Chapter 7

ANALYZING DISCREPANCIES IN WHOLE-NETWORK PROVENANCE

Raza Ahmad, Aniket Modi, Eunjin Jung, Carolina de Senne Garcia, Hassaan Irshad and Ashish Gehani

Abstract Data provenance describes the origins of a digital object. This information is particularly useful when analyzing distributed workflows because extant tools, such as debuggers and application profilers, do not support tracing through heterogeneous executions that span multiple hosts. In a decentralized system, each host maintains the authoritative record of its own activity in the form of a dependency graph. Reconstructing the provenance of an object may involve the assembly of subgraphs from multiple, independently-administered hosts. The collection of host-specific dependencies coupled with cross-host flows comprise the whole-network provenance, which can grow to terabytes for a small network.

Critical infrastructure assets face constant attacks and despite best efforts, some attacks, such as those using zero-day exploits, succeed. Whole-network provenance has become a common basis for post-attack forensic analyses with the creation of DARPA's Transparent Computing Program. This chapter describes and analyzes aspects of distributed querying, caching and response discrepancy detection used in forensic analyses that are specific to provenance.

Keywords: Distributed Provenance, Data Provenance, Discrepancy Detection

1. Introduction

Provenance collection and analysis are useful for studying distributed application programs. These programs may coordinate workflows across multiple interconnected hosts and combine the results [19]. This is important for consortia of institutions that share data and resources for large-scale tasks such as TeraGrid [3] and XSEDE [20]. Provenance metadata from these systems may span multiple administrative domains.

These records collected from a single host are termed whole-system provenance [17]. "Whole-network provenance" is defined as metadata that describes the relationships between whole-system provenance on individual hosts coupled with the set of distributed data flows connecting processes on the hosts.

Whole-network provenance became a common basis for detecting stealthy advanced persistent threats with the creation of DARPA's Transparent Computing Program [5]. Critical infrastructure assets in the form of network-facing services, such as access to code repositories and domain name resolution, may come under attack. Despite best efforts to secure critical infrastructure assets, attacks often succeed and subsequent forensic analyses are of utmost importance to identify the attack vectors and the scopes of the attacks. One aspect of forensic analysis involves querying provenance agents on hosts in a distributed system such as an enterprise or government organization. Systems that collect and analyze whole-network provenance are now being deployed at scale. For example, DISTDET has been installed on more than 22,000 hosts at over 50 industrial customers [6].

In these settings, individual hosts can send queries to other hosts to obtain the full provenance data of an item such as a file downloaded from a remote host. In a decentralized querying approach, each host receives responses from remote hosts to its own queries, but also forwards responses to queries from other hosts as well. Any subset of these responses can be stored in local storage to build a host cache. When a network is too slow or expensive, the host may run a provenance query on its own cache to obtain a preliminary query result.

Provenance metadata collected from remote hosts is not necessarily reliable and trustworthy. Some hosts may have buggy software, some may send outdated data, some may suffer from network fluctuations and some may be malicious. Provenance discrepancy is defined as the difference between truthful provenance and a response received by the querying or intermediate host. Since provenance is a record of the history of computation, the later metadata from a host can have more elements and relationships between the elements than before, but not less. This "append-only" nature of provenance metadata is leveraged to detect and report a discrepancy whenever a query response is missing an element from the previously-known provenance metadata in the cache.

The ability to detect discrepancies from missing graph elements is important in several real-world applications. Four scenarios include a product failure that exposes a company to legal liabilities in case of forensic analysis, a legal battle over patent infringement by a company to deny prior possession of references, an accident as a result of a computational error, and a claim of credit for a discovery after learning about a competitor's result [8]. These scenarios motivate the alteration of provenance data after an incident has occurred. Data modifications manifest themselves as deletions of old elements and insertions of new elements, which cause discrepancies in provenance data.

2. Background

The open-source SPADE middleware [12] is employed in this study. SPADE supports a number of operating systems for provenance management. In particular, it supports the use of the Linux Audit framework as a source to derive whole-system provenance [17]. However, the ideas in this research apply to any provenance management framework that supports decentralized operation.

A provenance graph $G(V, E)$ contains a set of vertices V and a set of edges E, where edges in E connect vertices in V. Each vertex $v \in V$ corresponds to an agent, process or artifact that is the subject or object of an operation. Each vertex is characterized by a unique key-value set of annotations $A(v)$: $A(v) = \{a_1, a_2, \ldots, a_n\}$ where $a_i = \langle key_i : value_i \rangle$. For example, a vertex representing an operating system process would contain annotations such as $\langle pid : 2 \rangle$, $\langle user : root \rangle$, $\langle time : 1345012 \rangle$. The annotation set is unique because there is only one process with a certain *pid* at a given *time*. Hence, to uniquely identify vertex v with a single attribute, a content-based hash identifier id_v is constructed by hashing the concatenation of all the key-value pairs: $id_v = hash(a_1 \parallel a_2 \parallel \cdots \parallel a_n)$.

Note that any change to a key-value pair results in changing the vertex to a different vertex. For example, if a malicious host changes the time in vertex $v = \{pid : 2, time : t_1\}$ to $\{pid : 2, time : t_2\}$, then the hash identifier would change and v would become a different vertex $v' = \{pid : 2, time : t_2\}$ and the provenance graph $G(V, E)$ would change to $G(V', E)$ where $V' = V \setminus \{v\} \cup \{v'\}$.

An edge in E is an operation on a pair of vertices and corresponds to a directed edge between them, specifying a data dependency. For example, a system `read()` call results in an edge from a process vertex to a file vertex and contains annotations such as $\langle size : 1024 \rangle, \langle time : 1345121 \rangle$. Each edge $e \in E$ is defined by the two vertices, X and Y, on which it is incident, and a set of annotations $A(e)$: $e = \{X, Y, A(e)\}$. Each edge is uniquely identified by a content-based identifier id_e by hashing the concatenation of the identifiers of the incident vertices id_X and id_Y and the elements of the annotation set $A(e)$: $id_e = hash(id_X \parallel id_Y \parallel a_1 \parallel a_2 \parallel \cdots \parallel a_n)$. As with a change to a vertex, any change to an annotation in $A(e)$ results in changing the edge by deleting the original edge and adding a new edge to E.

3. Whole-Network Provenance

Whole-network provenance is formally defined as the metadata that describes the intra-host whole-system provenance of each host in the network coupled with the inter-host flows between pairs of hosts. Using whole-network provenance graphs, the provenance of an object can be reconstructed by starting from one host and tracking back through other relevant hosts.

The provenance graph on a host H_i is defined as $G_{H_i} = (V_{H_i}, E_{H_i})$. The inter-host flow created between two hosts H_i and H_j is given by the tuple of network artifacts connecting them:

$$F_{i,j} = (n_i, n_j) : n_i \in G_{H_i}, n_j \in G_{H_j}, i \neq j, n_i = n_j$$

where n_i is the network artifact vertex on host H_i.

