Discrepancy Detection in Whole Network Provenance

Raza Ahmad - SRI *Eunjin Jung - University of San Francisco*
Carolina de Senne Garcia - Ecole Polytechnique *Hassaan Irshad - SRI* *Ashish Gehani - SRI*

Abstract

Data provenance describes the origins of a digital object. Such information is particularly useful when analyzing distributed workflows because extant tools, such as debuggers and application profilers, do not support tracing through heterogeneous runtimes that span multiple hosts. In decentralized systems, each host maintains the authoritative record of its own activity, represented as a dependency graph. Reconstructing the provenance of an object may involve the assembly of subgraphs from multiple, independently administered hosts. We term the collection of host-specific dependencies coupled with cross-host flows *whole-network provenance*. Such information can grow to terabytes for a small network. Aspects of distributed querying, caching, and response discrepancy detection that are specific to provenance are described and analyzed.

1 Introduction

Provenance collection and analysis is helpful for studying distributed applications that coordinate data and workflow across interconnected hosts, and then combine the results [9]. Consortia of institutions that share data and resources for large-scale tasks, such as TeraGrid [1] and XSEDE [10].

Provenance metadata from these systems may span many hosts over multiple administrative domains connected via the Internet. We define *whole-network provenance* to be the metadata that describes the relationships between agents, processes, and data artifacts on individual hosts coupled with the set of distributed data flows between processes on these hosts.

Each host in a network can also store provenance metadata from remote hosts to create its own *cache*. In a decentralized query approach, each host receives responses from remote hosts for its own query, but also forwards responses for queries from other hosts too. All or any subset of these responses can be stored in a local storage to build the cache. When the network is too slow or too expensive, the host may run a provenance query on its own cache to get a preliminary idea.

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 Some hosts may have buggy software, some may send outdated data, some may suffer from network fluctuations, and some may be malicious. We define *provenance discrepancy* 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 fewer. We propose to take advantage of this “append-only" nature of provenance metadata in the discrepancy detection process and report a discrepancy whenever a query response is missing an element from the previously known provenance metadata in cache.

The ability to detect discrepancies from the missing graph elements is important in several real-life applications. These applications contain scenarios which motivate altering the provenance data after an incident has occurred. Some of the previously described scenarios [2] include (i) a product failure that exposes a company to legal liabilities in case of a forensic analysis, (ii) a legal battle over patent infringement by a company to deny prior possession of references, (iii) an accident as a result of a computational error, or (iv) a claim of credit for a discovery after learning of a competitor's result. Data modifications manifest as the deletion of an old element and insertion of a new one. Discrepancies caused by additions are not distinguishable from legitimate provenance growth.

The rest of the paper is organized as follows. Section 2 describes related work and the semantics of the provenance metadata management system. Section 3 defines whole-network provenance. Section 4 elaborates on the mechanism for querying provenance metadata in a distributed environment. Section 5 explains the caching mechanism of provenance metadata from remote hosts. Section 6 explains the threat model and then presents our approach to detect two different attacks on distributed provenance. Finally, we conclude in Section 7.

2 Background

Several systems offer provenance management for different domains in a distributed environment, some of which are discussed and compared in [2]. Mendel [2] is a protocol with a three-pronged strategy that combines signature verification and cryptographic ordering witnesses to perform provenance verification in distributed environments. Wang *et al.* proposed a public-key linked chain provenance framework to protect the provenance metadata [11]. More recently, Liao and Squicciarini [6] developed a system that identifies anomalies in the MapReduce framework based on provenance information collected from within the framework. We utilize open-source middleware SPADE [5] in this paper. 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 data provenance* [8]. However, our ideas apply to any provenance management framework that supports decentralized operation.

Let a provenance graph be $G(V,E)$, containing a set of vertices V and edges E, where E connects the elements of V. Each vertex $v \in V$ corresponds to an agent, a process, or an artifact that is the subject or object of an *operation*. Each vertex is characterized by a unique key-value set of annotations $A(v)$: $A(v) = \{a_1, a_2, a_3, \cdots, a_n\}$ where $a_i = \langle key_i : value_i \rangle$. For example, a vertex representing an operating system process would contain annotations like $\langle pid : 2 \rangle$, $\langle user : root \rangle$, $\langle time : t_1 \rangle$. This annotation set is unique as there is only one process with a certain *pid* at any given *time*. Hence, to uniquely identify vertex v with a single attribute, we construct a content-based hash identifier id_v 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 changes to a key-value pair results in changing the vertex into a different one. 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 provenance graph $G(V,E)$ would change to $G(V',E)$ where $V' = V \setminus \{v\} \cup \{v'\}$.

