Outsourcing Graph Databases with Label-Constrain Query Verification

Kalikinkar Mandal
Department of Electrical Engineering
University of Washington
Seattle, USA 98195

Basel Alomair
King Abdulaziz City for Science and Technology
P.O Box 6086, Riyadh 11442,
Saudi Arabia

Radha Poovendran
Department of Electrical Engineering
University of Washington
Seattle, USA 98195

ABSTRACT

Graph databases have received a lot of attention due to the easy representation of complex and large data such as web, XML and social network data. In this paper, we consider a three party model where a data owner outources a graph database to a server in the cloud and allows a set of clients to make queries on the graph database. To assure the correctness of a query answer, the verification of the query answer is crucial. We present a framework for ensuring the integrity of graph databases and label-constrained graph queries. We first provide a construction of an authenticated data structure for graph databases. Our construction exploits the topology of the graph database to build the authentication tree. Then, we show how to construct the proof for the verification of answers of node and edge queries, label-constrain reachability query and pattern matching queries with data retrieval. The proof size of a query answer is proportional to the size of the answer. For the security purpose, the proposed scheme incurs $O(M)$ extra storage in the cloud where $M$ is the number of nodes in the graph database. Our scheme is secure against a computationally bounded adversary. Finally, we implement the proposed scheme in C using the PBC library to demonstrate its efficiency.

Keywords

Graph database, cloud computing, outsourcing computation, graph query integrity

1. INTRODUCTION

Graph databases have received a lot of attention because of the easy representation of complex and big data from applications such as social networks, web graphs, biological networks, semantic web and XML documents. These applications demand effectively managing graph data and querying on graph data. Graph databases support a number of queries such as reachability query [11][37][29], shortest-path query [7][11], pattern match query [42][14][15][26]. In recent years, a number of graph database projects have been initiated both from academia such as GraphLab [25], Pregel [27], Trinity [35] and from industry such as Neo4j [29], GraphDB [21], Titan [46] and OrientDB [50] which use graph models such as Resource Description Framework (RDF), property graph and labeled and multi-property graph. On the other hand, a number of graph query languages such as Gremlin, SPARQL, Cypher and Blueprints have been developed [17][35][29].

With the advent of cloud computing, the data and computation outsourcing has become very demanding both from industries and individuals. Consider an outsourcing model where there are three entities namely a graph database owner (or creator), a cloud server and a set of clients, known as three party model [19][33][16][40]. The owner $O$ outsource a graph database to a server $S$ in the cloud and allows the set of clients to make graph queries on the database. Moreover, the owner does not want to be the part of the interaction between the client and the cloud server and he remains offline. In this model, the client trusts the owner of the database, but does not trust the cloud server.

In this paper, we consider the problem of outsourcing a graph database to a cloud with the assurance of graph data and graph query integrity. We consider graph databases consisting of a finite set of nodes, a set of directed edges and each edge and each node have labels and properties or attributes, which contain data. Each node has a unique identifier. As the graph database is outsourced to the cloud, the owner wants the assurance of the integrity of the graph data. Since the clients don’t trust the cloud, the clients wants the assurance on the correctness of answers of the graph queries. We consider fundamental graph queries: node and edge queries, label-constrain reachability query and (label-) pattern matching query with retrieval of node and edge properties, which have applications in biological and social networks and XML data.

Previous works in [19][39][16] studied the authenticity of queries on graphs without node and edge labels and properties (e.g., [39] studies shortest path query). In [19], Goodrich et al. studied the verification of connectivity queries based on path hash accumulators. This approach, however, is not storage efficient for label-constrain queries as for label-constrain reachability query, it is not feasible to precompute all the paths of the label-constrain query reachability query for all pairs of nodes and all possible labeled sets. For which, the required storage is $O(M^2l^2)$ where $M$ is the number of nodes and $l$ is the number of different edge labels in the graph database. A signature based scheme can be applied for ensuring the integrity of queries on graph databases. This approach is also not efficient with respect to storage for graph databases with dynamic update operations. The main difficulty on the construction of a succinct proof is that the verification of the answer of the graph query as well as requested node and/or edge properties has to be done simultaneously.

1.1 Our Contribution

We present an efficient and secure scheme for the construction and verification of proofs of label constrain graph
queries with retrieval of node and edge properties or attributes while achieving the integrity of the graph database. Our approach for the security of graph data follows the paradigm introduced in \[6\] \[33\]. Our contributions in this paper are as follows. First, we show how to construct an authenticated data structure for a graph database using a bilinear accumulator. We propose two techniques for the construction of authenticated hash tree (AHT): the AHT construction based on node identifiers and the AHT construction by exploiting the structure of the underlying graph topology. In the later approach, we use cluster based approach to construct the authenticated hash tree.

Second, we present the construction of proofs and its verification algorithms of node and edge queries, label-constrain reachability query and (label) pattern matching query with data retrieval. The proof construction is based on membership queries. The proof verification algorithm does not reveal any information about graph data, other than the requested answer. The size of the proof of a query answer is proportional to the size of the answer of the query. Our solution requires \(O(M)\) extra storage in the cloud for the security of the graph data. The communication overhead for the proof is \(O(n)\) where \(n\) is the size of the answer. The proposed construction enjoys efficient updates for the graph data, proof constructions for advanced queries and their efficient verifications. The proposed construction is efficient with respect to computation and communication overhead and secure against computationally bounded adversaries.

Finally, we implement our scheme to evaluate the efficiency of the authenticated hash tree construction and proof generation and verification algorithms. We implement the proposed scheme in C with the Pairing-Based Cryptography (PBC) library \(1\). We constructed three graph databases using the real-world graphs taken from the Stanford SNAP dataset \[24\]. As a graph query language, we use Gremlin to execute different graph queries and to measure the size of the proofs constructed from identifier-based and cluster-based AHTs. Furthermore, the performance of the proof generation algorithm and the query verification are presented.

The rest of the paper is organized as follows. In Section 2 we present the cryptographic primitives that we use in this paper, the definition of the graph database and its operations, and queries we consider in this paper. In Section 3 we provide a security definition of the graph data and show how to encode graph data to construct authenticated hash tree. Section 4 presents the constructions of the authenticated data structure, proof generation and its verification. In Section 5 we evaluate the efficiency of the proposed scheme and in Section 6 we conclude the paper.

1.2 Related Work

A significant amount of effort has been put on efficient execution of (label-constrain) reachability query \[23\] \[12\] \[0\] \[2\] and pattern matching queries \[14\] \[15\] \[26\] \[22\] \[41\] \[3\] for graph databases. On the other hand, a few works focus on the security of graph database queries, for instance \[8\]. Since in this work we focus on the security of graph database query, we provide related works on outsourcing data and computation to clouds.

Authenticated Data Structure. An authenticated data structure allows a data owner to outsource data with data integrity guarantee. A number of works on authenticated data structures have been carried out [28] [19] [34] [18] [32] [53] where outsourcing objects are graphs without data, geometric objects [19], network data [18], XML data for pattern matching [32]. In [28], Martel et al. proposed the authenticated data structure for search DAG and showed how to construct verification objects. They also studied authenticated data structures for multi-dimensional range queries. In [19], Goodrich et al. proposed a technique for building authenticated data structure for graphs and geometric objects. Their authenticated data structure represents a graph and supports the authentication of graph queries such as connectedness queries and path queries. In [33], Papamanthou et al. presented secure protocols for authenticating membership queries on authenticated hash tables using the RSA accumulator. In [18], Ghosh et al. presented the construction of privacy-preserving authenticated data structure for network data where data are arranged in lists, trees, and partially-ordered sets with finite dimension. In [32], Papadopoulos et al. studied the authentication of pattern matching over a collection of textual data and XML documents where the textual data and XML documents are represented in form of tree. One may think that the collection of documents as a database, but they are different from graph data as no link or relationship between the XML documents or texts have been considered.

Query Verification on Relational Databases. In [40], Zheng et al. presented a construction for the query integrity of a dynamic outsourced database, which supports selections, projections, join and aggregate operations. Pang et al. presented a protocol based on signature aggregation for verifying the authenticity, completeness and freshness of query answers [31]. They proposed a technique for query verification. In [10], Cheng et al. proposed a mechanism for clients to verify the answer of a query on multidimensional datasets. They presented two schemes based on space partitioning KD-tree and data partitioning R-tree. Our work considers the query verification of graph databases where data are stored in nodes and two nodes may have a relationship that is represented by an edge, which also has data. The relationship between nodes makes our work different.

Query Verification for Graphs and Graph Databases. Recent works focus on the query verification for graphs as well as graph databases [10] [40] [8], but their work mainly focus on distance queries such as shortest path query and reachability query and pattern query for graphs without node and edge properties and labels. In [59], Yiu et al. proposed a scheme for shortest path verification in outsourced networks where the owner pre-computes all the shortest paths between pairwise nodes. However, our approach does not store any pre-computed answers for queries. The advantages of our scheme over others is it supports efficient updates on graph databases and verifications of answers of graph queries. In [8], Fan et al. studied authenticated (pattern) subgraph query for outsourced graph databases where the graph database consists of a number of small subgraphs. The proposed technique uses the Markle hash tree to authenticate subgraph queries. In [22], Groß presented a framework for certification and proofs method to sign committed graph topology consisting of a set of nodes and a set of edges where nodes and edges have labels. Their work focuses on how can a prover convince a verifier that the committed...
topology fulfills security properties. Note that all the existing works focus on graph queries where vertices and edges of graphs do not have properties and labels, except the work in \[22\]. Compared to work in \[22\], we consider update operations on graph database and support graph queries and their verifications.