The whole-network provenance graph is defined as:

$$G_{network} = \bigcup_i G_{H_i} \cup \bigcup_{i,j,i \neq j} F_{i,j}$$

where H_i is a host on the network and $F_{i,j}$ is a flow between two hosts H_i and H_j on the network.

In a centralized strategy, each host uploads its own provenance metadata periodically to a single repository that handles all provenance queries. This approach simplifies the coordination between hosts, but suffers from three limitations. First, all hosts in a network are required to periodically send all their provenance metadata to the central repository, although other hosts may not need much of it. Second, the central repository may become a performance bottleneck, especially in terms of bandwidth because simultaneous uploads from multiple hosts may render it unavailable for processing queries. Third, the reliability of the entire system decreases because the central repository becomes a single point of failure. Note that a data integrity compromise at the repository can affect the provenance metadata of the entire network.

The proposed approach employs a decentralized, peer-to-peer architecture. Each connected host in the network is independently responsible for collecting and storing its own metadata. Individual hosts can completely satisfy all local queries. They may also collect provenance metadata by querying other hosts in the network. The querying host then combines all the responses from the remote hosts.

This mechanism provides a scalable approach for whole-network provenance collection because it does not have the aforementioned limitations of a centralized approach. The mechanism also has four benefits. First, less resources are required per host – no single host is required to have sufficient resources to maintain complete copies of provenance from all hosts. Second, there is no wasted data transfer – all the transferred data is necessary to respond to specific queries. Third, there is resilience to network fluctuations – individual hosts can use their own caches to answer queries in the case of network instability. Fourth, individual hosts have the freedom to implement their own data management policies, such as the database to use and the retention period of archival copies.

At the heart of this decentralized metadata collection is a construct called the network artifact [9, 12]. Its key property is that it can be constructed without any explicit coordination at independent endpoints. In the context of a distributed system, a pair of network artifacts indicates a data flow between two hosts. For operating system provenance, network artifacts are constructed using the IP addresses and ports of the endpoints, combined with the times when the connections were established.

4. Distributed Querying

In a distributed, decentralized environment, the host that originates a query is responsible for collecting its responses. After resolving the query locally, the host contacts remote hosts through network artifacts that subsequently return their results and contact other hosts if required. The responses are stitched together at the originating host to create a single connected provenance graph. This approach enables remote hosts located the same distance away in the network to be contacted in parallel. Thus, the distributed querying time increases linearly with the height of the network topology tree regardless of the number of remote hosts.

A provenance management system that operates in a distributed environment may collect provenance metadata across several hosts. Two of the most common operations in collecting provenance are lineage and path queries. The lineage of an item traces its past (ancestors) or future impact (descendants). The response to a lineage query is a directed graph. Lineage queries are sent with a maximum depth d to limit the retrieved provenance because the size of a provenance graph could grow rapidly over multiple hosts.

To formally define a lineage ancestor query from a vertex v for depth d, it is necessary to first define the parent graph of v: $G_P(v) = (P, E)$,

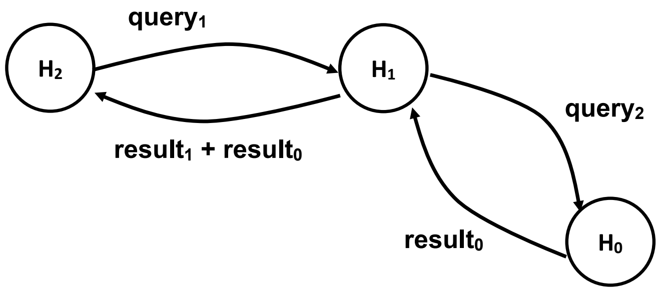

Figure 1. Interconnected hosts querying provenance in a distributed manner.

where P is a set of vertices such that $\forall p \in P$, an edge $e \in E$ exists and $e = (v, p)$. The lineage of v is given by:

$$\begin{aligned} l(v,d) &= G_P(v) \cup l(p, d-1) \;\; \forall p \in P \\ l(v,0) &= v \end{aligned}$$

The response to a lineage query is always a connected graph in which the directions of edges represent the information flow. Thus, given a graph $G_{response}$ sent in response to a lineage query q from vertex v, $\forall u \in G_{response}$, a path exists between any two vertices:

$$\exists\, u \rightsquigarrow v \qquad \text{(descendant query)}$$
$$\exists\, v \rightsquigarrow u \qquad \text{(ancestor query)}$$

Also, $\forall e = (x, y) \in G_{response}$:

$$x, y \in G_{response} \;\; \wedge \;\; \exists\, y \rightsquigarrow v \qquad \text{(descendant query)}$$
$$x, y \in G_{response} \;\; \wedge \;\; \exists\, v \rightsquigarrow x \qquad \text{(ancestor query)}$$

A path query requests the provenance between two objects. Its response is a set of chains from one element to another. The response to a path query is constructed by finding the intersection of lineage ancestor queries from the sink and lineage descendant queries from the source when obtaining all the paths from a particular source to a sink.

When a host needs to see the history of an artifact (e.g., downloaded file) – specifically, where the artifact originated and when and how it was changed before arriving at the host – the host may send a lineage ancestors query to its upstream hosts. The term query host refers to the host from which the lineage inquiry originates.

Figure 1 shows a network of three interconnected hosts where H_2 is the query host, H_1 is the intermediate host and H_0 is the source host. In this case, H_2 wishes to find the lineage of file f_2 on H_2 and learns that the file was downloaded from H_1. H_2 becomes the query host and sends $query_1$ to the upstream host H_1 requesting for provenance metadata of file f_2. H_1 observes that the provenance of f_2 on H_1 continues to H_0. This could happen in one of two cases – f_2 could have been downloaded from H_0 or the process that modified f_2 could have been involved in a network connection between H_1 and H_0. At this point, H_1 becomes the intermediate host and sends $query_2$ to the next upstream host H_0 requesting the provenance metadata of file f_2. If f_2 originated from H_0, then H_0 is the source host and it responds with $result_0$.

The origin and type of a query implicitly define whether one host is upstream or downstream of another. When a query is performed at H_2 about metadata that originated from H_1, H_1 is upstream of H_2 in the context of a lineage ancestors query (and its response). Similarly, H_2 is downstream of H_1 in this context.

However, the converse holds for a lineage descendants query. Specifically, if the query is targeted at host H_1 about metadata that flowed from the host to H_2, then H_2 would be upstream of H_1. Of course, the same pair of hosts could be upstream of each other in the context of different queries. In the rest of this chapter, lineage query is used as shorthand for a lineage ancestors query or a lineage descendants query, where the precise meaning is determined by the context.

5. Caching

It is assumed that each host manages its own cache of provenance metadata from remote hosts. Using cached data to save bandwidth and reduce latency is a common practice in distributed systems. Provenance metadata benefits from similar approaches [11]. When a host receives a response from an upstream host – as a querying host or intermediate host – the host adds the response to its cache. Each response is stored as a directed graph, so the cache is essentially a set of directed graphs.

