as closely as possible.

B.1 The graph construction algorithm

We say that the *provenance graph* $G(h) := (V(h), E(h))$ for a given history h is the graph that is produced by the *graph construction algorithm (GCA)*, which consists of the pseudocode in Tables 10 and 11, as well as the following steps:

1. Initialize V and E as empty sets.
2. If there are no more events, terminate.
3. Take the next event $e_k := (t_k, i_k, x_k)$ and invoke the corresponding HANDLE-EVENT function. If x_k is $\textsc{snd}$, continue with step 2
4. Feed the input that corresponds to e_k to the state machine A_i.

5. For each output A_i produces, invoke the corresponding HANDLE-OUTPUT function, then continue with step 2.

If h is finite, the GCA terminates because a) the pseudocode methods contain only loops that terminate after a finite number of steps, and b) we have assumed that there are no infinite derivations, so the state machines A_i can produce only a finite number of outputs in response to each input.

We also need two additional definitions. First, if h is a history, we say that a vertex $v \in G(h)$ is *stable* iff h contains an event on HOST(v) with a timestamp that is at least $2 \cdot \mathrm{T}_{\mathrm{prop}}$ higher than v's timestamp. Second, recall that each vertex in $V(h)$ has one of three colors, RED, BLACK, or YELLOW; we introduce a total order on these colors, namely RED>BLACK>YELLOW. If c_1 and c_2 are two colors such that $c_1 > c_2$, we say that c_1 *dominates* c_2. In other words, BLACK dominates YELLOW, and RED dominates both other colors.

B.2 Operations on provenance graphs

We now define several operations on provenance graphs, including union, projection, and a subgraph relation. These will be useful for stating the lemmata later in this section.

If $G_1 := (V_1, E_1)$ and $G_2 := (V_2, E_2)$ are two provenance graphs, we define the *union* of the two graphs, written $G_1 \cup^* G_2$, to be the provenance graph $G := (V, E)$ with

- $V := V_1 \cup V_2$, except that whenever V_1 and V_2 contain the same EXIST or BELIEVE vertex with different intervals, V contains only one such vertex, whose interval is the intersection of the two other intervals;
- $E := E_1 \cup E_2$; and
- The color of a vertex $v \in V$ is the dominant color among the color(s) it has in V_1 and V_2.

We say that a provenance graph G_1 is a *subgraph* of another provenance graph G, written $G_1 \subseteq^* G$, iff there exists another provenance graph G_2 such that $G_1 \cup^* G_2 = G$.

We write $G \,|\, i$ to denote the *projection* of G to i, i.e., the subgraph of G that consists of a) all the vertices v in G with HOST(v)=i, and b) any SEND or RECEIVE vertices on other nodes that are connected to these vertices by an edge. The vertices described by a) have the same color in G and $G \,|\, i$; the vertices described by b) are colored yellow in $G \,|\, i$. Note that, for any provenance graph G, $G = \bigcup_{i \in N} G \,|\, i$.

B.3 GCA is monotonic

Our first theorem says that, if we add more events at the end of a history, the provenance graph can grow, but the existing parts of the graph cannot change. This is important because when a query is issued, only a prefix of the execution is known. If the provenance graph could change after future events, it would be very difficult to reliably answer queries.

Theorem 1 *If a history h_1 is a prefix of another history h_2, then $G(h_1) \subseteq^* G(h_2)$, and any stable black vertices in $G(h_1)$ are also black in $G(h_2)$.*

Proof: We must show that a) all vertices and edges present in $G(h_1)$ are also present in $G(h_2)$; b) each EXIST or BELIEVE vertex in $G(h_1)$ either has the same interval in $G(h_2)$ or a smaller one, c) each vertex in $G(h_1)$ has the same color, or a more dominant color, in $G(h_2)$; and d) all black vertices in $G(h_1)$ are also black in $G(h_2)$.

a) follows from the fact that the GCA never removes any vertices or edges. EXIST and BELIEVE vertices can be updated, but only once, by closing their interval in line 19 or 34; therefore, b) also holds.

Next, we observe that, in each instance where the GCA colors a vertex yellow, that vertex has just been created. Thus, color transitions to yellow cannot occur. To prove property c), it remains to be shown that red vertices cannot become black. There are only two points in the code where a vertex is colored black, namely lines 136 and 117. In line 136, the vertex in question is a SEND vertex that was just removed from the `unacked` set, but vertices in `unacked` are always yellow – they may be colored red in lines 41, 46, or 120, but they are removed from the set immediately. In line 117, the vertex in question is a RECEIVE vertex, but it was just removed from the `ackpend` set; vertices in this set can be colored red in line 95, but they are immediately removed from the set. Therefore, property c) holds.

To prove property d), we must show that black vertices cannot become red. The only points in the code where a vertex is colored red are lines 41, 46, 89, 95, and 122. Line 89 can be ignored because ADD-RED-UNLESS-PRESENT only colors vertices it has just created. Line 41 only colors vertices in `pending`, line 46 only vertices in `unacked`, and line 95 only vertices in `ackpend`; vertices in these sets are always yellow. The vertex that is colored in line 122 has just been created. Therefore, claim d) holds. □

B.4 GCA is nontrivial

Many of the later theorems trivially hold for an empty provenance graph. Therefore, it is important to formalize the notion that the GCA returns a graph that is 'useful' in some way. Here, we prove that the EXIST vertices accurately represent the tuples that existed on correct nodes (on faulty nodes, it is not always clear whether a tuple can be said to exist or not, e.g., after an equivocation). We do not prove similar lemmata about the other vertex types, but Section 3.2 should give an intuition of what information the graph contains, and how it could be useful.