**Verifiable Computation.** Another line of work on outsourcing computation focuses on verifiable computation, which is concerned about the correctness of a computation outsourced by a client and performed by a remote server. In \[4\], Benabbas et al. presented a scheme for verifiable database where a weak client outsources database and computations and makes queries to the server. In \[6\], Canetti et al. presented a scheme for verifiable evaluation of set operations such as union and intersection, which also allows to verify the execution of queries composed of union and intersection operations.

2. **PRELIMINARIES**

In this section we review the cryptographic tools that we use to construct the schemes for graph databases. We also provide a definition of the graph database and the set of operations that we consider in this paper. \(\text{neg}(\lambda)\) is a negligible function of the security parameter \(\lambda\).

2.1 **Cryptographic Tools**

**Bilinear Mapping and ECRH Hash Function.** Let \(\mathbb{G}\) be a cyclic multiplicative group of prime order \(p\) generated by generators \(g\) and \(G_{\mathbb{G}}\) be a cyclic multiplicative group of the same order \(p\). A bilinear pairing \(e: \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_{\mathbb{G}}\) satisfies the following properties: 1) Bilinearity: \(e(g^a, g^b) = e(g, g)^{ab}\) for all \(g \in \mathbb{G}\) and \(a, b \in \mathbb{Z}_p^*\); 2) Non-degeneracy: \(e(g, g) \neq 1\); 3) Computability: there is an efficient algorithm to compute \(e(g, g)\) for all \(g\). A bilinear pairing instance generator \(\text{GenBilinPairing}\) is a probabilistic polynomial time algorithm which takes as input a security parameter \(\lambda\) and generates groups and the pairing with \(\text{pub} = (p, \mathbb{G}, G_{\mathbb{G}}, e, g)\). Given an instance of a bilinear pairing, the bilinear accumulator is constructed as follows. For a set \(X = \{x_1, x_2, ..., x_n\}\) with \(n\) elements in \(\mathbb{Z}_p^*\) and a randomly chosen \(s \in \mathbb{Z}_p^*\), the accumulation value for \(X\) is

\[\text{acc}(X) = g^{(x_1 + s)(x_2 + s) \cdots (x_n + s)}\]

The witness for element \(x_i\) with respect to \(X\) is

\[W_{x_i,X} = g^{(x_1 + s) \cdots (x_{i-1} + s)(x_{i+1} + s) \cdots (x_n + s)}\]

For a secret key \(s\), the set of values \(\{g^i, 0 \leq i \leq q\}\) is the public key for the accumulator. Given the public key, the membership of the element \(x_i\) in \(X\) can be verified by checking the following equality

\[e(W_{x_i,X}, g^{s_i}) = e(\text{acc}(X), g)\]

**Definition 1 (ECRH from q-PKE).** \[6\] Choose \(q \in O(\log(\lambda))\), run \(\text{GenBilinPairing}(1^\lambda)\) to generate \(\text{pub} = (p, \mathbb{G}, G_{\mathbb{G}}, e, g)\) and sample \(a, s \leftarrow \mathbb{Z}_p^* \times \mathbb{Z}_p^*\) with \(a \neq s\), then output the public key \(pk = (\text{pub}, g^a, g^{s}, ..., g^{a_{n}})\) and trapdoor information \(sk = (a, s)\). A hash value on \(c = (c_0, c_1, ..., c_q)\) is computed as

\[H(c) = (\prod_{i=0}^{q} g^{c_i s}, \prod_{i=0}^{q} g^{a c_i s})\]

**Definition 2 (ECRH).** \[6\] A hash function \(h\) is called a collision-resistant hash function \(h: \{0, 1\}^* \rightarrow \{0, 1\}^\lambda\) if for all probabilistic polynomial time (PPT) adversary \(A\), there is a negligible function \(\text{neg}(\lambda)\) such that, for all security parameter \(\lambda\),

\[\Pr[(x, y) \leftarrow A(1^\lambda, h) : x \neq y \land h(x) \neq h(y)]\]

and it is extractable if for all PPT adversaries \(A\), there exists a poly-size extractor \(E\) such that

\[\Pr\left[ y \leftarrow A(1^\lambda, h), x' \leftarrow E(1^\lambda, h), \text{s.t. } h(x) \neq y \land h(x') \neq y \right] = \text{neg}(\lambda)\]

**Definition 3 (g-Strong Diffie-Hellman Assumption).** \[5\] Let \(\mathbb{G}\) be a cyclic group of order \(p\) where \(p\) is a prime and \(g\) is a generator. For any polynomial time adversary \(A\), the following holds

\[\Pr\left[ pub \leftarrow \text{GenBilinPairing}(1^\lambda); s \leftarrow \mathbb{Z}_p^*; \right. \left. \gamma \leftarrow A(\text{pub}, (g, g^s, ..., g^{s_q})), \text{s.t. } \gamma = e(g, g)^{s}\right] = \text{neg}(\lambda)\]

**Authenticated Hash Tree.** \[33\] Let \(\epsilon (0 < \epsilon < 1)\) be a constant. The accumulation tree for \(\epsilon\), denoted by \(T(\epsilon)\) on \(X\), can be regarded as a flat tree with properties: 1) The leaf nodes of \(T(\epsilon)\) store the elements of \(X\), which are at 0-level; 2) \(T(\epsilon)\) has \(\left\lfloor \frac{1}{\epsilon} \right\rfloor\) levels; 3) Every non-leaf node of \(T(\epsilon)\) contains \(O(n^\epsilon)\) children; and 4) Level \(j\) of \(T(\epsilon)\) contains \(O(n^{1-\epsilon j})\) nodes. The advantages of using this accumulation tree is that the search at a non-leaf node can be performed in \(O(n^\epsilon)\) time and an update can be performed in \(O(n^\epsilon)\) time.

2.2 **Graph Database & Operations**

Several definitions of graph databases can be found in the literature \[38, 3, 23, 8, 41, 14, 2, 26, 11\]. We consider a set of operations that are often applied to directed graphs where both nodes and edges are labeled and have a set of properties or attributes. Symbolically, we let \(GD = (V, E, \Sigma_1, \Sigma_2, \zeta, \rho)\) denote the graph database where \(V\) is a finite set of nodes, \(E\) is a finite set of directed edges, \(\zeta : V \rightarrow \Sigma_1\) is a node-property function which assigns properties to a node in \(V\) and \(\rho : E \rightarrow \Sigma_2\) is an edge-property function which assigns a set of properties to an edge in \(E\). We assume that each node in the graph has a unique identifier. From now on, by node \(v\), we mean a node with identifier \(v\). We denote by \((u, v)\) a directed edge from node \(u\) to node \(v\) where node \(v\) is called the head of \((u, v)\) and \(u\) is called the tail of \((u, v)\). We use \(n\text{Property}(v)\) to denote the set of node properties of \(v\) which is obtained from the function \(\zeta\) and \(e\text{Property}((u, v))\) to denote the set of edge properties of edge \((u, v)\) obtained by applying \(\rho\). We denote by \(ON(u)\) the set of all outgoing head nodes of \(u\) and \(IN(u)\) is the set of all incoming tail nodes of \(u\).

For a graph database, if nodes or edges don’t have any labels, the label is set to NULL and similarly for properties. In \[23\], the nodes in the graph database don’t have any label. Whereas, in \[26\], the edges in the graph do not have any labels and properties.

2.2.1 **Graph Update Operations**

We consider a set of operations that are often applied to update a graph database.
2.2.2 Graph Database Queries

Graph databases support a variety of queries such as reachability queries and pattern matching queries. In this paper, we consider the node query, edge query, label-constraint reachability query and pattern matching query with requested node and/or edge properties, described below.

- **Node query**: A node query is a query that retrieves information associated with a graph node including label, properties, incoming edges and outgoing edges.
- **Edge query**: An edge query is a query for retrieving information about edge label and/or edge property of an edge in a graph database. For instance, existence of an edge in the graph, retrieving edge properties, tail and head nodes of that edge. An edge query can be for retrieving incoming or outgoing edges with certain labels and edges with common properties.
- **Label-constraint reachability query (LCRQ)**: A reachability query answers whether there is a path from a given node to another given node in a directed graph. A label-constraint reachability query, denoted by $\mathcal{R}(u, v)$, takes as input two nodes $u$ and $v$ and a labeled set $A$ and answers whether there is a path from $u$ to $v$ whose path-label is a subset of $A$ and returns paths $\{(u, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{n-1}, a_n, v)\}$, between $u$ and $v$, if exist, where $a_i \in A$ and $v_i$’s are intermediate nodes of the graph database. Moreover, a client can request to retrieve properties of node and edge of the label-constraint path.
- **Pattern query**: A graph pattern matching is to find all the matches in the graph database. This query takes as input a query graph $g$ and retrieves all subgraphs in the graph database $GD$. More specifically, we consider queries that are directed and have node labels as studied in [26, 42, 41] and clients request to retrieve properties of node and edge of the answer of the pattern queries.