Edges in E are the operations on a pair of vertices and correspond to a directed edge between them, attributing data dependency. For example, a system *read()* call results in an edge from a process vertex to a file vertex and contains annotations like $\langle size : 1024 \rangle, \langle time : 1345121 \rangle$. Each edge $e \in E$ is defined by the two vertices it is incident upon, X and Y, 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 changes to vertices, any change to an annotation in $A(e)$ results in changing the edge, i.e. deleting the original edge and adding a new edge to E.

3 Whole-Network Provenance

We formally define *whole-network provenance* to be 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. Through whole-network provenance graphs, we can reconstruct the provenance of a particular object, starting from one host and tracking back through other relevant hosts.

Let us define the provenance graphs on a host H_i 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 on the network H_i and H_j.

In a centralized strategy, each host uploads its own provenance metadata periodically to a single repository that handles all provenance queries. This approach simplifies coordination between the hosts but suffers from these limitations: i) all hosts in the network are required to periodically send all their provenance metadata to the central repository, even though other hosts may not need much of it, ii) the central repository may become a performance bottleneck, especially in terms of bandwidth as simultaneous uploads from multiple hosts may render the it unavailable for processing queries, and iii) reliability of the whole system decreases since the central repository becomes a single point of failure. An issue such as a data integrity compromise can affect the provenance metadata of the entire network.

We employ an approach that uses 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 can then combine all the responses from remote hosts.

This mechanism provides a scalable approach for whole-network provenance collection as it does not suffer from the issues of a centralized approach discussed above. It also has additional benefits: i) no single host is required to have sufficient resources to maintain complete copies of provenance from all hosts. ii) the amount of data transferred through a single connection is limited to the data required to respond to specific queries. iii) queries about local provenance can always be answered using the host's own store (cache). In the centralized approach, the availability of the provenance is reduced to that of the network connection to the server. iv) individual hosts have the freedom to implement their own data management policies, such as which database to use and how long to retain archival copies.

At the heart of this decentralized metadata collection is a construct called the *network artifact* [3,5]. Its key property is that it can be constructed without any explicit coordination at independent endpoints. In the context of distributed systems, 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 time when the connection was established.

4 Distributed Querying

In a distributed environment, the host which originates the query is responsible for collecting its responses. After resolving the query locally, it contacts remote hosts through network artifacts which subsequently return their results and contact other hosts if required. The final result is then stitched together at the origin host. This approach allows the remote

hosts at the same distance in the network to be contacted in parallel, increasing the running time of distributed querying linearly as the height of the network topology tree increases.

A provenance management system that operates in a distributed environment may collect provenance metadata across several hosts. Two of the most common operations to collect provenance are *lineage* and *path* queries. 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 as 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, we first define the parent graph of v as: $G_P(v) = (P, E)$, where P is a set of vertices such that $\forall p \in P$, there is an edge $e \in E$, and $e = (v, p)$. The result of lineage of v would then be written as:

$$l(v,d) = G_P(v) \cup l(p,d-1) \ \ \forall p \in P$$

$$l(v,0) = v$$

The response to a lineage query is always a connected graph in which the direction of edges represents the information flow. This means that in a graph $G_{response}$ sent in response to a lineage query q from a vertex $v, \forall u \in G_{response}$, there exists a path between any two given 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 asks for the provenance between two objects, and its response is a set of chains from one element to the other. The response to path queries can be constructed by finding the intersection of lineage ancestors query from the sink and lineage descendants query from the source, when getting all paths from a particular source to a sink.

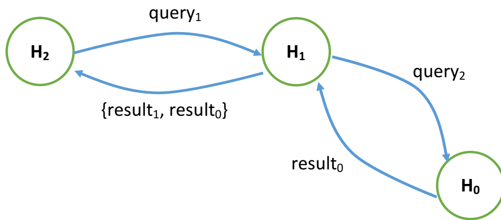

Figure 1: A network of interconnected hosts querying provenance in a distributed manner. H_2 is the query host, H_1 is the intermediate host and H_0 is called the source host.

When a host would like to see the history of an artifact (such as a downloaded file) – that is, where this artifact originated and when and how it was changed before coming to this host – then the host may send a *lineage ancestors* query to its *upstream* hosts. We use the term *query host* to refer to the host from which the lineage inquiry originates. Consider a network of 3 interconnected hosts as shown in Figure 1. H_2 wishes to find the lineage of a file f_2 on H_2, and learns that f_2 was downloaded from H_1. H_2 then becomes the *query host* and sends $query_1$ to *upstream* host H_1 asking for the 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. H_1 then becomes the *intermediate* host and sends $query_2$ to the further upstream host H_0 asking for the provenance metadata related to file f_2. If f_2 originated from H_0, then H_0 is the *source* host and responds with $result_0$.