Lemma 1 *If h is a history, $i \in N$ is a node that is correct in h, and t is a timestamp, then a) $G(h)$ contains a vertex* EXIST$(i, \tau, [t_1, t_2])$ *with $t_1 \leq t \leq t_2$ iff a tuple τ existed on i at time t, and b) there is at most one such vertex.*

Proof: We begin with a couple of observations. First, EXIST vertices are only added in function APPEAR-LOCAL-TUPLE, in line 10, which is reachable only through HANDLE-EVENT-INS and HANDLE-OUTPUT-DER; specifically, the insertion or derivation of τ at time t_1 will cause a vertex EXIST$(i, \tau, [t_1, \infty))$ to be added to $V(h)$. Second, an EXIST vertex can be updated at most once, in line 19, which closes the interval by updating the second timestamp. This line is reachable only through HANDLE-EVENT-DEL and HANDLE-OUTPUT-UND, i.e., the deletion or underivation of τ at time t_2 will cause the interval in the EXIST vertex that has been created at the preceding insertion (derivation) to be updated from $[t_1, \infty)$ to $[t_1, t_2]$. From this and the fact that the GCA is monotonic (Lemma 1), we can immediately see that the intervals in EXIST vertices for the same tuple cannot overlap, so claim b) holds.

Next, we must prove that 1) the existence of the tuple implies the existence of the vertex, and 2) the existence of the vertex implies the existence of the tuple. We begin with 1), i.e., we assume that τ existed at time t. Now, τ can either be a base tuple or a derived tuple. If it is a base tuple, it must have been inserted at least once before t, and this insertion must be reflected in the history. Let $e_k := (t_k, i, \textsc{ins}(\tau))$ be the event that corresponds to the last insertion before t, i.e., $t_k < t$. When e_k is reached, HANDLE-EVENT-INS will be invoked, which, as we have seen earlier, causes a suitable EXIST vertex to be added. A subsequent deletion can still update the second timestamp in this vertex, but, since τ existed at time t, this deletion can only happen after t, so the vertex still meets our criteria.

Now consider the case where τ is a derived tuple. Recall that the state machine A_i is deterministic. Since the history contains A_i's inputs and the GCA replays these exact same inputs to A_i in the same order, A_i goes through the same sequence of states as in the original execution. Thus, because τ existed at time t, A_i must output DER(τ) at least once before t. Let $t_k < t$ be the last such instance. HANDLE-OUTPUT-DER will be invoked with argument t_k and, as above, cause a suitable EXIST vertex to be added. As above, this vertex can still be modified when τ is underived later, but, since this can only happen after t, the vertex still meets our criteria.

Finally, we consider part 2) of the claim. Assume that $\textsc{exist}(i, \tau, [t_1, t_2]) \in V(h)$. As we have seen, this must mean that τ has been inserted or derived at time t_1. If $t_2 = \infty$, τ can never have been deleted or underived because this would have caused t_2 to be updated, so it must certainly have existed at $t > t_1$. If $t_2 < \infty$, we know that a) a deletion or underivation happened at t_2, and b) no such deletion or underivation can have happened *before* t_2, since EXIST vertices can be updated at most once. But since $t_2 \geq t$, τ must still have existed at time t, and the claim holds. □

B.5 GCA is compositional

A key property of SNP is that the provenance graph is compositional, i.e., we can run the GCA separately on the events from each node, and then stitch together the resulting subgraphs. This is important for two reasons. First, we want the property that the subgraph for a correct node cannot (with very few exceptions) depend on events on faulty nodes, so that an adversary cannot 'change history'. Second, compositionality is important for efficient query processing, since it enables us to reconstruct the subgraph for individual nodes without considering events on other nodes. We formalize the compositionality of the GCA with the following theorem:

Theorem 2 *For any history h, $G(h \,|\, i) = G(h) \,|\, i$.*

Proof: We first prove $G(h \,|\, i) \subseteq^* G(h) \,|\, i$. Observe that, with two important exceptions, the GCA state is completely partitioned by node, so the 'extra' events in h (compared to $h \,|\, i$) do not influence the vertices and edges that are created by events on i. The two exceptions are SEND and RECEIVE vertices: a SEND(m) can be created on SRC(m) when a RCV(m) event is processed on DST(m), and a RECEIVE(m) vertex can be created on DST(m) when RCV(ACK(m)) is processed on SRC(m). However, the remote vertex either already exists, or it is created as a 'standalone' vertex that is not connected to any other vertex on the remote note. In the case of SEND, this does not matter because a SEND vertex can never be a predecessor of a vertex on the same node. Although RECEIVE could in principle have a forward edge to a BELIEVE-(DIS)APPEAR vertex and thus influence the outcome of subsequent HANDLE-OUTPUT invocations, this forward edge is created only when the corresponding RCV event is processed; a RECEIVE that is created *only* in response to a remote event cannot influence further edges or vertices on its node. Finally, note that any remote, disconnected vertices that are not subsequently connected by events on the remote node are colored yellow in line 56 or line 76. These are the only vertices that can have a different color in the two graphs, but since they are yellow and yellow is dominated by both other colors, the property still holds.