2.3 Authenticated Data Structure Scheme

Suppose $GD$ is a graph database that supports update and query operations. Since the graph database supports the update operations, we let $GD_t$ denote the initial graph database. At time $t$ after applying update operations, the graph database is denoted by $GD_t$. $auth(GD_t)$ is the authenticated information for the graph database $GD_t$ and $d_t$ is a succinct digest of $GD_t$ calculated based on some secret key known only to the owner. Our definition of the ADS for a graph database is similar to the original definition of ADS [34].

**Definition 4.** An authenticated data structure (ADS) scheme for a graph database consists of six polynomial time algorithms, which are defined as follows.

- $(sk, pk) \leftarrow GDB \cdot GenKey(\lambda)$: On input to security parameter $\lambda$, the key generation algorithm generates a secret key $sk$ and a public key $pk$.
- $(auth(GD_0), d_0) \leftarrow GDB \cdot Setup(GD_0, sk, pk)$: The setup phase consists of three algorithms:
  - $NodeOrder \leftarrow GraphProcessing(GD_0)$: This algorithm is run to learn about the structure of the graph and finds an ordering $GD_0$ of the graph nodes.
  - $acc \leftarrow EncNodeData(v, sk)$: This algorithm takes a node $v$ with data at nodes and edges and the secret key and outputs the node accumulation value $acc_v$.
- $(auth(GD_t), d_t) \leftarrow CalculateAuthInfo(NodeOrder, GD_t, sk)$: Given the ordering of nodes $NodeOrder$ and the secret keys $sk$ and $pk$, the setup algorithm computes an authenticated data structure $auth(GD_t)$ and its digest $d_t$ based on the ordering of nodes in $NodeOrder$.
- $(auth(GD_{t+1}), d_{t+1}, updt) \leftarrow GDB \cdot Update(u, auth(GD_t), d_t, updt, pk, sk)$: On input update $u$ to $GD_t$ and $auth(GD_t)$ and the digest $d_t$, the update algorithm returns $auth(GD_{t+1})$ and an updated digest $d_{t+1}$ and some auxiliary information $updt$.
- $(GD_{t+1}, auth(GD_{t+1}, d_{t+1})) \leftarrow GDB \cdot Refresh(u, GD_t, auth(GD_t), d_t, updt, pk)$: On input update $u$ on $GD_t$, $auth(GD_t)$, digest $d_t$, auxiliary update $updt$, and $pk$, the refresh algorithm outputs an updated authenticated data structure $auth(GD_{t+1})$ and its digest $d_{t+1}$.
• \{\text{ans}(q), \Pi(q)\} \leftarrow \text{GDB \text{-} Query}(q, GD_t, \text{auth}(GD_t), pk): \On input query \( q \) on \( GD_t \) and \( \text{auth}(GD_t) \), the query algorithm returns \( \text{ans}(q) \) to a query \( q \) and a proof \( \Pi(q) \).

• \{\text{accept}, \text{reject}\} \leftarrow \text{GDB \text{-} Verify}(q, \text{ans}(q), \Pi(q), d_t, pk): \The verification algorithm takes as input a query \( q \), answer \( \text{ans}(q) \), proof \( \Pi(q) \), a digest \( d_t \), and \( pk \) and it outputs \text{accept} or \text{reject}.

We describe the above six algorithms in Section 2 for graph databases.

Correctness and Security. Let \( \text{VerifyCond}(GD_t, q, \text{ans}(q)) \) be an algorithm which checks whether \( \text{ans}(q) \) is the answer of query \( q \) on the graph data \( GD_t \). The ADS is correct if for all \( (GD_t, \text{auth}(GD_t), d_t) \) and for all queries \( q \) and its answers \( \text{ans}(q) \) whether \( \text{VerifyCond}(D_t, q, \text{ans}(q)) \) accepts so does \( \text{Verify}(q, \text{ans}(q), \Pi(q), d_t, PK) \). The ADS is secure if for all large \( \lambda \) and for all polynomial time adversary \( A \) the following holds for \( 0 \leq j \leq t \):

\[
\Pr \left[ (q, \text{ans}(q), \Pi(q), j) \to \text{As.t.} \right.
\left. \text{accept} \leftarrow \text{Verify}(q, \text{ans}(q), \Pi(q), d_t, PK) \right. \land
\left. \text{reject} \leftarrow \text{VerifyCond}(D_t, q, \text{ans}(q)) \right] \leq \text{neg}(\lambda).
\]

3. GRAPH DATA ENCODING AND NODE ACCUMULATION

In this section, we provide a security definition of graph data. We then describe the graph processing and node data encoding algorithms of the set up phase.

3.1 Security Definition of the Graph Data

Our security notions of graph data are as follows: When the graph data is outsourced to an untrusted cloud server, a computationally bounded adversary should not be able to alter graph data such as removing nodes, edges and their labels/properties from the graph, changing the direction of the edges, changing the number of incoming and outgoing edges at a node and altering any data at nodes or edges. Moreover, if the adversary inserts additional fake information in the graph, she will be caught by the data owner. Our security notion on the graph database keeps the topology of the graph database and the data unaltered. We achieve these security goals by providing an encoding scheme in Section 3.3 and 3.4. For query security, a computationally bounded adversary should not be able to convince an honest client by constructing a fake proof of an answer of a query. We provide a formal definition of the query security below.

**Definition 5 (Query Security).** Let \( \lambda \) be a security parameter and \( A \) be a computationally bounded adversary. On input \( pk \) and a graph query \( Q \), the adversary \( A \) generates an answer \( \text{ans}(Q)^* \) and a proof \( \Pi^* \) such that

\[
\Pr \left[ \text{Verify}(\text{ans}(Q)^*, \Pi^*) = 1 \land \text{ans}(Q)^* \text{ is not the correct answer of } Q \right] = \text{neg}(\lambda)
\]

where \( \text{Verify}(\cdot) \) is the verification algorithm run by the client.

3.2 Graph Topology Processing

We exploit the topology of the underlying graph of the graph database to build the authentication tree for the graph data for achieving efficiency in the proof generation and verification algorithms. We use a clustering algorithm as a topology processing tool to find an ordering of nodes of the graph database. The clustering algorithm finds the groupings of the nodes of the graph into clusters in such a way that there are many edges in each cluster and few edges between clusters. Since we consider reachability and pattern queries, the cluster based node ordering helps to reduce the size of the proof of query answers.

We find the ordering of the nodes of the graph database as follows. Let \( \text{Comp}_1, \text{Comp}_2, ..., \text{Comp}_t \) with \( \text{Comp}_1 \cap \text{Comp}_2 = \emptyset, i \neq j \) be the clusters of the graph after applying a clustering algorithm. Graph clustering algorithms can be found, for instance, in [2]. Using this partition, we construct a new graph as follows: 1) Introduce a vertex for each cluster and draw an edge between every pair of vertices. This is a complete graph with \( t \) vertices and \( \binom{t}{2} \) edges. 2) Assign the number of directed edges between clusters \( \text{Comp}_i \) and \( \text{Comp}_j \), as a weight on edge \((i,j)\) between vertices \( i \) and \( j \), \( 1 \leq i, j \leq t \). If there is no edge between a pair of clusters, assign the weight of the corresponding edge as zero. We denote this graph by \( G_0(t) \). In \( G_0(t) \), we consider all the paths \( P \) of length \( t - 1 \) in which a path \( P \) contain each vertex exactly once. The weight of a path \( P \), denoted by \( w(P) \), is the sum of the weights of all the edges on the path \( P \). We then find a path \( P^* \) of length \( t - 1 \) for which \( w(P^*) \) is the maximum. The ordering of the vertices in the path \( P^* \) decides the ordering of the clusters.

We organize leaf nodes of the accumulation tree in such a way that, in a cluster, the distance between highly connected nodes of the graph database is small in the accumulation tree. In each \( \text{Comp}_i \), the ordering of the nodes are determined based on their identifier. The ordering of the clusters and the ordering of the nodes in each cluster determine the ordering of the nodes of the graph database and we denote it by \( \text{NodeOrder} \). The ordering of the node identifiers in \( \text{NodeOrder} \) is a permutation on the node identifiers. Note that this is a one-time processing of the graph data. If any data update comes at nodes or edges, the ordering of the nodes remains unchanged. On the other hand, if any node or edge needs to be inserted or deleted in the graph, there is no need to run the clustering algorithm again, only the position of the nodes needs to be decided using the predetermined clustering.

3.3 Encoding graph data

Assume that there are \( M \) nodes in the graph database described in Section 2.2 and each node has a unique identifier (ID). Let \( \{1, 2, ..., M\} \) be the node identifiers of the graph database. We provide an encoding rule to encode the graph data that allows to efficiently update node and edge labels and properties and node adjacencies. Let \( h : \{0, 1\}^* \to \{0, 1\}^\lambda \) be a collision-resistant hash function. We use the function \( \mu(\cdot) \) to convert an image of the hash function to an element of \( Z_p^\lambda \) where \( p \) is a \( \lambda \)-bit prime.