When a host has a lineage or path query that involves remote hosts, the cache can be also used to obtain a (potentially outdated) local response when communications between the network and other hosts are not reliable or too expensive, and also when low latency is more important than freshness. This cache is denoted as G_{cache} because it contains provenance graphs created from previously-received query responses from other hosts.

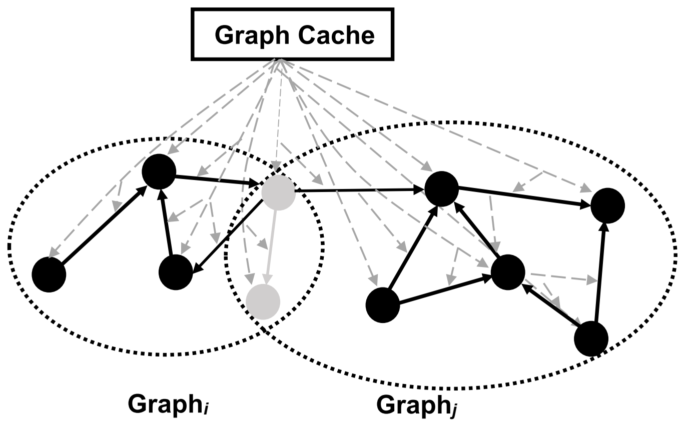

Figure 2. Cache containing responses to two queries with partial overlap.

Figure 2 shows an example graph cache containing two previously-received query responses, $Graph_i$ and $Graph_j$. The shaded vertices and edges are shared by both graphs and stored only once to save memory. When the response to a query overlaps with the existing cache (even if the query is sent for the first time), the G_{cache} of the host is used to detect discrepancies. The cache has pointers to all the vertices and edges in the graphs it contains. This enables searches of the union of all the graphs in the cache.

Merging a new response $G_{response}$ with the existing cache G_{cache} without redundancy starts by identifying the intersection of sets G_{cache} and $G_{response}$. One approach for computing $G_{cache} \cap G_{response}$ is to construct a bijection between the graphs using McKay's algorithm [15]. However, this requires the construction of a canonical form that requires $O(2^n)$ time, where $n = |G_{cache} \cup G_{response}|$. Therefore, an alternative approach that leverages provenance metadata represented as a property graph is employed.

All vertices and edges have content-based identifiers as described in Section 2. Specifically, the identifier of a vertex is computed by hashing the catenation of the sorted set of annotations associated with the vertex. In the case of an edge, the hash takes as input the identifiers of the two endpoint vertices and the annotations associated with the edge; the resulting hash is the identifier of the edge. In this setting, the problem is reduced to sorting the identifiers of the vertices and edges of each graph. The intersection of the two graphs contains the elements present in both sorted sets. The operations can be performed in linear time by traversing the two sorted sets in lockstep.

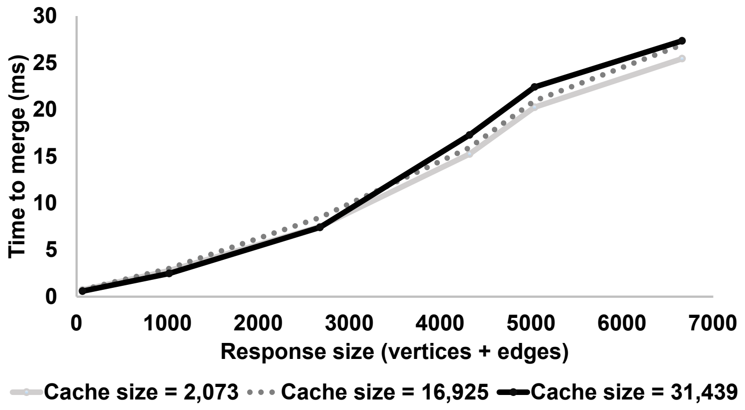

Figure 3. Impact of response size on merging time in the graph cache.

Figure 3 shows the linear relation between response size and time taken to merge responses into a fixed-size cache for varying cache sizes (numbers of vertices and edges). This is significant because larger cache sizes do not increase the merge time significantly.

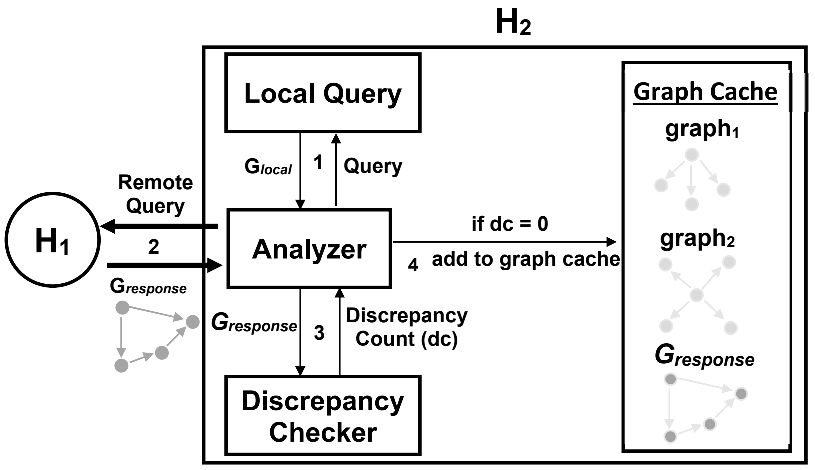

Figure 4. Querying and discrepancy detection workflow.

Figure 4 shows the querying and discrepancy detection workflow. The analyzer module in host H_2 acts as a query manager:

- The analyzer module receives a query from a user, sends it to the local query module and receives the response G_{local}.

- If the local query module indicates that a remote host needs to be consulted, the analyzer prepares a remote query and sends it to H_1, which responds with a provenance graph $G_{response}$.
- The analyzer checks the signature of $G_{response}$. If the signature is valid, it forwards $G_{response}$ to the discrepancy checker, which returns the discrepancy count dc.
- If the discrepancy dc is zero, $G_{response}$ is added to the graph cache G_{cache} and is shown to the user along with G_{local}. Otherwise, the discrepancy checker reports dc to the analyzer. It is important to note that the discrepancy count is proportional to the number of different discrepancies detected.

5.1 Eviction Policy

The cumulative metadata can grow very large in an environment when whole-network provenance is being collected, For example, during the DARPA Transparent Computing engagements [5], terabytes of provenance records were collected from a small network. If all provenance queries are resolved across a distributed system and their responses are cached at the intermediate and original querying hosts, the metadata would increase monotonically with a large storage overhead.