The *origin* and *type* of a query implicitly define whether we term one host as being *upstream* or *downstream* of another. When a query is performed at H_2 about metadata that originated from H_1, H_1 is referred to as being upstream of H_2 in the context of a *lineage ancestors* query (and its response). Similarly, H_2 is said to be downstream of H_1 in this context. However, the converse holds for a *lineage descendants* query – that is, if the query was performed at host H_1 about metadata that flowed from the host to H_2, then H_2 would be termed as upstream of H_1. (Note that the same pair of hosts can be upstream of each other in the context of different queries.) In the rest of the paper, we use *lineage query* as shorthand for *lineage ancestors* or *lineage descendants* query (with the meaning determined by the context).

5 Caching

We assume 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 [4]. When a host receives a response from an upstream host, whether as a querying host or an intermediate host, the host adds the response to its cache. Each response is stored as a directed graph, so the cache looks like a set of directed graphs.

When a host has a lineage or path query that involves remote hosts, this cache can be also used to get a (potentially outdated) local response when network to other hosts is not reliable or too expensive, and also when low latency is more important than freshness. We call this cache G_{cache} as it contains provenance graphs combined from previously received query responses from other hosts. Figure 2 shows an example graph cache containing previously received query responses, $Graph_i$ and $Graph_j$. Vertices and edges shown in blue are shared by both graphs and stored only once for storage ef-

ficiency. When the response to a query overlaps with the existing cache, even if this is the first time this query is sent, we use its G_{cache} to detect discrepancies due to deniability attack. The cache contains pointers to all vertices and edges in the graphs that it contains. This enables search in the union of all the graphs contained in the cache.

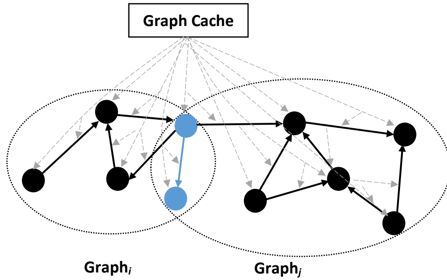

Figure 2: Illustration of cache containing responses to two queries with partial overlap.

Merging a new response $G_{response}$ with the existing cache G_{cache} without redundancy starts with identifying an intersection between G_{cache} and $G_{response}$. One approach to compute $G_{cache} \cap G_{response}$ is to construct a bijection between the graphs using McKay's algorithm [7]. However, this would entail constructing a canonical form that would take $O(2^n)$ time, where $n = |G_{cache} \cup G_{response}|$. We use an alternative approach that leverages provenance metadata represented as a property graph. All vertices and edges have *content-based identifiers*. Typically, this is the hash of its annotations for a vertex, and the hash of its annotations and the endpoint vertex identifiers for an edge, as discussed in Section 2. In this setting, the problem is reduced to sorting each graph's vertex and edge identifiers. The intersection of the two graphs consists of the elements present in both sorted sets. This can be performed in linear time by traversing the two sorted sets in lockstep.

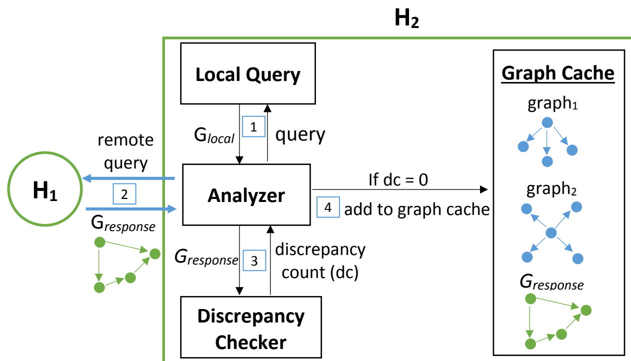

Figure 3: Workflow of querying and discrepancy detection.

The workflow diagram of our system is given in Figure 3. Analyzer module in Host H_2 acts as a query manager. (1) It receives a query from the user, sends it to the local query module and gets a response back, G_{local}. (2) 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, $G_{response}$. (3) The analyzer then checks the signature of $G_{response}$, and if it is valid, forwards $G_{response}$ to the discrepancy checker, which returns C. (4) If C is zero, $G_{response}$ is added to the Graph cache, G_{cache}, and shown to the user together with G_{local}. Otherwise, the discrepancy checker reports C to the analyzer. It is important to note that a high discrepancy count is proportional to the number of different discrepancies that our approach detects.

6 Discrepancy Detection

We define a *provenance discrepancy* as the difference between truthful provenance and a response received by the querying or intermediate host. A host may have experienced overwhelming workload and omitted some provenance metadata or may be replaying an old response from another host. Upon getting such a response, the receiving host could then 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 needs to verify the authenticity and integrity of the received data. We assume 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 that 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 will 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 that originate from the kernel (for multiple reasons, including the global view available and 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 the 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

Our threat model consists of the two following attacks on the desired properties. Note that any provenance metadata given

as a response to a remote query could be affected by more than one of these attacks.