Next, we prove that $G(H) \,|\, i \subseteq^* G(h \,|\, i)$, i.e., we have to show that $G(h) \,|\, i$ does not contain additional vertices or edges. Now, recall from above that vertices are only created on the node whose events or outputs are being processed, with the exception of SEND and RECEIVE. Thus, the only vertices that could be in $G(h) \,|\, i$ but not in $G(h|i) \,|\, i$ are SEND and RECEIVE vertices. But a SEND(m) could only be created remotely when a RCV(m) event on another node is processed; since we have assumed that signatures cannot be forged, any such m must have been actually sent by i, and $h|i$ must contain the corresponding SND event, which will cause the SEND(m) vertex to be created locally as well. Similarly, a RECEIVE(m) could only be created remotely when a matching ACK is processed on another node – but then i must have actually sent the ACK, and the corresponding RCV event must be in $h|i$. $G(h) \,|\, i$ cannot contain any extra edges either because the only edges that can be created in response to remote events are those between SEND and RECEIVE. Hence, the claim follows. □

B.6 No red vertices on correct nodes

Since we want to use the color black to denote 'correct' vertices, it is important to have the property that stable vertices on correct nodes are always black, no matter what the adversary may cause the faulty nodes to do. Note the restriction to stable vertices: if a SEND vertex has been added very recently, the corresponding ACK may not have arrived yet, so the GCA has no choice but to color it yellow until the ACK arrives.

Lemma 2 *If h is a history and $i \in N$ is a node that is correct in h, then all stable vertices $v \in V(h)$ with* HOST$(v) =$ *i are colored black.*

Proof: There are only five points in the GCA that can cause a vertex to be colored red, namely in lines 41, 46, 89, 95, and 122. We consider these in turn.

Assume a vertex on i was colored red in line 41. Then A_i must have produced a SND output that was added to the `pending` set in line 189 but was not removed in line 124 before FLAG-ALL-PENDING was invoked. Now, FLAG-ALL-PENDING is invoked whenever a RCV, INS, or DEL event is seen in the history (lines 100, 107, and 130). But a correct node immediately sends all required messages before it proceeds to the next input. Therefore, A_i must have produced a SND output, but i did not actually send the corresponding message. This contradicts our assumption that i is correct in h.

The case for line 122 is quite similar. This line is executed only when a SND event is found in the history that does not correspond to a SND output that has previously been produced (and added to `pending`). But since i is correct, it cannot have fabricated an output, and since A_i is deterministic, the outputs produced during graph construction are the same as the ones produced in the original execution, so we have another contradiction.

Line 95 is reached when a new RECEIVE vertex is created in response to a RCV(m) event and added to `ackpend` (line 140), but the RCV(m) is not immediately followed by a SND(ACK(m)) event; any other event would cause line 95 to be reached either indirectly via FLAG-ALL-PENDING or directly via line 126. But since i is a correct node, it will send an ACK(m) immediately after receiving a message m, so the RECEIVE vertex will be removed from `ackpend` again in line 116 before FLAG-ACK is reached.

Line 89 is only reachable from HANDLE-EXTRA-MSG, which is not used in the basic GCA. (This is invoked when a node forks its log, which a correct node would not do either.)

Finally, assume that a vertex v on i was colored red in line 46. This means that a) v must have been in `unacked`, and b) its timestamp must be older than $2{\cdot}\mathrm{T}_{\mathrm{prop}}$. SEND vertices are added to `unacked` in line 58 when they are created, and removed in line 135 when the corresponding acknowledgment is seen in the history. If one of i's SEND vertices was in this set at the end of graph reconstruction, i must have failed to receive an ACK within $2{\cdot}\mathrm{T}_{\mathrm{prop}}$, but this cannot be the case if h is quiescent. This completes the proof. □

B.7 Red vertices on faulty nodes

Our goal is to use the color red for vertices in G that are definitely the result of misbehavior; thus, it is important to prove that at least one red vertex will always exist on a faulty node. However, keep in mind that G is defined in terms of the true global history h; hence, if a node i is faulty in h but *not* detectably faulty, $G(h)$ will contain a red vertex on i, but we may not be able to reconstruct that vertex in $G_\nu(\epsilon)$, at least for any ϵ that is available on correct nodes.

Lemma 3 *If h is a history and $i \in N$ is a node that is faulty in h, then at least one vertex $v \in V(h)$ with* HOST$(v) = i$ *will be colored red.*

Proof: Recall from Section A.3 what it means for i to be faulty: it can 1) send a message without receiving any input; it can 2) fail to send an acknowledgment immediately after each received message, it can 3) send a message that A_i would not have produced, or it can 4) fail to send a message that A_i did produce.[3] Note that we are *not* considering "lies" such as equivocation or failure to record an incoming message; these types of misbehavior only make sense in the context of SNooPy, where nodes can make claims about their histories (for this, see Appendix C). We consider each of the four types of misbehavior in turn.

If i sends an extra message m that is not an acknowledgment and does not correspond to any output of A_i, this will be detected in line 119 because m does not match any elements in `pending` (recall that such an entry is created for each SND output in line 189). This immediately causes a SEND vertex v with HOST$(v) = i$ to be created in line 120, which is then colored red in line 122. Therefore, claims 1) and 3) hold.