One way to keep the cache size from growing arbitrarily is to implement an eviction policy. Such a policy can be framed at the granularity of individual graph elements, similar to previous approaches for distributed provenance cache management [10]. However, this leads to two shortcomings. First, if individual vertices and edges are removed from a provenance graph, the graph may become disconnected. This would violate the property that a provenance graph obtained from a lineage query is a single, connected graph (as described in Section 4). Second, evicting an element from the intersection of a new response and previously-cached responses is indistinguishable from the case where the response contains a discrepancy.

If an old response G exists such that $G \subset G_{cache}$, then the host can discard G without loss of information. However this requires old responses to be evaluated periodically, which would increase the time complexity of cache management. Instead, a provenance-aware first-in first-out (FIFO) eviction policy is employed that removes the complete response graph components from the cache instead of individual graph elements.

Measurements of the impact of the eviction policy on the number of detected discrepancies shows a clear trade-off between the cache size and effectiveness of discrepancy detection. This was accomplished by executing a series of queries $q_1, q_2, \ldots, q_n$ and adding their responses

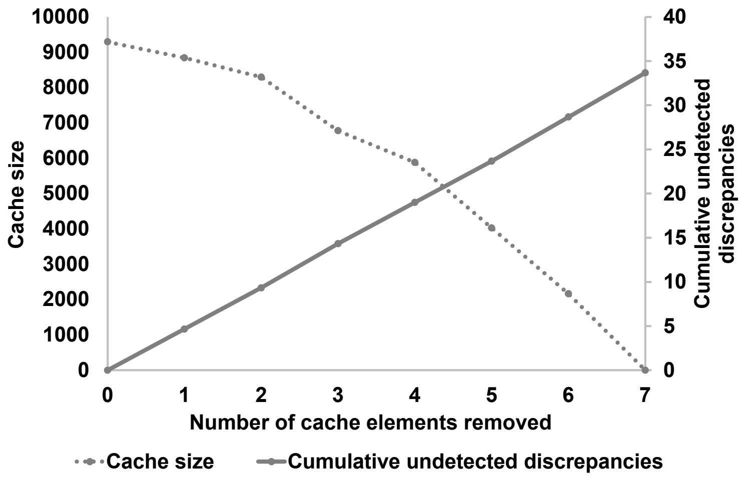

Figure 5. Eviction policy impact on number of undetected discrepancies.

$r_1, r_2, \ldots, r_n$ to the cache in the same order. The responses were removed one by one to reduce the cache size. However, before and after removing a response r_i, query q_i was sent again and a fixed number of edges and vertices in the response was deleted. This enabled the measurement of the number of discrepancies that went undetected when r_i was absent from the cache.

Figure 5 shows the impact of the FIFO eviction policy on cache size (number of vertices and edges) and the number of discrepancies that go undetected. In the beginning, the cache contains seven graph responses and there is no eviction. As a result, the number of detected discrepancies at the time is also the maximum. As cache elements are removed one by one, the cache size decreases and the number of discrepancies that go undetected increase. When all seven graphs in the cache are removed, no discrepancy is detected by the algorithm because there is nothing left in the cache to compare with the new query response.

5.2 Graph Storage

A provenance graph can be stored in any way that a directed graph with annotations is stored. For example, SPADE [12] provides the Postgres relational database, Neo4j graph database and Apache Kafka streams as storage options. While storing the entire graph provides the most information to detect a discrepancy, the storage required grows rapidly. In fact, when using TRACE data sets, the storage required grew by approximately 1 GB per hour [13].

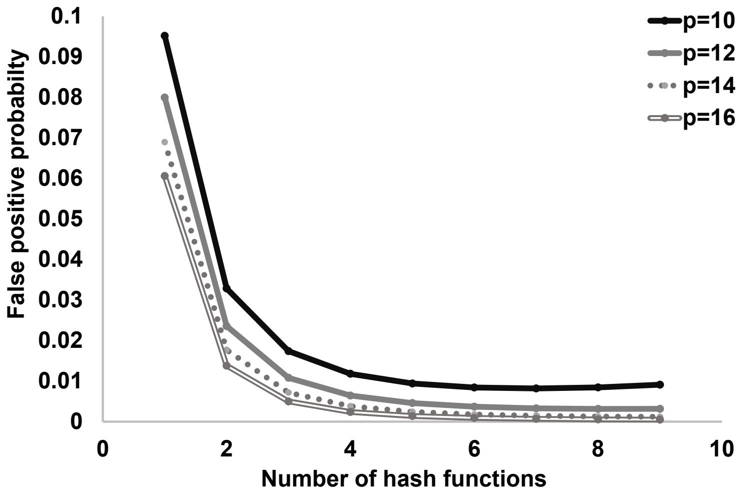

Figure 6. False positive rates for varying numbers of hash functions.

The rapid growth not only consumes storage, but also network bandwidth. If the cache is built by periodically circulating the provenance graph from each host, the rapid metadata growth would burden the storage of every host and every connection between hosts in the network.

Instead of storing the entire graph, a Bloom filter may be used to store the vertex and edge identifiers. Discrepancy detection relies on membership tests, that is, checking if a certain vertex or edge is in a particular provenance graph. A Bloom filter offers a trade-off between space (and bandwidth) and the false positive response rate.

Figure 6 shows how the probability of returning a false positive in the membership test changes as more hash functions are employed for varying p, which is the ratio between the size of the Bloom filter m and the number of elements (vertices or edges) n. As more hash functions are used, the false positive rate quickly decreases and then plateaus.

If the host periodically circulates the changes in the provenance graph to update the whole-network provenance stored at each host, the Bloom filter could contain only the newly added vertices and edges created since the last Bloom filter was sent. Each host could keep the Bloom filters separately in its cache or merge a subset. Merging the Bloom filters saves space and also reduces the time complexity of the membership test in discrepancy detection.

Figure 7 shows that merging Bloom filters increases the false positive rate. When the ratio of the Bloom filter size to the number of elements p is 100 and nine hash functions are used, merging ten Bloom filters

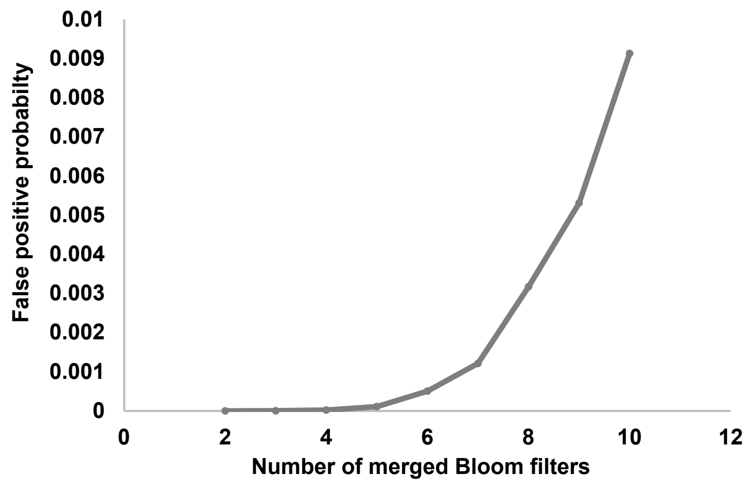

Figure 7. False positive rate using merged Bloom filters.

resulted in a 1% false positive rate. The ratio p and number of hash functions k can be selected to minimize the false positive rate in the merged Bloom filters.