- **Node property:** Let \( m_j \) be the number of different properties of a node with identifier \( v_j, j \in [M] \). We combine node properties with its node ID and node label to convert to an element of \( Z_p^\lambda \). For each node property \( l_{v_j,i}, 1 \leq i \leq m_j \) of node \( v_j \), we compute \( nl_{v_j,i} = \mu(h(v_j||nlbl_{v_j}||l_{v_j,i})) \) for \( j \in [M] \) and \( i \in [m_j] \) where \( nlbl_{v_j} \) is the label for the node with ID \( v_j \). We denote the node property set by \( NL_{v_j} = \{nl_{v_j,i} : nl_{v_j,i} \in Z_p^\lambda, i \in [m_j]\} \).
- **Adjacent nodes**: Each node in the graph has two
types of edges, namely incoming edges and outgoing
edges. For each outgoing edge \((v_j, u)\) from node \(v_j\)
to node \(u\) with label \(elbl(v_j, u)\), we compute a digest
as \(h(v_j || u)\) and then convert it to an element of \(Z_p^*\)
by applying the function \(\mu = an(v_j, u) = \mu(h(v_j || u))\).
For all outgoing edges of \(v_j\), we construct the outgoing
node set, denoted by \(ON_{v_j}\), as \(ON_{v_j} = \{an(v_j, u) = \mu(h(v_j || u)) : u \in ON(v_j)\}\) for
\(j \in [M]\). Similarly, for an incoming edge from node \(u\) to
node \(v_j\), the edge digest is given by \(h(u||v_j)\) and the incoming
node set is constructed as \(IN_{v_j} = \{in(u,v_j) = \mu(h(u||v_j)) : u \in
IN(v_j)\}\) where \(IN(v_j)\) contains all tail endpoints to
node \(v_j\) for \(j \in [M]\).

- **Edge properties**: Let \(n_{(v_j, u)}\) be the number of
properties of an edge \((v_j, u)\) from node \(v_j\) to \(u\) and \(l_{bl}\),
be the \(ith\) property of the edge \((v_j, u)\). Each edge
property is converted to an element of \(Z_p^*\) as \(el_i = \mu(h(v_j || u, elbl(v_j, bl)))\), \(i \in [n_{(v_j, u)}]\). We denote the edge
property set by \(EL_{(v_j, u)} = \{el_{i}, el_i \in Z_p, i \in [n_{(v_j, u)}]\}\).

### 3.4 Node accumulation

Recall that, in the graph database, each node is coupled
with four different types of information, namely a set of node
properties, a set of incoming and outgoing edges and a set of
edge properties between edges. The collision resistant
property set by \(EL_{(v_j, u)} = \{el_{i}, el_i \in Z_p, i \in [n_{(v_j, u)}]\}\).

The edge property accumulation value is given by

\[
acc(\mathcal{EL}_{(v_j, u)}) = g^{(sk + \mu(h(n_{(v_j, u)})\cdot
\mu(h(v_j || l_{elbl(v_j, bl)})))\cdot \prod_{i=1}^{n_{(v_j, u)}} (sk + nl_{el})}.
\]

The set of all outgoing edge property accumulation values
are given by \(ELP_{v_j} = \{\mu(acc(\mathcal{EL}_{(v_j, u)})) : w \in
ON(v_j)\}\).

For incoming edges to node \(v_j\) and outgoing edges from
node \(v_j\), the incoming node accumulation value and outgoing
node accumulation value are computed as

\[
acc(\mathcal{NL}_{v_j}) = g^{(sk + \mu(h(IN(v_j)))\cdot \prod_{w \in
IN(v_j)} (sk + nl_{in})}}.\]
\[
acc(\mathcal{ON}_{v_j}) = g^{(sk + \mu(h(ON(v_j)))\cdot \prod_{w \in
ON(v_j)} (sk + nl_{in})}}.
\]

A collision resistant node property accumulator prevents
from altering node properties and the number of node properties,
an edge property accumulator prevents from altering
edge properties, the number of edge properties and swapping
edge properties between edges. The collision resistant
incoming and outgoing edge accumulators ensure the adjacency
of nodes of the graph and their respective numbers.

The security definitions of the above accumulators are
provided in Appendix.

**Lemma 1.** Assume there exists a collision-resistant hash
function \(h\). According to the above construction of the node
accumulator, the probability that one can manipulate 1) node
property, 2) edge property, 3) the node adjacency lists and 4)
indegree and outdegree is \(\text{neg}(\lambda)\).

**Proof (Sketch).** The existence of a collision resistant
hash function ensures distinct values in sets \(\mathcal{NL}_{v_j}, \mathcal{EL}_{(v_j, u)},
\mathcal{IN}_{v_j}, \mathcal{ON}_{v_j}\). According to the q-SDH assumption,
for a computationally bounded adversary, the probability to find
another set \(\mathcal{NL}'\) corresponding to another set of node
properties \(\{v_{ij}, i \leq 1 \leq m\}\) such that \(acc(\mathcal{NL}) = acc(\mathcal{NL}')\)
is \(\text{neg}(\lambda)\). Similarly, for edge properties and incoming
and outgoing edges, the probability of manipulating incoming,
outgoing edges and their labels/properties is \(\text{neg}(\lambda)\).

Using the above accumulation values, we compute a node
accumulation value of \(NA_{v_j}\) where

\[
NA_{v_j} = \{\mu(acc(\mathcal{NL}_{v_j})), \mu(acc(\mathcal{ON}_{v_j})), \mu(acc(\mathcal{IN}_{v_j})),
\mu(acc(\mathcal{EL}_{(v_j, u)})) : w \in ON(v_j)\}.
\]

One can compute the node accumulation value using an
accumulation tree \(T(\epsilon)\) 0 < \(\epsilon\) < 1 on the values in \(NA_{v_j}\). If
the total number of elements in \(NA_{v_j}\) is \(k\), the depth of the
accumulation tree is \(\lceil \log k \rceil\) and each non-leaf node of
\(T(\epsilon)\) has \(k^{1-\epsilon}\) children. However, with this approach, the depth
of the accumulation tree will increase for the fixed number of
children of non-leaf nodes of \(T(\epsilon)\). As a result, the com-
plexity of the update operation and proof generation will
increase. We compute the node accumulation value in such
a way that the update operation, proof generation and its
verification are efficient. We compute the node accumulation
value as follows.

\[
acc(\mathcal{ELS}_{v_j}) = g^{\prod_{w \in ON(v_j)} (sk + \mu(acc(\mathcal{EL}_{(v_j, u)})))}
\]
\[
X_1(sk) = (sk + \mu(acc(\mathcal{NL}_{v_j}))) (sk + \mu(acc(\mathcal{ELP}_{v_j})))
\]
\[
X_2(sk) = (sk + \mu(acc(\mathcal{ON}_{v_j}))) (sk + \mu(acc(\mathcal{IN}_{v_j})))
\]
\[
acc(\mathcal{NA}_{v_j}) = g^{X_1(sk) \cdot X_2(sk)}.
\]

There are several reasons for the above choice of encoding
and computing the node accumulation value. One may compute
a single digest for the edge label and properties using a
hash function, but for graph data, which does not allow to
efficiently update edge properties. One can also compute
a digest for the node properties using a hash function by
putting all node properties and the node label together, but
similarly, it does not allow to have efficient update. The
same argument is also applied to the outgoing and incom-
ing edge accumulations. Moreover, in the query verication
phase, clients need to know all edge properties and node
properties to perform the verication, which may not be des-
irable in certain applications for the privacy reasons. If the
server does not store the accumulation values for the
edge labels, the complexity for computing a witness of an
edge labels remains unchanged.

We can observe that one needs to update two accumu-
lation values to update a node property, incoming edge or
outgoing edge and needs to update three accumulation val-
ues to update an edge property. The node accumulation
value to be updated if any update is applied on node la-
bel properties, edge labels or properties and incoming
and outgoing edges. We have the following definition of the
collision resistant node accumulator.

**Definition 6 (Collision Resistant Node Acc.).** Given
a set \(NA_{v_j}\) for node labels, incoming, outgoing edges and
outgoing edge labels for node \( v_j \) and the bilinear parameter \( pub = (p, \mathbb{G}_T, e, g) \) chosen according to security parameter \( \lambda \) and the elements of \( \mathbb{G}_T \) \( \{g, g^y, g^{y^2}, \ldots, g^{y^k}\} \) for some randomly chosen \( s \in \mathbb{Z}_p^* \), the probability that a computationally bounded adversary can find \( \mathcal{NA}' \) such that \( \text{acc}(\mathcal{NA}) = \text{acc}(\mathcal{NA}') \) is \( \neg\text{neg}(\lambda) \). The adversary is allowed to find the set \( \mathcal{NA}' \) by changing node labels, edge labels and incoming and outgoing edges.

**Lemma 2.** Assuming the existence of collision-resistant hash function \( h \). The algorithm for the node accumulation value is secure under the \( q \)-Strong DDH assumption.