If i fails to send an acknowledgment after receiving a message m, this will cause the vertex $v :=$ RECEIVE(m) to be colored red in line 95. (Clearly, HOST$(v) = i$ because RECEIVE vertices are always created on the receiving node.) The reason is that, in HANDLE-EVENT-RCV, v is added to the `ackpend` send; any subsequent HANDLE-EVENT calls other than HANDLE-EVENT-SND will immediately call FLAG-ALL-PENDING and thus lead to line 95. HANDLE-EVENT-SND can also reach line 95 via the FLAG-ACK call at the end; the only way v can remain black is if v is removed from `ackpend` in line 116, which only occurs if the transmitted message is an acknowledgment for m. Therefore, claim 2) holds.

If i fails to send a message m that A_i produces, this will cause the SEND(m) vertex to be colored red in FLAG-ALL-PENDING, specifically in line 41. This method is called from lines 100, 107, and 130, i.e., whenever A_i accepts the next input. A_i's SND outputs are added to the `pending` set in line 189, and they are removed in line 124 only when the corresponding SND event is seen in the history. If m is output by A_i but not sent by i in h, (m,SEND(m)) will be added but not removed, and the red SEND vertex will be generated by the next call to FLAG-ALL-PENDING. Therefore, claim 4) holds. This completes the proof.

Note that there are two additional ways in which vertices can be colored red, but neither is relevant for the present lemma. Line 89 can be reached only from HANDLE-EXTRA-MSG, which is not used here, and line 46 cannot be reached because we have assumed that h is quiescent. □

B.8 GCA uses colors appropriately

Now we can easily prove the following corollary:

Theorem 3 *If h is a history, then at least one vertex $v \in V(h)$ with* HOST$(v) = i$ *will be colored red if and only if $i \in N$ is faulty in h.*

Proof: This follows directly from the combination of Lemma 2 and Lemma 3. □

C. GRAPH WITH HISTORIES

In this section, we extend the model from Section B to a distributed setting. We no longer assume that the 'true' history of the primary system is known; rather, we assume that nodes can make claims about their local histories. We model these claims as a history map $\varphi(m)$, which is attached to each message m. Correct nodes set this map to their own history prefix at the time the message is sent; faulty nodes can set it arbitrarily. The problem, then, becomes how to reconstruct the provenance graph from a set of history maps, e.g., the ones that correspond to messages received by a correct node. The resulting provenance graph may not be the 'true' provenance graph that would be generated from the actual execution, but we can establish a close correspondence between the two.

C.1 History map

We say that a history $h|i$ of a node i is *valid* if it conforms to A_i; if i is correct in a history h, one trivial example of a valid history of i is $h|i$. A pair (h_1, h_2) of histories of i is *consistent* if h_1 is a prefix of h_2, or vice versa.

Let $M(e)$ denote the set of messages received by the nodes in an execution e. We assume that there exists a *history map* φ that associates every message $m \in M(e)$ with a history

[3] Recall that, in Section A.3, correctness was defined with respect to a hypothetical state machine that is fed with all the inputs that i actually receives. Hence, we do not need to consider cases where i ignores inputs or injects fictitious inputs.

of $\text{SRC}(m)$. For a correct node, $\varphi(m)$ is the prefix of the local execution $e|\text{SENDER}(m)$ up to an including $\text{SND}(dest, m)$. Thus, for any message m sent by a correct node, $\varphi(m)$ is valid, and for every pair of messages m and m' sent by a correct node, $\varphi(m)$ and $\varphi(m')$ are consistent.

C.2 Observability and detectable faultiness

We say that a message m_0 is *causally precedes* another message m_k if there exists a sequence of messages $m_1, \dots, m_{k-1}$ such that, for all $j = 1, \dots, k$, $\text{RCV}(m_{j-1})$ belongs to $\varphi(m_j)$. We say that a message m is *observable* in e from a correct node i if there exists a message m' such that a) $\text{RCV}(\text{m}')$ belongs to $e \,|\, i$, and b) m causally precedes m'.

We say that a *symptom* of a fault on a node i in an execution e is a set S of at most two messages that meet one of the following two criteria:

1. $S := \{m\}$ such that $\text{SRC}(m) = i$ and $\varphi(m)$ is not valid for i, or
2. $S := \{m_1, m_2\}$ such that $\text{SRC}(m_1) = \text{SRC}(m_2) = i$ and $\varphi(m_1)$ is not consistent with $\varphi(m_2)$.

We say that a node i is *detectably faulty* in an execution e if there exists a symptom S such that each $m \in S$ is observable from some correct node. We say that a symptom S of a detectable fault on i in e is observable from a correct node j if *each* message $m \in S$ is observable from j.

[16] additionally defines the concept of detectable ignorance, which captures the case where a node directly refuses to respond to an incoming message. Here, we do not consider this case; instead, we assume that a correct node will raise an alarm whenever another node does not acknowledge one of its messages within $2{\cdot}\text{T}_{\text{prop}}$. Formally, we say that an execution e is *quiescent* iff, for any node i that is correct in e, each event $(t_x, i, \text{SND}(m)$ is followed by a $(t_y, i, \text{RCV}(\text{ACK}(m))$ such that $t_x < t_y \leq t_x + 2{\cdot}\text{T}_{\text{prop}}$. In the following, we assume that all histories and executions are quiescent.