6. Discrepancy Detection

A provenance discrepancy is defined as the difference between truthful provenance and a response received by a querying or intermediate host. A host may have experienced an overwhelming workload and omitted some provenance metadata or it may have replayed an old response from another host. Upon getting such a response, the receiving host could detect a discrepancy if the discrepancy occurred in any of the previously-received responses.

Before the query host uses the provenance metadata it received from upstream hosts, it has to verify the authenticity and integrity of the received data. It is assumed that every host has the public keys of other remote hosts and that the response from each host is digitally signed using the private key of the host. The query nodes can check cryptographic signatures to detect if the intermediate nodes modified the metadata from upstream nodes before forwarding them to the downstream nodes. However, when any host fabricates its own provenance metadata, it can also provide a proper signature for the fraudulent metadata. The query host would not be able to detect this attack using the cryptographic signatures. Similarly, when an intermediate host replays a previously-received response from its upstream hosts, the cryptographic signature would still verify normally and the query host would not be able to detect that the response is outdated.

Whole-network provenance is typically inferred based on records originating from the kernel; this is due to multiple reasons, including the global view available and the higher bar for tampering. Consequently, in practice, the primary threat to the soundness of the provenance being reported is the loss of records along the data path from the occurrence of the relevant event to persistent storage. A missing record can translate to a variety of effects in the provenance stream, the simplest of which is a missing instance of a relation.

6.1 Threat Model

The threat model comprises two attacks on the desired properties. Note that any provenance metadata given as a response to a remote query could be affected by one or more of these attacks.

6.1.1 Omission Attack on Integrity. In this attack, a source or intermediate host provides fabricated metadata by deleting or modifying its own provenance metadata. The fabrication may be intentional or it may be due to network fluctuations, errors or software bugs.

As an example, assume that H_1 has experienced an overwhelming workload and failed to record some of its own provenance metadata in persistent storage. Also, H_1 may have previously provided a truthful response to a query from H_2. The result would be equivalent to modifying or deleting an element from a truthful provenance graph. The discrepancy detection approach does not require that the same query that gave rise to the fraudulent response had to be performed earlier. The discrepancy would be detected as long as the deletion in the fraudulent response is in the portion that overlaps with an earlier truthful response to a query.

6.1.2 Replay Attack on Freshness. In this attack, an intermediate host resends (replays) a previously-received response to a downstream host containing outdated provenance metadata from an upstream host. For example, H_1 in Figure 1 may not forward $query_2$ to H_0 and repeat an old response from H_0 to H_2 to save computing and network resources. Note that H_1 cannot modify or produce a fraudulent $result_0$ without H_2 detecting it because of the cryptographic signature.

The threat model does not include the case where a remote host only adds fraudulent data to the authentic provenance metadata in a monotonically increasing manner. Consider a case where a remote host adds the same fraudulent provenance metadata in addition to the authentic data to all the responses it generates. In this case, all the other hosts would not be able to tell if the remote host is lying because the cryptographic signature would be valid and all the responses would be consistent with each other. From a user's standpoint, there is no difference between such an addition and a valid insertion to the provenance graph.

6.2 Omission Attack Detection

A discrepancy in a whole-network provenance graph G' is defined as an invalid modification of the topology (modifying or deleting a vertex or edge) or schema specifications (changing the annotations of a vertex or edge) of G, where G is a truthful response to a provenance query. It is important to note that, when an adversary changes the schema specifications, it appears as if the adversary deleted a vertex or an edge in G and added a new one to it. In other words, all discrepancies appear as deletions and/or additions of vertices and edges in a whole-network provenance graph.

The proposed scheme detects if any vertices and/or edges present in the previous responses (i.e., G_{cache}) are deleted in a later response (i.e., $G_{response}$). More specifically, Algorithm 1 computes the discrepancy count dc defined as the number of vertices and edges missing from $G_{response}$, number of dangling edges (both incident vertices are not in $G_{response}$) and number of dangling vertices (no incoming edges in $G_{response}$). More formally, if $G_{cache} = (V_c, E_c)$ and $G_{response} = (V_r, E_r)$, then the disrepancy count is given by:

$$\begin{aligned} dc \;=\; & |V_c \setminus V_r| + |E_c \setminus E_r| + |\{e = (X,Y) \in E_r | (X \notin V_r) \vee (Y \notin V_r)\}| \\ & + |\{X \in V_r | \nexists (Y,X) \in E_r, \forall Y\}| \end{aligned}$$

6.3 Empirical Analysis

The empirical analysis employed a small experimental network comprising two hosts, H_1 and H_2, with H_1 the source host of file f. File f was transferred via `scp` to H_2, which generated a provenance trace. During the transfer, both the hosts constructed provenance graphs of their internal system activity. H_1's provenance graph comprised 36,612 vertices and 126,999 edges whereas H_2's provenance graph comprised 128,119 vertices and 446,098 edges.

To track the descendants of file f, H_1 sent a lineage query q with a maximum lineage depth of eight. It originated from f and traveled

Algorithm 1: Discrepancy detection algorithm.

```
Data: $G_{cache} = \cup_{\forall t<t_r} G(t)$, $G_{response} = G(t_r)$: Provenance graphs;
$d_{max}(G_{response})$: Maximum lineage query depth of $G_{response}$ computed via
breadth-first search
Result: $dc$: Discrepancy count in $G_{response}$
1  $C \leftarrow 0$
   /* Count missing vertices */
2  for each vertex $X \in G_{cache}$ and $X \notin G_{response}$ do
3      if $d(X) < d_{max}(G_{response})$ then
4          $C \leftarrow C + 1$
5      end
6  end
   /* Count missing edges */
7  for each edge $e = (X, Y)$ such that $e \in G_{cache}$ and $e \notin G_{response}$ do
8      if $d(X) < d_{max}(G_{response})$ then
9          $C \leftarrow C + 1$
10     end
11 end
   /* Count dangling edges */
12 for each edge $e = (X, Y) \in G_{response}$ do
13     if $X \notin G_{response}$ or $Y \notin G_{response}$ then
14         $C \leftarrow C + 1$
15     end
16 end
   /* Count dangling vertices */
17 for each vertex $X \in G_{response}$ do
18     if $\nexists\ e = (A, B)$ such that $X = B$ then
19         $C \leftarrow C + 1$
20     end
21 end
22 return $dc$
```

to H_2, which returned the response graph $G_{response}$. Next, the algorithm executed on H_1 and returned the discrepancy count by comparing $G_{response}$ with G_{cache}. The final result to the query q included graphs G_{local} and $G_{response}$. G_{local} comprised 2,283 vertices and 3,740 edges from H_1 whereas $G_{response}$ comprised 327 vertices and 404 edges from H_2.