**Proof (Sketch).** According to our construction, an adversary can forge a node accumulation value if and only if she solves the DDH problem for the node property accumulation value, incoming and outgoing edge accumulation values or the edge accumulation values. To forge a node property accumulation value, the adversary needs either 1) to find two node properties \( l_{v_{j,i}} \) and \( l_{v_{j,i}}' \) such that \( h(v_j) || nbl_{v_{j,i}} || en(l_{v_{j,i}}) = h(v_j) || nbl_{v_{j,i}} || en(l_{v_{j,i}}') \) or 2) to solve the DDH problem. Similar arguments can be applied to the incoming and outgoing accumulation values. Again, in order to forge an edge accumulation value, the adversary needs to solve the DDH problem for one of the edge property accumulation values. The edge property forgery is equivalent to forging the hash function or solving the DDH assumption. According to Lemma 1, a computationally bounded adversary can manipulate a node property, an edge property and adjacency list of the node with negligible probability. Therefore, the construction of the node accumulation value is secure under the DDH assumption and existence of collision resistance hash functions. \( \square \)

**Witness Calculation.** We now describe how to compute witnesses for the existence of a node property, an edge label/property and the adjacency of that node. The witness computation to be performed by the server to prove the existence of a node related information such as node label and node properties. To compute a witness for a set of node properties \( l_{v_{j,i}} \), \( 1 \leq k \leq t \) for a graph node with ID \( v_j \), the server computes

\[
lp(x) = \prod_{i=\mathbb{N}, v_{j,i}} (x + \mu l_{v_{j,i}}) = \sum_{i=0}^{m_j-t} c_i x^i, c_i \in \mathbb{Z}_p^*,
\]

where \( nlp \) is \( \mu \) and \( l_{v_{j,i}} \) is \( h(v_j) || nbl_{v_{j,i}} || en(l_{v_{j,i}}) \). The server then applies the ECRH function on \( lp(x) \) and computes the witness \( W_{nlp} = g^{(sk + \mu (h(m_j))) lp(x)} \) using the public key \( pk \).

Therefore, the time complexity to compute the witness for \( t \) node properties is \( O((m_j - t) \log(m_j - t)) \) as the above polynomial expansion takes \( O((m_j - t) \log(m_j - t)) \) time. The verification for the node properties \( l_{v_{j,i}} \), \( t \in \{i_1, \ldots, i_t\} \) is done by first computing \( \omega(sk) = g^\sum_{t \in \{i_1, \ldots, i_t\}} l_{v_{j,i}} \) using the ECRH function and then checking the following equality

\[
e(W_{nlp}, \omega(sk)) = e(\text{acc}(\mathcal{NL}_{\mathcal{V}_j}), g).
\]

Similarly, the server can compute a witness for an edge property/label using the ECRH function without knowing the secret key with time complexity \( O((nlp_{(v_{j,u})}) \log(nlp_{(v_{j,u})}) - t) \) where \( t \) is the number of edge properties for which the witness to be calculated. Likewise, the server can compute a witness for an outgoing edge and an incoming edge of \( v_j \) in time \( O(ON(v_j)) \) \log(ON(v_j)) \) and \( O(IN(v_j)) \log(IN(v_j)) \), respectively, where \( ON(v_j) \) is the set of head nodes of all the outgoing edges of \( v_j \) and \( IN(v_j) \) is the set of all tail nodes of incoming edges of \( v_j \).

When the node accumulation value is known, the witness for a node accumulation is computed as

\[
W_{l_{v_{j,i}}} = g^{\mu lp(sk)}
\]

\[
wt(x) = (x + \mu \text{acc}(\mathcal{OL}_{v_{j,i}})) \cdot (x + \mu \text{acc}(\mathcal{IN}_{v_{j,i}}))
\]

\[
W_{\text{acc}(\mathcal{NL}_{v_{j,i}})} = g^{\mu wt(x)}
\]

The witness for the set node properties \( \{l_{v_{j,i}}, 1 \leq k \leq t\} \) is given by

\[
\{(W_{l_{v_{j,i}}}, \text{acc}(\mathcal{NL}_{v_{j,i}})), (W_{acc(\mathcal{NL}_{v_{j,i}})}, \text{acc}(\mathcal{NA}_{v_{j,i}}))\}
\]

The witness for an edge label or a set of edge properties of an edge \( e(v_{j,u}) \) can be found in the similar way. The witness is given by

\[
\{(W_{P}, \text{acc}(\mathcal{EL}_{e(v_{j,u})})), (W_{\text{acc}(\mathcal{EL}_{e(v_{j,u})})}, \text{acc}(\mathcal{EL}_{P})),
\}

\[
(W_{\text{acc}(\mathcal{EL}_{P})}, \text{acc}(\mathcal{NA}_{e(v_{j,u})}))\}
\]

where \( P \) is an edge label or a set of edge properties and \( W_P \) denotes the witness for \( P \).

Given a node accumulation value, the time complexity to compute a witness for a set of node properties is \( O(|ON(v_j)|) \cdot n + |IN(v_j)| + (m_j - t) \) where \( |O| \) is poly-log overhead and \( n \) is the maximum number of edge properties for an outgoing edge and \( m_j \) is the number of node properties. For a set of edge properties, the time complexity to compute the witness by the server is \( O(|ON(v_j)| \cdot n + |IN(v_j)| + m_j + |ON(v_j)|) \). When the node accumulation value, edge accumulation value and incoming and outgoing edge accumulation values are pre-computed, the time complexities for computing witnesses for a set of node properties and a set of edge properties are \( O(m_j) \) and \( O(|ON(v_j)| n) \).

**Lemma 3.** Let \( \lambda \) be a security parameter. Given a collision-resistance hash function \( h \), a computationally bounded adversary can alter, an incoming (or outgoing) edge with the number of incoming (outgoing) edges, and a node label/property with the number of properties, with probability \( \text{neg}(\lambda) \). Moreover, the adversary can alter an edge label/property with probability \( \text{neg}(\lambda) \).

**Proof.** The proof is similar to that of Lemma 2. \( \square \)

4. AUTHENTICATED DATA STRUCTURE FOR GRAPH DATABASE

In this section we present the construction of an authenticated data structure for graph data and describe the proof construction and its verification algorithms for label-constrain graph queries with data retrieval.

4.1 Setup and Updates on Graph Database

\{sk, pk\} ← GDB · GenKey(1^\lambda): For a security parameter \( \lambda \), the source/owner runs the key generation algorithm to generate a secret key \( sk \) and a public key \( pk = (g, g^{sk}, \ldots, g^{sk^k}) \).
\{auth(GD_0), d_0\} \leftarrow \text{GDB} \cdot \text{Setup}(GD_0, sk, pk)$. The initial setup phase consists of processing the topology of the underlying graph of the graph database, calculating of accumulation values for all nodes and the construction of the accumulation tree. Let $GD_0$ be the initial graph database and $M$ be the number of nodes in the graph. The owner will build an accumulation tree on the graph nodes in which nodes of the graph are leaf nodes of the accumulation tree. While building the accumulation tree, the owner can sort the nodes of the graph databases by the lexicographical order of their node identifiers and then place the sorted nodes as the leaf nodes in the accumulation tree. We call this accumulation tree building approach is Identifier-based Acc. Tree construction.

Another approach to build an accumulation tree is based on graph clustering and we call it Cluster-based Acc. Tree. To construct an accumulation tree based on this approach, the owner applies the technique described in Section 3.2.2 to find the ordering of the nodes of the graph database. We let $\text{NodeOrder} = \{v_1, v_2, \ldots, v_M\}$ denote the ordering of the node identifiers, which is a permutation on the node identifiers $\{1, 2, \ldots, M\}$. A mapping between the node ordering in NodeOrder and the graph node identifier is implemented by the server to accelerate the proof generations of the answers of the queries and the update operation by the data owner. The mapping is also sent to the server. For each node with identifier $v_{i,j} \in [M]$, the owner first computes the accumulation values for all the nodes in the graph described in Section 3.4. We let $R = \{R_0, R_1, \ldots, R_{\text{level}}\}$ be the set of node accumulation values for the nodes with identifiers in NodeOrder. We construct an accumulation tree, denoted by $GAT(e)$, $0 < \epsilon < 1$, over $R$ where each node in $GAT(e)$ has $M^\epsilon$ children and the depth of $GAT(e)$ equals $\lceil \frac{1}{\epsilon} \rceil$. Figure 1 shows the construction of such an accumulation tree.

We denote the level of the root by the 0th level and the level of the leaf nodes by $l = \lceil \frac{1}{\epsilon} \rceil$th level. Each node of $GAT(e)$ contains a value, denoted by $\zeta(u)$ which is computed as follows. At the level values at the leaf nodes of $GAT(e)$ are $\{R_0, R_1, \ldots, R_{\text{level}}\}$. At the level $l$ of $GAT(e)$, $0 \leq i \leq l - 1$, for a node $u$, $\zeta(u)$ is computed as $\zeta(u) = \mathbb{P}_{v \in N(u)}(\mu(h(v) || nbl_{v_i} || nprop_{new}))$ where $N(u)$ is the set of children of $v$, as computed in 6. Finally, the owner sends the authentication information $\text{auth}(GD_0) = \{R_0, R_1, \ldots, R_{\text{level}}\}$ to $GAT(e)$, $\zeta(\text{root})$ where root is the root node of $GAT(e)$. We denote by $\text{Path}(R_i)$ the path from the node $v_i$ with value $R_i$ to the root node in $GAT(e)$. Note that the accumulation tree $GAT(e)$ is built on top of the graph data.

\{auth(GD_{t+1}), d_{t+1}, upd\} \leftarrow \text{GDB} \cdot \text{Update}(u, \text{auth}(GD_t), d_t, upd, sk, pk)$. The graph data update operation can be classified as: (i) insertion and deletion of nodes and edges and their properties. See Section 2.2.2 for details. When the owner of $GD_0$ updates any data at nodes or edges, he also needs to update the graph database $GAT(e)$. The owner performs the update operations as follows.