C.3 Evidence sets and views

We define an *evidence set* ϵ to be a sequence of messages $\nu := (m_1, m_2, \dots, m_k)$. We say that m_x is the *primary message* for a node $i \in N$ in ϵ iff a) $\text{SRC}(m_x) = i$ and b) there is no message $m_y \in \nu$, $y < x$, such that $\text{SRC}(m_y) = i$. We say that m_y is the *dominant message* for a node $i \in N$ in ϵ if either a) m_y is the primary message for i, or b) $\varphi(m_x)$ is a prefix of $\varphi(m_y)$, where m_x is the primary message for i, and ϵ contains no other message m_z such that $\varphi(m_y)$ is a prefix of $\varphi(m_z)$. In other words, if an evidence set contains conflicting history mappings from a given node, then the primary message controls which mapping is used, and the dominant message is the one with the longest mapping.

We say that the *view* $\nu(\epsilon)$ from an evidence set ϵ is the combination of the history maps of all dominant messages in ϵ. We define the provenance graph $G_\nu(\epsilon)$ to be the graph $G(\nu(\epsilon))$, except that at the end of the GCA, HANDLE-EXTRA-MSG is additionally called for each $m \in \epsilon$ that is not a primary message.

We say that an evidence set ϵ is *probative* for a vertex v if, for any set of messages M, $G_\nu(\epsilon + M)$ contains v. We say that ϵ is probative for an edge (v_1, v_2) if, for any set of messages M, $G_\nu(\epsilon + M)$ contains the edge (v_1, v_2).

C.4 onotonicity

Theorem 4 *Let ϵ be an evidence set and m a message. Then $G_\nu(\epsilon) \subseteq^* G_\nu(\epsilon + m)$.*

Proof: Let $i := \text{SRC}(m)$. We first observe that the view $\nu(\epsilon)$ must be a subsequence of the view $\nu(\epsilon + m)$: if ϵ did not contain a message from i, then $\nu(\epsilon)|i = \emptyset$ and $\nu(\epsilon + m)|i = \varphi(m)$; otherwise $\nu(\epsilon{+}m)$ can only be different from $\nu(\epsilon)$ if m becomes the dominant message for i, in which case $\nu(\epsilon{+}m)|i$ is an extension of $\nu(\epsilon)|i$, and $\nu(\epsilon{+}m)|j = \nu(\epsilon)|j$ for all $j \neq i$.

There are four cases we have to consider, depending on whether m is primary and/or dominant in $\epsilon + m$. If m is dominant but not primary, $\nu(\epsilon)$ is a prefix of $\nu(\epsilon + m)$, and due to the monotonicity of the graph construction algorithm (Lemma 1), the claim holds. If m is primary (and, by implication, also dominant), we know that $\nu(\epsilon)$ does not yet contain any events on $\text{SRC}(m)$, so, by the compositionality of the graph construction algorithm (Lemma 2), the claim holds. If m is neither primary nor dominant, we have $\nu(\epsilon + m) = \nu(\epsilon)$, so, if $\varphi(m)$ is already part of $\nu(\epsilon) \,|\, i$, the claim holds trivially; otherwise, the addition of m will cause an invocation of HANDLE-EXTRA-MSG, which will add a red SEND vertex to G_ν, and the claim holds as well. □

C.5 View accuracy

The following theorem says that, given valid evidence from a history h, the GCA can always accurately reconstruct the vertices on the correct nodes, no matter what the faulty nodes do. A simple corollary says that these vertices must all be colored black.

Theorem 5 *Let $i \in N$ be a node and h be a history such that 1) $h \,|\, i$ ends in $\text{SND}(m)$, and 2) i is correct in h. Let ϵ be an evidence set that consists of messages from h, including m. Then $G_\nu(\epsilon) \,|\, i = G(h) \,|\, i$.*

Proof: First, we show that m must be the dominant message for i in ϵ. Recall from Section C.1 that a correct node always sets the history map of a message m to its own history prefix at the time of transmission. This means that a) $\varphi(m) = h|i$, and b) for any message m' that was sent by i in h, $\varphi(m')$ is a prefix of $\varphi(m)$. The latter obviously includes all other messages from i that ϵ may contain, so m must be the dominant message for i.

Next, we show that $G_\nu(\epsilon) = G(\nu(\epsilon))$. According to the definition of G_ν, any discrepancy could only come from the invocations of HANDLE-EXTRA-MSG at the end. Observe that HANDLE-EXTRA-MSG will only create vertices that do not already exist. But since i is correct, any SND events for messages it has sent in h must already be in $\varphi(m)$, so the corresponding vertices on i must already exist.