The query execution time was measured as starting when H_1 sent q until H_1 completely executed the discrepancy detection algorithm. To evaluate the algorithm overhead at the query host H_1, the baseline performance was first established by measuring the query execution time for q without the detection algorithm in place. Several independent iterations of the query q were executed with the detection algorithm, each with varying numbers of modifications to the response graph. The

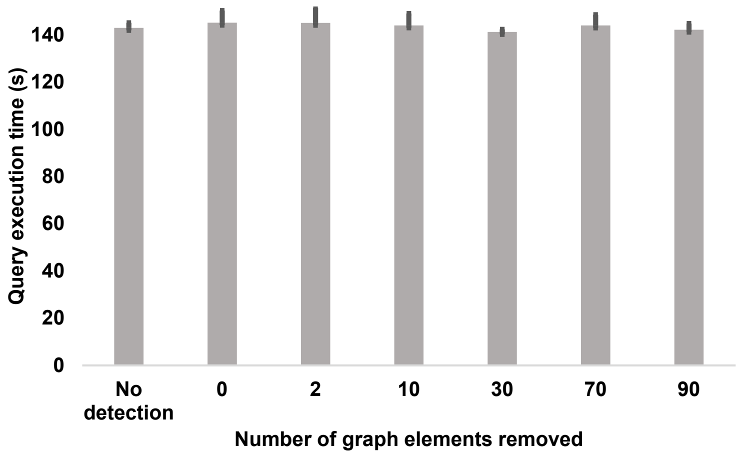

Figure 8. Query execution time with discrepancy detection for a lineage query.

modifications were induced by dropping the same number of vertices and edges from $G_{response}$, where the number of dropped vertices ranged from zero to 90.

Figure 8 shows that the query execution time did not change significantly with the number of eliminated graph elements. No detection refers to the case when discrepancy checking was not executed. The algorithm imposed less than 0.4% overhead over the baseline. In fact, the query execution time without the algorithm (no detection) is comparable to the case where 90 vertices and 90 edges were removed. This is because most of the query execution time is attributed to the network latency between hosts.

6.4 Replay Attack Detection

The query and response structures were modified to include unique, unpredictable nonces chosen by the query host. When the query host issues a remote query, it sends a new nonce along with the query. Malicious intermediate hosts may choose not to forward the entire query and cause the query host to time out, but they cannot fabricate a response from upstream hosts with the matching nonce. The upstream and source hosts respond with their own provenance metadata along with a nonce and signature computed over their provenance metadata and nonces. The downstream and query hosts discard responses that do not contain valid cryptographic signatures for the (query, nonce) pairs. This can increase the overhead at the query host because it needs to keep track of the (query, nonce) pair until it receives all the responses. However, a timeout was introduced at the query host so it would discard the (query, nonce) pair after waiting for a certain time period.

Note that this mechanism does not interfere with the ability of the query host to use its own cache to answer a remote query, but it clearly does not allow an intermediate host to reuse responses from its own cache because the nonce would not match. The querying host may decide to send a remote query with a lower depth value to check if there is a change in the provenance metadata before it sends a remote query with the maximum depth necessary. If there is no change in the provenance metadata in nearby hosts, the querying host may use its own cache to answer the lineage or path query.

6.5 Correctness Proofs

The correctness of the discrepancy detection algorithm is proved using induction over the size of an isolated discrepancy. An isolated discrepancy is defined as a maximal connected subgraph of vertices and edges contained in the previous response G_{cache} but missing in $G_{response}$. In general, there may be multiple isolated discrepancies in $G_{response}$.

Theorem 1: *Algorithm 1 detects an isolated discrepancy of any size.*
Proof: *Proof by induction on the size of discrepancy k.*

- *Base Step: $k = 1$. If the discrepancy is a single vertex from G_{cache} missing from $G_{response}$, then Lines 2-6 would detect the discrepancy. If the discrepancy is a single edge from G_{cache} missing from $G_{response}$, then Lines 7-11 would detect the discrepancy. Thus, any discrepancy of size $k = 1$ is detected by Algorithm 1.*

- *Inductive Step: Assume that Algorithm 1 detects an isolated discrepancy of size up to k. An isolated discrepancy of size $k + 1$ is the union of an isolated discrepancy of size k and an additional vertex/edge connected from the discrepancy of size k being deleted from $G_{response}$.*

  There are three possible cases for the additionally-deleted vertex or edge – vertex, incoming edge and outgoing edge:

  – *Vertex: This is the case where an additional vertex connected to an edge in the discrepancy of size k is deleted. If the vertex has an incident edge in $G_{response}$, then Lines 12–16 of the algorithm would detect that the vertex is missing. If the vertex does not have an incident edge in $G_{response}$, by the definition of an isolated discrepancy, all its incident edges are in the discrepancy of size k. If an incident edge of the vertex in G_{cache}*

is not in the discrepancy of size k and not in $G_{response}$, then the size of the isolated discrepancy would be of size $k+2$ (= k + missing vertex + missing incident edge), not $k+1$. While the newly deleted vertex does not increase dc, the discrepancy of size k is detected due to the industive hypothesis that the algorithm detects any isolated discrepancy of size k. Thus, the algorithm detects the discrepancy of size $k+1$.

- *Incoming Edge: This is the case where an additional incoming edge to a vertex in the discrepancy of size k is deleted. The other vertex x associated with the edge must be in $G_{response}$ and in G_{cache}, so Lines 2–6 of the algorithm would detect the discrepancy and increase dc.*
- *Outgoing Edge: This is the case where an additional outgoing edge from a vertex in the discrepancy of size k is deleted. The other vertex y associated with this edge must be in $G_{response}$, and is detected as a discrepancy in Lines 17–21 unless there is another edge that goes to vertex y. If there is another edge to y, then the algorithm would still detect the discrepancy based on the discrepancy of size k, but would not return a higher discrepancy count dc.* □

Theorem 2: *Algorithm 1 detects any number of isolated discrepancies of any size.*
Proof: *Each isolated discrepancy is connected to a legitimate vertex or edge in the dependency graph. If it is a vertex, then the vertex would miss a path from/to other parts of the graph and the algorithm would detect it. If it is an edge, then the edge would miss a vertex and become a dangling edge. Lines 12–16 in the algorithm specifically detect this discrepancy.* □

6.6 Probabilistic Analysis

Algorithm 1 detects any discrepancy that occurs in $G_{response} \cap G_{cache}$ and rejects $G_{response}$. Thus, for any $G_{response}$ with a discrepancy to bypass the detection algorithm, all the discrepancies such as missing vertices and edges should occur in $G_{response} \backslash G_{cache}$. Assume that the size of $G_{response}$ is s (equal to the number of vertices and edges in $G_{response}$), and the probability that any vertex or edge is removed from $G_{response}$ is p_{Ω}. Then, the expected number of missing vertices and edges from $G_{response}$ is $p_{\Omega} \times s$. When the probability that any vertex or edge in