To insert a new node property $nprop_{new}$ at a node $v_j$, the owner first computes $\mu(h(v_j) || nbl_{v_j} || nprop_{new})$ and then performs $\text{acc}(\mathcal{N}_{v_j}^{new}) = \text{acc}(\mathcal{N}_{v_j}^{\epsilon})(\mu(n_{v_j} || nprop_{new}))$. For the inserted node property, the server replaces the values in $\text{auth}(GD_t)$ with the corresponding values $\text{upd}.$

4.2 Graph Queries and Verifications

In this section, we describe the techniques for constructing proofs (or verification objects) for the graph queries described in Section 2.2.2 and their verification techniques. **Main idea.** Our proof construction is based on the observation that the answer of a graph data query can be expressed
as a sequence of nodes and/or edges with labels/properties.

The main idea behind the construction of a proof for an answer is to split the answer of the query into disjoint sub-answers sequentially by maintaining the order and then construct proofs for all disjoint sub-answers. In the proof construction algorithm, a proof for a sub-answer is constructed through the witness computation and using the authenticated hash tree $GAT(\varepsilon)$. With this approach, the proofs for label-constrain queries with any node and edge properties can be constructed. We start by describing the proof construction algorithm for node and edge queries.

### 4.2.1 Node Query & Edge Query

In a graph database, nodes contain a set of properties and edges representing relationship between nodes have labels and also contain data. A node query is a query that retrieves information stored at nodes such as node properties, the node label, adjacent nodes. For a social network database, an example of a node query can be used to retrieve information associated with an edge such as edge label and data stored on edges, and the tail node and head nodes of an incoming or outgoing edge.

#### Node query

Assume that a client makes a node query $nQry(v_j)$ to find a subset of node properties of specific interest at node $v_j$. Let $NL_{v_j} = \{l_{v_j,1}, ..., l_{v_j,m_{v_j}}\}$ be the set of all node properties and $ans(nQry(v_j)) = \{l_{v_j,1}, l_{v_j,2}, ..., l_{v_j,i_k}\}$, $1 \leq k \leq t$, be the answer of the node query $nQry(v_j)$. We provide the proof generation and verification algorithm of $ans(nQry(v_j))$ in Algorithm 1.

Assume that a client makes a query requesting to find all nodes in the graph database with certain property. For instance, for a research paper citation database, find all nodes with node label "paper" for which the number of citations is greater than equal to the number $t$. In the citation graph, nodes may have labels "paper", "book", "article", etc. Let $ans(nQry(cn \geq t)) = \{(v_{i_1}, cn_{i_1}), (v_{i_2}, cn_{i_2}), ..., (v_{i_k}, cn_{i_k})\}$ be the answer of the query $nQry(cn \geq t)$ where $cn$ denotes the number of citations. One approach to construct a proof for $nQry(cn \geq t)$ is to construct proofs for each $(v_{i_j}, cn_{i_j}), 1 \leq i \leq k$ by computing the witness for $(v_{i_j}, cn_{i_j})$ w.r.t. the node accumulation value of $v_{i_j}$ and witnesses for nodes $u \in Path(v_{i_j}, root)$, as done in Algorithm 1. However, by exploiting the ordering of the nodes in the answer, the proof size can be reduced for witnesses for nodes in $Path(v_{i_j}, root)$.

We describe how to reduce the proof size with the above example.

#### Optimization of the proof size

Based on the answer, the server computes the positions of the nodes in the accumulation tree. We sort the positions of the nodes $v_n$ in the answer $ans(nQry(cn \geq t))$ and assume that $\{n_1, ..., n_k\}$ are the sorted positions of the nodes and $\{v_{n_1}, v_{n_2}, ..., v_{n_k}\}$ are the ordering of the nodes. We do $[M^+]$-base representation of $n_i$'s of length $\lceil \frac{d}{2} \rceil$. Let $n_i = a_{i1}a_{i2}...a_{i\lceil \frac{d}{2} \rceil - 1}$ and $n_{i+1} = b_{i1}b_{i2}...b_{i\lceil \frac{d}{2} \rceil - 1}$ be the $[M^+]$-base representation of $n_i$ and $n_{i+1}$, respectively where $0 \leq a_i, b_i \leq [M^+] - 1$. If for some $j < \lceil \frac{d}{2} \rceil$, $a_i = b_i$, $j \leq i \leq \lceil \frac{d}{2} \rceil - 1$, the size of the proofs for $(v_{i}, cn_{i})$, and $(v_{i+1}, cn_{i+1})$ can be optimized by $(\lceil \frac{d}{2} \rceil - j)$ as $Path(v_{n_i}, root)$ and $Path(v_{n_{i+1}}, root)$ have a common path from label $j$ in the accumulation tree. Our choice of encoding the graph data at nodes allows to place the nodes at any positions in the accumulation tree without altering the topology of the underlying graph of the graph database.

#### Edge query

When the server needs to convince the client that an edge, an edge label or a set of edge properties is associated with the given edge, the server computes a witness of it as a proof. It can be seen from the construction of node accumulation values, to construct a proof for an edge label or a subset of edge properties, the server needs to prove that the edge label or the subset of edge properties are corresponding to that edge and then ensure the edge is a true edge in that graph. The proof construction for a subset of edge properties or an edge label is done through witness computation. Moreover, the proof for the existence of an outgoing or incoming edge can be computed similarly.

**Algorithm 1** Proof construction for node query

```
1: procedure PROOF_GENERATION(ans(nQry(v_j))) =
2: Assume \{l_{v_j,1}, l_{v_j,2}, ..., l_{v_j,ik}\}, nlbl_{v_j} is the answer of \nQry(v_j)
3: Compute the witness for \nQry(v_j) as \W_{ans(nQry(v_j))} = \prod_{i \in NL_{v_j} \backslash \{ans(nQry(v_j))\}} \gamma(h(v_j, nlbl_{v_j} \{i\}) \mu, \gamma \epsilon EC\ \mu \cdot \text{pk})
4: Compute the witness for \acc(NL_{v_j}) w.r.t. node accumulation as \W_{\acc(NL_{v_j})} = g^{sk}(\mu(\acc(NL_{v_j})) \epsilon EC\ \mu \cdot \text{pk})
5: Let \Path(v_j, r) be the set of all nodes in the path from node v_j to the node root r in \GAT(\varepsilon).
6: Compute the witness for every node u in \Path(v_j, r) and assume \Pi_{Path} = \{(W_u, \zeta(u) : u \in \Path(v_j, r))\} is the witness and the label at node u.
7: \textbf{return} Proof for the answer of nQry(v_j) is \Pi = \{(W_{ans(nQry(v_j))}, \acc(NL_{v_j})), (W_{\acc(NL_{v_j})}, \acc(NA_{v_j}))\} \cup \Pi_{Path}
8: end procedure
```

#### 4.2.2 Label-constraint reachability query

In this subsection, we show how to construct the proof for a label-constrain reachability query with node and/or edge properties retrieval and provide its time complexity. Let
Verification an LCRQ query with node and edge properties is

Algorithm 2 Proof verification for node query

1: \textbf{procedure} \textsc{Proof\_Verification}(q, \text{ans}(\text{q\_Qry}(v_i)), \Pi)
2: \text{Client first computes } p(s_k) = \prod_{i=0}^{n} (s_k + \mu(h(v_j)[\text{nlbl}_v](v_{i-1}, v_i))) \text{ using ECRH function and } pk.
3: \text{Client then checks the quality } e(W_{\text{ans}(q_i)(v_i)}, g^{p(s_k)}) = e(\text{acc}(\text{NL}_v), g).
4: \text{If the above quality passes, client checks the qualities sequentially } e(W_{\text{acc}(\text{NL}_v)}, g^{s_k}, g^{\mu(\text{acc}(\text{NL}_v))}) = e(R_{\text{v}_i}, g)
\text{and } e(W_{\text{ns}_v}, g^{s_k}g^{\mu(\text{acc}(\text{nlbl}_v))}) = e(\xi(u_{i+1}), g), i = 1, 2, \ldots, [\frac{n}{2}]
5: \text{Return } \Pi \text{ of all the verification pass, the answer is correct.}
6: \textbf{end procedure}

\textit{ans}((v_0 \xrightarrow{A} v_n)) = \{(v_0, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{n-1}, a_n, v_n)\} \text{ be the answer of an LCRQ query } v_0 \xrightarrow{A} v_n \text{ where } A \text{ is the labeled set. The security notion for an LCRQ query is as follows. When a client makes an LCRQ query, he knows the starting node } v_0 \text{, end node } v_n \text{ and the edge labeled set } A, \text{ but, she does not have knowledge about the intermediate nodes and edges in the answer of } v_0 \xrightarrow{A} v_n: \text{ whether 1) intermediate nodes } v_i \text{'s do exist; 2) edges } (v_i, v_{i+1}), i = 0, \ldots, n - 1 \text{ exist; 3) } a_i \text{ is the correct edge label of } (v_i, v_{i+1}) \text{ and } a_i \in A; 4) \text{ the nodes and edges in } v_0 \xrightarrow{A} v_n \text{ appear in the proper order; and 5) correct node and edge properties.}

Our choice of the node accumulation value construction allows us to ensure the above properties in the proof construction of an LCRQ query with node and edge property request. Moreover, we use the fact that if } p = p_1(p_2) \text{ is a path, then } head(p_1) = tail(p_2). \text{ We present the proof construction for an LCRQ with node and edge properties in Algorithm 3 and its verification in Algorithm 4. If one wants to construct a proof of an LCRQ query without node and edge property request, the proof can be constructed using the witness of outgoing edges (as shown in Section 3.4) in steps 4-7 of Algorithm 3 and the verification is based on the witness verification of the outgoing edge accumulation. We omit the details of the proof generation of an LCRQ query without node and edge properties. When the LCRQ query answers multiple paths, the server generates the proof for each path. However, several optimizations on the proof size can be done: If the same node appears on different paths, we put the witness of that node only once in the proof. After applying optimizations, the proof size is } O(n).