Finally, we show that $G(\nu(\epsilon))|i = G(h|i)|i$. Since m is dominant for i, we get from the definition of views that $\nu(\epsilon)|i = \varphi(m)$. But the history map returns only node-local events, so $\varphi(m) = \varphi(m)|i$, and since i is correct and m was the last message i sent in h, we know that $\varphi(m) = h|i$. Taken together, we have $\nu(\epsilon)|i = h|i$, and thus obviously $G(\nu(\epsilon))|i = G(h|i)|i$, which according to compositionality (Lemma 2) is the same as $G(h)|i$. □

C.6 View completeness

Theorem 6 *Let h be a history, $i \in N$ a node that is correct in h, $j \in N$ a node that is detectably faulty in h, and ϵ the set* *of messages received by the correct nodes in h. Let ϵ' be the set of messages that can be obtained by recursively extracting all the messages from $\nu(\epsilon)$. Then $G_\nu(\epsilon+\epsilon')$ contains at least one red vertex v with* HOST$(v) = j$*.*

Proof: According to the definition of a detectable fault in Section C.2, there must be a symptom S of the fault on j. We first show that, for any evidence ϵ'' (including $\epsilon'' = \emptyset$), $G_\nu(\epsilon + \epsilon'' + S)$ contains a red vertex v with HOST$(v) = j$. First, consider the case where $S = \{m\}$ and $\varphi(m)$ is not valid for j. If $\varphi(m)$ is inconsistent with $\nu(\epsilon + \epsilon'') \,|\, j$, HANDLE-EXTRA-MSG will be invoked with m as its argument, which will add a suitable red vertex; otherwise $\varphi(m)$ must be a prefix of $\nu(\epsilon + \epsilon'' + m) \,|\, j$; according to Theorem 1 (GCA is incremental) and Theorem 3 (GCA uses colors appropriately), the red vertex must appear in $G(\nu(\epsilon+\epsilon''+m) \,|\, j)$ and thus, according to Theorem 2 (GCA is compositional), also in $G_\nu(\epsilon + \epsilon'' + m) \,|\, j$. Now consider the case where $S = \{m_1, m_2\}$ and $\varphi(m_1)$ is not consistent with $\varphi(m_2)$. In this case, m_2 must be be inconsistent with $\nu(\epsilon + \epsilon'' + m_1)$, and thus HANDLE-EXTRA-MSG will be invoked with m_2 as its argument, which will add a suitable red vertex.

Now recall that the symptom must be observable from a correct node, i.e., each $m \in S$ must causally precede some message on a correct node. Since ϵ already contains all the messages received by correct nodes, there must exist, for each $m \in S$, a chain of messages $m_1, \ldots, m_k$ such that $m_1 = m$, $m_k \in \epsilon$, and, for each $x = 1, \ldots, k-1$ RCV(m_x) belongs to $\varphi(m_{x+1})$. But since ϵ' was obtained by recursively extracting messages from $\nu(\epsilon)$, we must have obtained each m_x along the way, starting with m_k and working our way along the chain to m_1. Hence, $S \subseteq \epsilon'$, and the claim follows. □

D. MAPPING TO SNOOPY

As a final step, we establish a correspondence between the GCA and the SNooPy system. In SNooPy, the logs maintained by the graph recorder are essentially histories, except that, for convenience, the latter contain an explicit ACK entry type instead of a RCV(ACK). The history map φ is implemented efficiently by the authenticators: rather than attaching the entire history to each message m, the nodes include a digest of the history. Commitment ensures that this history matches the requirements for $\varphi(m)$, i.e., it ends with SND(m)).

SNooPy securely evaluates $\varphi(m)$ simply by challenging SRC(m) to return the history that corresponds to m's authenticator a_m; since $H(\cdot)$ is a cryptographic hash function (assumption 3 in Section 5.2) and therefore second pre-image resistant, it is infeasible for SRC(m) to compute a second history that also matches a_m. This is, in essence, what RETRIEVE attempts to do, except that of course a faulty node can send a response that does not match the authenticator, or fail to respond at all. SNooPy handles this by initially coloring the queried vertex yellow, and by changing its color only after a response is received. Since a failure to respond to RETRIEVE (when it is invoked on a properly signed authenticator) itself constitutes a fault, leaving the vertex yellow in this case is permissible.

Vertex	May have inbound edges from...	May have outbound edges to...
INSERT	-	APPEAR
DELETE	-	DISAPPEAR
APPEAR	INSERT, DERIVE	EXIST, SEND, DERIVE
DISAPPEAR	DELETE, UNDERIVE	EXIST, SEND, UNDERIVE
EXIST	APPEAR, DISAPPEAR	DERIVE, UNDERIVE
DERIVE	EXIST, BELIEVE, APPEAR, BELIEVE-APPEAR	APPEAR
UNDERIVE	EXIST, BELIEVE, DISAPPEAR, BELIEVE-DISAPPEAR	DISAPPEAR
SEND	APPEAR, DISAPPEAR	RECEIVE
RECEIVE	SEND	BELIEVE-APPEAR, BELIEVE-DISAPPEAR
BELIEVE-APPEAR	RECEIVE	BELIEVE, DERIVE
BELIEVE-DISAPPEAR	RECEIVE	BELIEVE, UNDERIVE
BELIEVE	BELIEVE-APPEAR, BELIEVE-DISAPPEAR	DERIVE, UNDERIVE