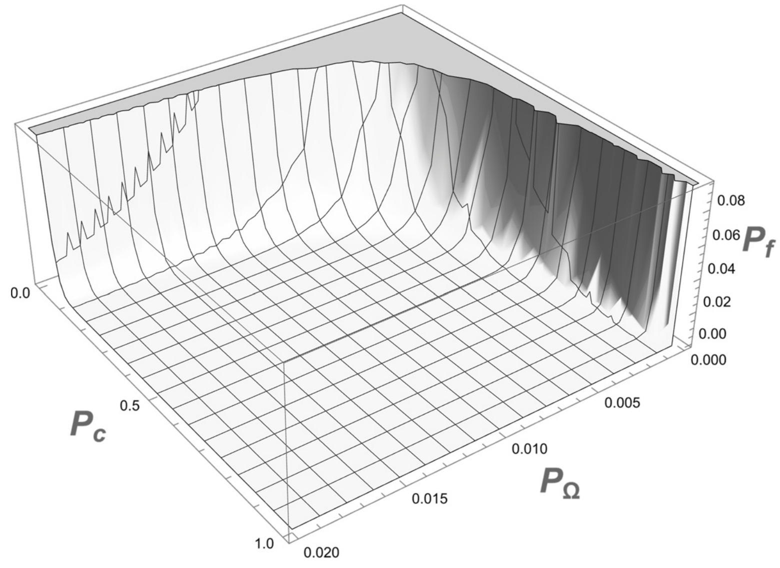

Figure 9. Probability of discrepancy detection failure.

$G_{response}$ is already in G_{cache} is equal to p_c, the probability p_f of all the missing vertices and edges occurring in $G_{response} \setminus G_{cache}$ is given by:

$$p_f = \frac{\binom{(1-p_c)*s}{p_\Omega * s}}{\binom{s}{p_\Omega * s}}$$

The probability p_f is the upper bound of the algorithm not detecting any discrepancy and accepting $G_{response}$ with missing vertices and/or edges. The algorithm would detect that $G_{response}$ is missing vertices or edges if there are any dangling vertices and edges, and the probability of all the missing vertices and edges being arranged such that there are no dangling vertices and edges is strictly less than one.

Figure 9 shows how the probability p_f changes when p_Ω ranges from 0 to 0.2 and p_c ranges from 0 to 1. When the system launches, there is little overlap between G_{cache} and $G_{response}$, and p_c is close to zero. As G_{cache} builds up, the overlap increases and p_f decreases as well.

Figure 9 also shows that p_f quickly decreases as p_c increases. Also, as p_Ω increases, it is less likely that all the missing vertices and edges would be in $G_{response} \setminus G_{cache}$; thus, p_f decreases.

Figure 9 also shows that probability p_f quickly decreases as p_Ω increases. For example, when p_Ω is 0.005 and p_c is 0.5, p_f is 0. In other words, when there is 50% overlap between G_{cache} and $G_{response}$, Algorithm 1 would detect that $G_{response}$ is missing 0.5% or more vertices and/or edges. Once the overlap increases to 90%, the algorithm would detect $G_{response}$ is missing 0.1% or more vertices and/or edges.

7. Related Work

Several systems offer metadata or provenance management in distributed environments. FusionFS [25] implements distributed file metadata management based on distributed hash tables. ExSPAN [28] is a generic framework for provenance management that employs the distributed query processing capabilities of declarative networks. It extends a traditional relational database management system for provenance collection and retrieval.

Several systems have been used to track the provenance of scientific applications. The open-source workflow management system Taverna [24] enables biologists to add application-level annotations of data provenance. CMCS [16] applies an informatics-based approach for synthesizing multi-scale chemistry information. ESSW [7] is a metadata storage system for earth scientists.

None of the systems mentioned above address the problem of discrepancy detection in distributed environments. In many cases, they are customized to specific application domains. In contrast, SPADE adopts a domain-agnostic approach. This enables the enhancements described in this chapter to be utilized in a wide range of settings.

Providing security for data provenance in distributed environments has also been discussed in the literature. Wang et al. [22] proposed a public-key linked chain provenance framework to protect provenance metadata. The Mendel protocol incorporates a three-pronged strategy that combines signature verification and cryptographic ordering witnesses to perform provenance verification in distributed environments [8]. In decentralized settings, where each host signs its own responses, such cryptographic protections cannot address the concerns raised in this chapter.

Some systems focus on specific security aspects that relate to their target domains. Cheney [4] outlined a formal model of security properties for provenance. The Trio system enables the source of uncertainty to be traced after tracking the provenance of database elements [23]. TAP [26] and DTaP [27] are time-aware provenance models that explicitly represent time, distributed state and state change in order to secure queries in the absence of trusted nodes in a network. Liao and Squicciarini [14] developed a system that identifies anomalies in the MapReduce framework based on provenance information collected from within the framework.

Other systems have used provenance metadata in critical infrastructure. Sultana et al. [18] demonstrated that provenance can be used for data integrity in large-scale sensor networks, where the collected data supports decision making in critical infrastructure assets. When a base station knows the communication paths in the network, the complete path of any data sent from a source sensor to the base station can be encoded in a Bloom filter. This enables the base station to compare the provenance to the known path. Each datum from the source comes with a sequence number. The base station can tell if a packet is missing from the skipped sequence number and identify malicious node(s) using the path information of the next packet.

Provenance has also been used in intrusion detection. Hassan et al. [21] employed a provenance graph in cluster auditing to process system audit information in an efficient manner. The provenance graph generated from system audit information is used to monitor hosts in a cluster during normal operation and also to reconstruct attacks in forensic investigations. Berrada et al. [2] evaluated five categories of unsupervised anomaly detection algorithms on provenance data collected via DARPA's Transparent Computing Program, which includes advanced persistent threats.

However, none of the above approaches detect the types of discrepancies addressed by Algorithm 1 in this chapter. The approach is prototyped in cyber infrastructure that is available for researchers to modify and deploy in their own environments. Additionally, code for the core functionality, such as caching and discrepancy detection, is available at the SPADE open-source repository. In contrast, the implementations of many other systems for securing provenance have not been released to the research community.

8. Conclusions

This chapter has introduced the notion of whole-network provenance that represents dependency metadata within and across hosts in distributed systems. First, it shows how the slice of whole-network provenance related to a local artifact or process is reconstructed by issuing specific distributed queries. Next, it demonstrates how each host can build a cache of provenance records received in response to queries made to remote hosts. Finally, it describes an approach that detects discrepancies in provenance metadata distributed across several hosts by comparing previously-cached responses against new responses. The fact that provenance grows monotonically is leveraged to detect a discrepancy in the event that a later response is missing an element in an earlier response.