Omission attack on Integrity: A source or intermediate host provides fabricated metadata by deleting or modifying its own provenance metadata. This fabrication may be intentional, or may be due to network fluctuations, errors, or software bugs. For example, H_1 experiences overwhelming workload and failed to store some of the provenance metadata in the persistent storage.

Replay attack on Freshness: 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 its own computing and network resources (Note that H_1 cannot modify or produce fraudulent $result_0$ without H_2 detecting, owing to the cryptographic signature.)

In these examples, H_1 may have previously provided a truthful response to a query from H_2. However, H_1 may have experienced overwhelming workload and failed to record some of its own provenance metadata in the persistent storage and ended up with modifying or deleting an element from a truthful provenance graph. Note that our detection approach does not require that the same query that gave rise to the fraudulent response had to have been performed earlier. As long as the deletion in the fraudulent response is in the portion that overlaps with an earlier truthful response (to any query), the discrepancy will be detected.

Our threat model does not include the case where a remote host only *adds* fraudulent data to the authentic provenance metadata in a monotonic, increasing fashion. Consider a case when 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 cannot tell if this remote host is lying because the cryptographic signature will 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

We define *discrepancy* in a whole-network provenance graph G' 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 attacker changes schema specifications, it appears as if the attacker deleted a vertex or an edge in G and added a new one to it. In other words, all discrepancies appear as deletion and/or addition of vertices and edges in a whole-network provenance graph. Our scheme detects if any vertices and/or edges that were present in the previous responses, i.e. G_{cache}, are deleted in a later response, i.e. $G_{response}$.

6.3 Replay Attack Detection

We changed the query and response structure to include a nonce chosen by the query host. We assume that nonces are not repeated and unpredictable. When the query host sends 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 the nonce and the signature which is computed over their provenance metadata and the nonce. The downstream and query hosts discard any response that does not contain the valid cryptographic signature for the (query, nonce) pair. This can increase the overhead at the query host as it needs to keep track of the (query, nonce) pair until it receives all the responses. However, we added a timeout at the query host so that the query host discards the (query, nonce) pair after waiting for a certain time period.

Note that this does not interfere with the query host's ability to use its own cache to answer a remote query, but clearly does not allow an intermediate host to re-use responses from its own cache as the nonce would not match. The querying host may decide to send a remote query with a smaller depth value to check if there is any 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.

7 Conclusion

We introduced the notion of whole-network provenance that represents dependency metadata both within and across hosts in a distributed system. First, we described how the slice of whole network provenance related to a local artifact or process can be reconstructed by performing specific distributed queries. Next, we outlined how each host can build a cache of provenance records received in response to queries made to remote hosts. Finally, we provide an approach that detects discrepancies in provenance metadata distributed across several hosts by comparing previously cached responses to new ones. The fact that provenance grows monotonically is leveraged to detect a discrepancy in the case that a later response is missing an element in an earlier response.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant ACI-1547467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Charlie Catlett. The philosophy of TeraGrid: Building an open, extensible, distributed terascale facility. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, 2002.

[2] Ashish Gehani and Minyoung Kim. Mendel: Efficiently verifying the lineage of data modified in multiple trust domains. In *19th ACM International Symposium on High Performance Distributed Computing*, pages 227–239. ACM, 2010.

[3] Ashish Gehani, Minyoung Kim, and Tanu Malik. Efficient querying of distributed provenance stores. In *8th Workshop on the Challenges of Large Applications in Distributed Environments*, pages 613–621. ACM, 2010.

[4] Ashish Gehani and Ulf Lindqvist. Bonsai: Balanced lineage authentication. In *23rd Annual Computer Security Applications Conference*, pages 363–373, Dec 2007.

[5] Ashish Gehani and Dawood Tariq. SPADE: Support for provenance auditing in distributed environments. In *13th International Middleware Conference*, pages 101–120. Springer-Verlag New York, Inc., 2012.

[6] Cong Liao and Anna Squicciarini. Towards provenance-based anomaly detection in MapReduce. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 647–656. IEEE, 2015.

[7] Brendan McKay. Computing automorphisms and canonical labelings of graphs. In *Combinatorial mathematics*, pages 223–232. Springer, 1978.

[8] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-Fi: collecting high-fidelity whole-system provenance. In *28th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2012.

[9] Yu Shyang Tan, Ryan K.L. Ko, and Geoff Holmes. Security and data accountability in distributed systems: A provenance survey. In *10th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1571–1578. IEEE, 2013.

[10] John Towns, Timothy Cockerill, Mayhal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science Engineering*, 16(5):62–74, Sep. 2014.

[11] Xinlei Wang, Kai Zeng, Kannan Govindan, and Prasant Mohapatra. Chaining for securing data provenance in distributed information networks. In *31st IEEE Military Communications Conference*, pages 1–6. IEEE, 2012.