Table 1: Edges that may be present in the provenance graph

```
variables
    pending : set          ▷ Output messages not yet seen
    ackpend : set          ▷ RECEIVE vertex, no ACK yet
    unacked : set          ▷ Sent messages not yet acked
    nopreds : set          ▷ SEND without incoming edge
end variables

function APPEAR-LOCAL-TUPLE(i, $\tau$, $v_{why}$, t)
    $v_1 \leftarrow$ APPEAR(i, $\tau$, t)
    $v_2 \leftarrow$ EXIST(i, $\tau$, [t,$\infty$))
    $v \leftarrow v \cup \{v_1, v_2\}$
    $e \leftarrow e \cup \{(v_{why}, v_1), (v_1, v_2)\}$
end function

function DISAPPEAR-LOCAL-TUPLE(i, $\tau$, $v_{why}$, t)
    $v_1 \leftarrow$ DISAPPEAR(i, $\tau$, t)
    $v_2 \leftarrow$ EXIST(i, $\tau$, [$t_{ins}$, $\infty$))
    $v \leftarrow v \cup \{v_1\}$
    $v_2$.REPLACE-WITH(EXIST(i, $\tau$, [$t_{ins}$, t]))
    $e \leftarrow e \cup \{(v_{why}, v_1), (v_1, v_2)\}$
end function

function APPEAR-REMOTE-TUPLE(i, $\tau$, j, $v_{why}$, t)
    $v_1 \leftarrow$ BELIEVE-APPEAR(i, j, $\tau$, t)
    $v_2 \leftarrow$ BELIEVE(i, j, $\tau$, [t,$\infty$))
    $v \leftarrow v \cup \{v_1, v_2\}$
    $e \leftarrow e \cup \{(v_{why}, v_1), (v_1, v_2)\}$
end function

function DISAPPEAR-REMOTE-TUPLE(i, $\tau$, j, $v_{why}$, t)
    $v_1 \leftarrow$ BELIEVE-DISAPPEAR(i, j, $\tau$, t)
    $v_2 \leftarrow$ BELIEVE(i, j, $\tau$, [$t_{ins}$, $\infty$))
    $v \leftarrow v \cup \{v_1\}$
    $v_2$.REPLACE-WITH(BELIEVE(i, j, $\tau$, [$t_{ins}$, t]))
    $e \leftarrow e \cup \{(v_{why}, v_1), (v_1, v_2)\}$
end function

function FLAG-ALL-PENDING(i, t)
    FLAG-ACKPEND(i)
    for each (i, ?, $v_1$) $\in$ pending do
        $v_1$.SET-COLOR(RED)
        pending $\leftarrow$ pending $\setminus$ {(i, ?, $v_1$)}
        unacked $\leftarrow$ unacked $\setminus$ {(i, $v_1$)}
    end for
    for each (i,$v_2$) $\in$ unacked with $v_2$.T<t-2·$T_{prop}$ do
        $v_2$.SET-COLOR(RED)
        unacked $\leftarrow$ unacked $\setminus$ {(i, $v_2$)}
    end for
end function

function ADD-SEND-VERTEX(m, $v_{why}$, t)
    $v_1 \leftarrow$ V.GET(SEND(SRC(m), DST(m), m, t))

    if $v_1 = \perp$ then
        $v_1 \leftarrow$ SEND(SRC(m), DST(m), m, t)
        $v \leftarrow v \cup \{v_1\}$
        $v_1$.SET-COLOR(YELLOW)
        nopreds $\leftarrow$ nopreds $\cup$ {$v_1$}
        unacked $\leftarrow$ unacked $\cup$ {(SRC(m),$v_1$)}
    end if

    if ($v_1 \in$ nopreds) and ($v_{why} \neq \perp$) then
        $e \leftarrow e \cup \{(v_{why}, v_1)\}$
        nopreds $\leftarrow$ nopreds $\setminus$ {$v_1$}
    end if

    return $v_1$
end function

function ADD-RECEIVE-VERTEX(m, t)
    ADD-SEND-VERTEX(m, $\perp$, TXMIT(m))
    $v_1 \leftarrow$ V.GET(RECEIVE(DST(m), SRC(m), m, t))

    if $v_1 = \perp$ then
        $v_1 \leftarrow$ RECEIVE(DST(m), SRC(m), m, t)
        $v \leftarrow v \cup \{v_1\}$
        v.SET-COLOR(YELLOW)
    end if

    if $v_2 \leftarrow$ V.GET(SEND(SRC(m), DST(m), m, ?)) then
        $e \leftarrow e \cup \{(v_2, v_1)\}$
    end if

    return $v_1$
end function

function ADD-RED-UNLESS-PRESENT($v_1$)
    if $v_1 \notin v$ then
        $v \leftarrow v \cup \{v_1\}$
        $v_1$.SET-COLOR(RED)
    end if
end function

function FLAG-ACKPEND(i)
    for each (i, $v_1$) $\in$ ackpend do
        $v_1$.SET-COLOR(RED)
        ackpend $\leftarrow$ ackpend $\setminus$ {(i, $v_1$)}
    end for
end function
```