The DISTDET provenance-based attack detection system has been installed on more than 22,000 hosts at over 50 industrial customers [6]. Future research will focus on deploying the proposed system in real network environments.

The views and conclusions in this chapter are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, expressed or implied, of the National Science Foundation or the U.S. Government.

8.0.1 Acknowledgment. This research was supported by the National Science Foundation under Grant no. ACI 1547467.

References

[1] R. Ahmad, E. Jung, C. de Senne Garcia, H. Irshad and A. Gehani, Discrepancy detection in whole-network provenance, *Proceedings of the Twelfth USENIX Conference on Theory and Practice of Provenance*, article no. 5, 2020.

[2] G. Berrada, J. Cheney, S. Benabderrahmane, W. Maxwell, H. Mookherjee, A. Theriault and R. Wright, A baseline for unsupervised advanced persistent threat detection in system-level provenance, *Future Generation Computer Systems*, vol. 108, pp. 401–413, 2020.

[3] C. Catlett, The philosophy of TeraGrid: Building an open, extensible, distributed terascale facility, *Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[4] J. Cheney, A formal framework for provenance security, *Proceedings of the Twenty-Fourth IEEE Computer Security Foundations Symposium*, pp. 281–293, 2011.

[5] Defense Advanced Reseach Projects Agency, Transparent Computing (archived), Arlington, Virginia (`darpa.mil/program/transparent-computing`), 2023.

[6] F. Dong, L. Wang, X. Nie, F. Shao, H. Wang, D. Li, X. Luo and X. Xiao, DISTDET: A cost-effective distributed cyber threat detection system, *Proceedings of the Thirty-Second USENIX Security Symposium*, pp. 6575–6592, 2023.

[7] J. Frew and R. Bose, Earth System Science Workbench: A data management infrastructure for earth science products, *Proceedings of the Thirteenth International Conference on Scientific and Statistical Database Management*, pp. 180–189, 2001.

[8] A. Gehani and M. Kim, Mendel: Efficiently verifying the lineage of data modified in multiple trust domains, *Proceedings of the Nineteenth ACM International Symposium on High Performance Distributed Computing*, pp. 227–239, 2010.

[9] A. Gehani, M. Kim and T. Malik, Efficient querying of distributed provenance stores, *Proceedings of the Nineteenth ACM International Symposium on High Performance Distributed Computing*, pp. 613–621, 2010.

[10] A. Gehani, M. Kim and J. Zhang, Steps toward managing lineage metadata in grid clusters, *Proceedings of the First Workshop on the Theory and Practice of Provenance*, article no. 7, 2009.

[11] A. Gehani and U. Lindqvist, Bonsai: Balanced lineage authentication, *Proceedings of the Twenty-Third Annual Computer Security Applications Conference*, pp. 363–373, 2007.

[12] A. Gehani and D. Tariq, SPADE: Support for provenance auditing in distributed environments, *Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 101–120, 2012.

[13] H. Irshad, G. Ciocarlie, A. Gehani, V. Yegneswaran, K. Lee, J. Patel, S. Jha, Y. Kwon, D. Xu and X. Zhang, TRACE: Enterprise-wide provenance tracking for real-time APT detection, *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4363–4376, 2021.

[14] C. Liao and A. Squicciarini, Towards provenance-based anomaly detection in MapReduce, *Proceedings of the Fifteenth IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 647–656, 2015.

[15] B. McKay, Computing automorphisms and canonical labelings of graphs, in *Combinatorial Mathematics*, D. Holton and J. Seberry (Eds.), Springer, Berlin Heidelberg, Germany, pp. 223–232, 1978.

[16] C. Pancerella, J. Hewson, W. Koegler, D. Leahy, M. Lee, L. Rahn, C. Yang, J. Myers, B. Didier, R. McCoy, K. Schuchardt, E. Stephan, T. Windus, K. Amin, S. Bittner, C. Lansing, M. Minkoff, S. Nijsure, G. von Laszewski, R. Pinzon, B. Ruscic, A. Wagner, B. Wang, W. Pitz, Y. Ho, D. Montoya, L. Xu, T. Allison, W. Green and M.

Frenklach, Metadata in the Collaboratory for Multi-Scale Chemical Sciences, *Proceedings of the International Conference on Dublin Core and Metadata Applications*, pp. 121–129, 2003.

[17] D. Pohly, S. McLaughlin, P. McDaniel and K. Butler, Hi-Fi: Collecting high-fidelity whole-system provenance, *Proceedings of the Twenty-Eighth Annual Computer Security Applications Conference*, pp. 259–268, 2012.

[18] S. Sultana, G. Ghinita, E. Bertino and M. Shehab, A lightweight secure scheme for detecting provenance forgery and packet drop attacks in wireless sensor networks, *IEEE Transactions on Dependable and Secure Computing*, vol. 12(3), pp. 256–269, 2015.

[19] Y. Tan, R. Ko and G. Holmes, Security and data accountability in distributed systems: A provenance survey, *Proceedings of the Tenth IEEE International Conference on Embedded and Ubiquitous Computing*, pp. 1571–1578, 2013.

[20] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. Peterson, R. Roskies, J. Scott and N. Wilkins-Diehr, XSEDE: Accelerating scientific discovery, *Computing in Science and Engineering*, vol. 16(5), pp. 62–74, 2014.

[21] W. Ul Hassan, M. Lemay, N. Aguse, A. Bates and T. Moyer, Towards scalable cluster auditing through grammatical inference over provenance graphs, *Proceedings of the Twenty-Fifth Network and Distributed Systems Security Symposium*, 2018.

[22] X. Wang, K. Zeng, K. Govindan and P. Mohapatra, Chaining for securing data provenance in distributed information networks, *Proceedings of the IEEE Military Communications Conference*, 2012.

[23] J. Widom, Trio: A system for integrated management of data, accuracy and lineage, *Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, pp. 262–276, 2005.

[24] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. Balcazar Vargas, S. Sufi and C. Goble, The Taverna workflow suite: Designing and executing workflows of web services on the desktop, web or in the cloud, *Nucleic Acids Research*, vol. 41(WS), pp. W557–W561, 2013.

[25] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross and I. Raicu, FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing

systems, *Proceedings of the Second IEEE International Conference on Big Data*, pp. 61–70, 2014.

[26] W. Zhou, L. Ding, A. Haeberlen, Z. Ives and B. Loo, TAP: Time-aware provenance for distributed systems, *Proceedings of the Third USENIX Workshop on the Theory and Practice of Provenance*, 2011.

[27] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. Loo and M. Sherr, Distributed time-aware provenance, *Proceedings of the VLDB Endowment*, vol. 6(2), pp. 49–60, 2012.

[28] W. Zhou, M. Sherr, T. Tao, X. Li, B. Loo and Y. Mao, Efficient querying and maintenance of network provenance at Internet-scale, *Proceedings of the Twenty-Ninth ACM SIGMOD International Conference on Management of Data*, pp. 615–626, 2010.