Figure 10: Provenance graph: Variables and library functions

```
function HANDLE-EVENT-INS(i, $\tau$, t)
    FLAG-ALL-PENDING(i, t)
    $v_1 \leftarrow$ INSERT(i, $\tau$, t)
    $v \leftarrow v \cup \{v_1\}$
    APPEAR-LOCAL-TUPLE(i, $\tau$, $v_1$, t)
end function

function HANDLE-EVENT-DEL(i, $\tau$, t)
    FLAG-ALL-PENDING(i, t)
    $v_1 \leftarrow$ DELETE(i, $\tau$, t)
    $v \leftarrow v \cup \{v_1\}$
    DISAPPEAR-LOCAL-TUPLE(i, $\tau$, $v_1$, t)
end function

function HANDLE-EVENT-SND(i, m, t)
    if m is ACK($m'$) then
        if $v_1 \leftarrow$ v.GET(RECEIVE(SRC($m'$), i, $m'$, t)) then
            ackpend $\leftarrow$ ackpend $\setminus$ $\{(i, v_1)\}$l
            $v_1$.SET-COLOR(BLACK)
        end if
    else if not $\exists v_3$: (i, m, $v_3$) $\in$ pending then
        $v_2 \leftarrow$ ADD-SEND-VERTEX(m, $\perp$, t)
        unacked $\leftarrow$ unacked $\setminus$ $\{(i, v_2)\}$
        $v_2$.SET-COLOR(RED)
    else
        pending $\leftarrow$ pending $\setminus$ $\{(i, m, v_3)\}$
    end if
    FLAG-ACKPEND(i)
end function

function HANDLE-EVENT-RCV(i, m, t)
    FLAG-ALL-PENDING(i, t)
    if m is ACK($m'$) then
        ADD-RECEIVE-VERTEX($m'$, TXMIT(m))
        $v_1 \leftarrow$ v.GET(SEND(i, SRC(m), $m'$, TXMIT($m'$)))
        if (i,$v_1$) $\in$ unacked then
            unacked $\leftarrow$ unacked $\setminus$ $\{(i, v_1)\}$
            $v_1$.SET-COLOR(BLACK)
        end if
    else
        $v_1 \leftarrow$ ADD-RECEIVE-VERTEX(m, t)
        ackpend $\leftarrow$ ackpend $\cup$ $\{(i, v_1)\}$
        if m is ($+\tau$, ?) then
            APPEAR-REMOTE-TUPLE(i, $\tau$, SRC(m), $v_1$, t)
        else
            DISAPPEAR-REMOTE-TUPLE(i,$\tau$,SRC(m),$v_1$,t)
        end if
    end if
end function
```

```
function HANDLE-OUTPUT-DER(i, $\tau$ :- $\tau_1$, $\tau_1$, ..., $\tau_k$)
    $v_1 \leftarrow$ DERIVE(i, $\tau$, $\tau$ :- $\tau_1$, $\tau_1$, ..., $\tau_k$, t)
    $v \leftarrow v \cup \{v_1\}$
    for each x $\in$ $\{1 \ldots k\}$ do
        if $v_2 \leftarrow$ v.GET(BELIEVE-APPEAR(i,?,$\tau_x$,t)) then
            $e \leftarrow e \cup \{(v_2, v_1)\}$
        else if $v_3 \leftarrow$ v.GET(APPEAR(i, $\tau_x$, t)) then
            $e \leftarrow e \cup \{(v_3, v_1)\}$
        else if $v_4 \leftarrow$v.GET(BELIEVE(i,?,$\tau_x$,[?,$\infty$))) then
            $e \leftarrow e \cup \{(v_4, v_1)\}$
        else
            $e \leftarrow e \cup$ {(EXIST(i, $\tau_x$, [?, $\infty$)), $v_1$)}
        end if
    end for
    APPEAR-LOCAL-TUPLE(i, $\tau$, $v_1$, t)
end function

function HANDLE-OUTPUT-UND(i, $\tau$ :- $\tau_1$, $\tau_1$, ..., $\tau_k$)
    $v_1 \leftarrow$ UNDERIVE(i, $\tau$, $\tau$ :- $\tau_1$, $\tau_1$, ..., $\tau_k$, t)
    $v \leftarrow v \cup \{v_1\}$
    for each x $\in$ $\{1 \ldots k\}$ do
        if $v_2 \leftarrow$ v.GET(BELIEVE-DISAPP.(i,?,$\tau_x$,t)) then
            $e \leftarrow e \cup \{(v_2, v_1)\}$
        else if $v_3 \leftarrow$ v.GET(DISAPPEAR(i, $\tau_x$, t)) then
            $e \leftarrow e \cup \{(v_3, v_1)\}$
        else if $v_4 \leftarrow$v.GET(BELIEVE(i,?,$\tau_x$,[?,$\infty$))) then
            $e \leftarrow e \cup \{(v_4, v_1)\}$
        else
            $e \leftarrow e \cup$ {(EXIST(i, $\tau_x$, [?, $\infty$)), $v_1$)}
        end if
    end for
    DISAPPEAR-LOCAL-TUPLE(i, $\tau$, $v_1$, t)
end function

function HANDLE-OUTPUT-SND(i, m, t)
    if m is ($+\tau$, s) then
        $v_{why} \leftarrow$ v.GET(APPEAR(i, $\tau$, t))
    else
        $v_{why} \leftarrow$ v.GET(DISAPPEAR(i, $\tau$, t))
    end if
    $v_1 \leftarrow$ ADD-SEND-VERTEX(m, $v_{why}$, t)
    pending $\leftarrow$ pending $\cup$ { (i, m, $v_1$) }
end function

function HANDLE-EXTRA-MSG(m)
    (t, i, j) $\leftarrow$ (TXMIT(m), SRC(m), DST(m))
    ADD-RED-UNLESS-PRESENT(SEND(i, j, m, t))
    ADD-RED-UNLESS-PRESENT(RECEIVE(j, i, m, t))
end function
```

Figure 11: Provenance graph: Functions for handling events (left) and outputs from the state machine (right)