Complexity of an LCRQ query. According to Algorithm 4, the number of bilinear operations required by the verification algorithm is } (4n + 2) + C \text{ where } C \leq (n + 1) \cdot [\frac{1}{2}] \text{ and } n \text{ is the length of the path. The cluster based authenticated hash tree construction allows us to reduce the size of the proof as well as the time for verification. In worst case, the verification algorithm needs to perform } (4n + 2) + (n + 1) \cdot [\frac{1}{2}] \text{ bilinear operations to verify the proof. For an LCRQ query without node and edge properties, the number of bilinear operations required by the verification algorithm is } 3n + n \cdot [\frac{1}{2}] \text{ in worst case. The time complexity for computing the proof an LCRQ query with node and edge properties is } O(n(M^* + T)) \text{ where } T \text{ is maximum among the number of node properties, edge properties and the number outgoing and incoming edges of the nodes in } \text{ans}(v_0 \xrightarrow{A} v_n).

Algorithm 3 Proof generation of an LCRQ with node and edge properties

1: \textbf{procedure} \textsc{Proof\_Generation}(v_0 \xrightarrow{A} v_n)
2: \text{Let } P = \{(v_0, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{n-1}, a_n, v_n)\} \text{ be a label-constrained path and } D = (D_0, D_1, \ldots, D_{n-1}, D_n) \text{ be the data requested from nodes } v_i, 0 \leq i \leq n \text{ where } D_i = (rNL_i, rEL_i) \text{ is requested node properties and edge properties at node } v_i.
3: \text{for } i = 0 \text{ to } n - 1 \text{ do}
4: \text{Compute the witness for node property } rNL_i \text{ as } W_{rNL_i} = g^{(s_k + \mu(h(v_i)))} \prod_{p \in P \cap NL_i} g^{(s_k + \mu(h(v_i)))} \text{ using the ECRH function and } pk.
5: \text{Compute the witness for edge label } a_{i+1} \text{ and edge properties } rEL_i \text{ of } (v_i, a_{i+1}, v_{i+1}) \text{ as } W_{rEL_i} = g^{\sum_{p \in P \cap rEL_i} g^{(s_k + \mu(h(v_{i+1})))}} \text{ using the ECRH function and } pk.
6: \text{Compute the witness for } acc(\mathcal{E}_L(v_i, v_{i+1})) \text{ as } W_{acc(\mathcal{E}_L(v_i, v_{i+1}))} = g^{\sum_{p \in P \cap acc(\mathcal{L}_v)} g^{(s_k + \mu(h(v_i)))}}
7: \text{Let } Path(v_i, root) \text{ be the set of all nodes in the path from node } v_i \text{ to the root node } r \text{ in } GAT(e).
8: \text{Compute the witness for every node } v_i \text{ in } Path(v_i, root) \text{ and assume } \Pi_{GAT,i} = \left\{ (w, \xi(w)) \mid w \in Path(v_i, r) \right\} \text{ is the witness and the label at node } w.
9: \text{return } \Pi_{vn} = \left\{ (W_{rNL_i}, acc(\mathcal{NL}_v)), (W_{rEL_i}, acc(\mathcal{EL}_v)), (W_{acc(\mathcal{E}_L)}, acc(\mathcal{NL}_v)) \right\} \cup \Pi_{GAT,i}.
10: \text{end for}
11: \text{Compute the witness for the node property } rNL_n \text{ as } W_{rNL_n} = g^{(s_k + \mu(h(v_n)))} \prod_{p \in P \cap NL_n} g^{(s_k + \mu(h(v_n)))} \text{ and } \Pi_{vn} = (W_{rNL_n}, acc(\mathcal{NL}_v)).
12: \text{return } \Pi = (\Pi_0, \Pi_1, \ldots, \Pi_{n-1}, \Pi_{vn}).
13: \textbf{end procedure}

4.2.3 Label-Pattern Matching query

We consider the following graph pattern queries. Given a graph database GD and a graph pattern G_q, a pattern matching query computes the set of matches of G_q in GD where G_q is a directed graph with labeled nodes. A graph pattern can be written as \(Q = (G_q, A_S)\) where G_q is a graph and A_S is the set of labels on nodes of G_q. In a pattern matching query, the structure of the pattern graph and node labels are known to the client. Since the nodes and edges of the graph database contain data, a client can also make queries to retrieve data from nodes and edges for the match pattern if it is found. The main idea behind the construction of a proof of an answer of a graph pattern query without node and edge property request is to make of use only the node property accumulator and the outgoing node accumulator. In the proof construction, the server constructs witnesses with respect to queried node labels and outgoing edges of the answer. Since the client knows the node labels of the pattern graph, the client uses node labels and the answer to verify the correctness of the answer. For the pattern matching query with node and edge property re-
Therefore, the probability that a computationally bounded adversary needs to generate at least one false proof at a node for node label, edge label, node properties, or edge properties from the nodes and edges of the pattern graph is \( \neg \text{neg} \). Moreover, the probability that a computationally bounded adversary can create a false proof of a label-pattern matching query is \( \neg \text{neg} \).

**Algorithm 4** Proof verification of an LCRQ

1: procedure **Proof_Verification** 
2: \( (v_0 \overset{A}{\rightarrow} v_n, \Pi) \)
3: for \( i = 0 \) to \( n - 1 \) do
4: Compute \( \omega_i, N_{L_i} \)\( = \prod_{\text{Prop} \in \Pi_i} \omega_{N_{L_i}}^{g(h_{E}(v_{i+1}, n_{lb}(v_{i+1})))} \) using the ECRH function and \( pk \)
5: Check \( e(W_i, \omega_i, N_{L_i}) = e(\text{acc}(N_{L_v}), g) \)
6: Compute \( \omega_{ i+1} = \prod_{\text{Prop} \in \Pi_{i+1}} \omega_{N_{L_{i+1}}}^{g(h_{E}(v_{i+1}, n_{lb}(v_{i+1})))} \) using the ECRH function and \( pk \).
7: Check \( e(W_{i+1}, \omega_{i+1}, N_{L_{i+1}}) = e(\text{acc}(N_{L_v}), g) \)
8: Compute \( \omega = g^{ak}(\text{acc}(\mathcal{E}_{L_v})) \)\( = e(\zeta(\omega_{i+1}), g), j = 1, 2, \ldots, \lceil \frac{k}{2} \rceil \)
9: Compute the witness for the required node labels, denoted by \( W_{\text{acc}(\mathcal{E}_{L_v})}, \text{acc}(\mathcal{N}_{L_v}), \omega \)\( = e(\text{acc}([A_v]), g) \)
10: Compute \( \omega = g^{ak} \cdot g^0(\text{acc}(\mathcal{E}_{L_v})) \)\( = e(\zeta(\omega_{i+1}), g), j = 1, 2, \ldots, \lceil \frac{k}{2} \rceil \)
11: end procedure
12: return accept if all bilinear quality check passes, Else return reject.

**Algorithm 5** Graph pattern matching query proof construction

1: Input: Query \( Q, \text{ans}(Q) \) and a set of node labels for \( G_Q \)
2: Output: Proof \( \Pi_{\text{ans}(Q)} \)
3: procedure **Proof_Generation** 
4: Assume \( \text{ans}(Q) = G_Q = (V_{G_Q}, E_{G_Q}) \) is the graph pattern with \( n \) nodes and labels \( \{i_1, \ldots, i_k\} \)
5: for each node \( v \in V_{G_Q} \) do
6: Let \( rNP_v = \{prop_1, \ldots, prop_l\} \) be the required node properties of \( v \).
7: Compute the witness for the required node labels, denoted by \( W_{rNP_v} \) using \( \Pi_{rNP_v} = (W_{rNP_v}, \text{acc}(\mathcal{N}_{L_v})) \)
8: Let \( U = \{u_1, u_2, \ldots, u_k\} \) be the adjacent nodes of \( v \) in \( \text{ans}(Q) \) and \( ON(v) \) be the outgoing edges of \( v \).
9: for each \( u \in U \) do
10: Let \( rEL_{(v,u)} = \{nlbl_{i_1}, \ldots, nbl_{i_k}\} \) be the required edge properties
11: Compute the witness for the required edge labels, denoted by \( W_{rEL_{(v,u)}} = (W_{rEL_{(v,u)}}, \text{acc}(\mathcal{E}_{L_{(v,u)}})) \)
12: Compute \( \Pi_{(v,u)} = (W_{rEL_{(v,u)}}, \text{acc}(\mathcal{E}_{L_{(v,u)}})) \)
13: end for
14: end for
15: Return Proof \( \Pi_{\text{ans}(Q)} = \{\Pi_v : v \in V_{G_Q}\} \).
16: end procedure

**Algorithm 6** Graph pattern query proof verification

1: procedure **Proof_Verification** 
2: \( (Q, \text{ans}(Q), \Pi_{\text{ans}(Q)}) \)
3: Client first run a graph isomorphism algorithm to check \( \text{ans}(Q) \) is isomorphic to the queried graph \( G_Q \).
3: if \( \text{ans}(Q) \cong G_Q \) then
4: for each node \( v \in \text{ans}(Q) \) do
5: Verify each bilinear equality in \( \Pi_{\text{ans}(Q)} \) using respective answers and witnesses.
6: end for
7: if all bilinear verifications are passed then
8: Return accept
9: end if
10: end if
11: Return false.
12: end procedure

**Theorem 1** Assuming the existence of an (extractable) collision-resistance hash function and a collision-resistant bilinear accumulator. Then there exists an authenticated data structure for graph databases that supports update, refresh, query and verify operations on graph databases.

5. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of the proposed scheme. More specifically, our experiment focus on
the computation time of node accumulations, proof generation and its verification. We have developed three different graph databases using the wiki-Vote graph, Slashdot social graph and citation graph from the Stanford SNAP data records.

We provide a description of the construction of three graph databases below. We start by providing our experimental setup.

**EXPERIMENTAL SETUP.** We have implemented the scheme in C using PBC library [1] for using pairing operations. In the PBC library, for pairing, we choose the symmetric pairing where $G_1 = G_2$ and each bilinear group element is an element of an elliptic curve and is of size 128 bytes. The security level offered by this is 128-bit. We use SHA-256 hash function to compute the hash digests on graph data. We truncated the output of the hash function to 160 bits. The experiments were conducted on a 64-bit Desktop with 3.60GHz Intel Core i7 CPU and 12 GB RAM.

**Table 1: Graph database and accumulation tree statistics**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Wiki-Gdb</th>
<th>cit-HepPh-Gdb</th>
<th>slashDot-Web-Gdb</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>71415</td>
<td>810549</td>
<td>818971</td>
</tr>
<tr>
<td># Edges</td>
<td>105689</td>
<td>421578</td>
<td>544474</td>
</tr>
<tr>
<td>Acc. tree depth</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td># Node prop.</td>
<td>10/50/100/200</td>
<td>10/50/100/200</td>
<td>10/50/100/200</td>
</tr>
<tr>
<td># Edge prop.</td>
<td>5/10/20/50</td>
<td>5/10/20/50</td>
<td>5/10/20/50</td>
</tr>
</tbody>
</table>

**GRAPH DATA DESCRIPTION.** A graph database can be viewed as a combination of a graph and a set of data at nodes and edges. We use Wikipedia Vote (wiki-vote) graph, Arxiv High Energy Physics paper citation graph and Slashdot social graph to build three graph databases and denote these databases by Wiki-Gdb, cit-HepPh-Gdb and slashDot-Web-Gdb, respectively. The number of nodes and edges of these graphs can be found in Table 1 and the in degree and out degree distribution of those three graphs can be found in Table 1. For the Wiki-vote graph in Table 1, there is only one label named “voted”, but we make a little modification on the labeling on edges, we randomly assign five different labels on the edges for executing label-constrain queries. We denote the edge labels by “voted i”, $1 \leq i \leq 5$ and we label a node by “person”. In Wiki-Gdb, at each edge, we assign five random properties and at each node, assign ten property values.

For cit-HepPh-Gdb, the node and edge labels are “paper” and “cited”, respectively. We randomly generate ten property values for each node and five property values for each edge, respectively. In slashDot-Web-Gdb, the edge labels are “friend” and “foe” and the number of node properties and edge properties are ten and five, respectively. Moreover we constructed three more variants for each of these three graph databases by assigning the following number of node and edge properties (50, 10), (100, 20) and (200, 50).

**EXPERIMENTAL RESULTS.** The proposed scheme consists of a node accumulation algorithm, authenticated tree building algorithm, proof generation and proof verification algorithms. We use the above three graph databases and measure the average timing for each algorithm. We built the authenticated hash tree on top of the graph database. For Wiki-Gdb, cit-HepPh-Gdb and slashDot-Web-Gdb, the authenticated tree construction time is about 5.116 mins, 22.71 mins, and 35.65 mins respectively including the computation time for node accumulation values and the key generation. We present the average timing for each function in Table 2. We also conducted an experiment on the timing for the construction of accumulation tree for other three graph databases. Figure 3 presents the time required to construct the authenticated tree for the above the number of node and edge properties.

We have performed an experiment on the proof generation and its verification algorithm to measure the time for different number of node properties, edge properties and outgoing edges. We generate the the proof and verify the result till the node accumulation. Note that the node accumulation time and proof generation time depend on the numbers of node properties, edge properties, incoming and outgoing edges and what are being retrieved from the node. For the node property experiment, the proof construction is performed for half of the node properties. Similarly, for the edge properties and outgoing edges, the proof generation is done for the half of the edge properties and half of the outgoing edges, respectively. In the edge property experiment, we have varied different number of edge properties for an outgoing edge. From the construction of the node accumulation we can observer that if we store the edge accumulation values then the witness generation for node properties, outgoing and incoming edges are very efficient. This is also confirmed by our experiment as shown in Figure 2. We provide the proof generation and proof verification time in Figures 2 where OutE, InE, NP, and EP denote the numbers of outgoing edges, incoming edges, node properties and edge properties, respectively.

According to our encoding scheme, the proof size for a node query is $(512 + (\frac{1}{2^n} - 1) \cdot 256)$-byte where 512 is the proof size up to accumulation nodes. The proof size of an edge query is $(768 + (\frac{1}{2^n} - 1) \cdot 256)$-byte where 768 is the proof size till node accumulation value. For a node property/label and edge property/label related information, the proof size is at most $(1280 + (\frac{1}{2^n} - 1) \cdot 256)$-byte. In particular, for Wiki-Gdb, the proof size for a node property query property information is 1920 bytes. For cit-HepPh-Gdb, the proof size for a node query and edge query is 2816 bytes. For slashDot-Web-Gdb, the proof size for a node and edge query property information is 2560 bytes.

To demonstrate the effectiveness of clustering based authenticated hash tree construction, we executed label-constrain queries to measure the proof size reduction. We use Gremlin as a query language to execute label-constrain queries on the above three databases. We use the clustering algorithm in [13] and followed the steps described in Section 3.2 to find the ordering of graph nodes in the authenticated tree. We choose the number of clusters to be 3 and 5 and executed label-constrain queries of lengths 5 and 7. We compared the proof sizes for the same query for the ID-based accumulation tree construction and cluster-based accumulation tree construction. Our results show that the proof size for the cluster-based accumulation tree construction is less than or equal to the proof size for the identifier-based accumulation tree construction in 74% – 82% cases.

**6. CONCLUSIONS**

In this paper we have studied the problem of outsourcing graph databases where nodes and edges contain data. We presented an authenticated data structure scheme for
Table 2: Timing for accumulation tree construction, proof generation, proof verification algorithms on three graph datasets

<table>
<thead>
<tr>
<th>Database</th>
<th>Node acc. proof</th>
<th>Acc. tree proof</th>
<th>Proof Gen</th>
<th>Proof Vrf</th>
<th>Node acc.</th>
<th>Building acc. tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wiki-Gdb</td>
<td>567 ms</td>
<td>56 ms</td>
<td>623 ms</td>
<td>138 ms</td>
<td>304.54 sec</td>
<td>2.42 sec</td>
</tr>
<tr>
<td>cit-HepPh-Gdb</td>
<td>227 ms</td>
<td>72 ms</td>
<td>299 ms</td>
<td>329 ms</td>
<td>1304.64 sec</td>
<td>20.03 sec</td>
</tr>
<tr>
<td>slashDot-Web-Gdb</td>
<td>605 ms</td>
<td>79 ms</td>
<td>684 ms</td>
<td>429 ms</td>
<td>2008.17 sec</td>
<td>40.84 sec</td>
</tr>
</tbody>
</table>

Figure 2: Timing for generating proofs and verifying proofs when half the number of node or edge properties where chosen

(a) Timing for node and edge properties when OutE = 200, InE = 200, NP = 200, and EP = 25
(b) Timing for out edge properties when OutE = 300, InE = 250 and NP = 200 and EP = 20

Figure 3: Timing for the construction of accumulation trees for three graph databases for different number of node properties and edge properties

the integrity of the graph database and graph queries. Our approach exploits the structure of the graph database for the construction of the authenticated hash tree. We considered label-constrain graph queries such as LCRQs and pattern matching queries and presented the construction of the proof generation and its verification algorithms. The proof size of an query answer is proportional to that of the answer. Our scheme incurs an extra storage of $O(M)$ for the security functionalities and communication overhead $O(n)$ where $n$ is the size of the answer. We have implemented the proposed scheme in C using the PBC library to evaluate its performance.

7. REFERENCES


GraphDB. http://ontotext.com/products/ontotext-graphdb/graphdb-standard/


OrientDB. http://orientdb.com/

APPENDIX

A. NODE RELATED ACCUMULATORS

In this section we present the definition of node related accumulators.

DEFINITION 7 (Collision Resistant Node Prop. Acc). Given the set \( \mathcal{N} = \{n_1, n_2, ..., n_m\} \) constructed from the set of node labels \( \{v_i, 1 \leq i \leq m\} \) where \( n_i \in \mathbb{Z}_p^* \) and the bilinear parameter \( \text{pub} = (\pi, \mathbb{G}, \mathbb{G}_T, e, g) \) chosen according to security parameter \( \lambda \) and the elements of \( \mathbb{G}_T \) \( \{g, g^2, g^3, ..., g^s\} \) for some randomly chosen \( s \in \mathbb{Z}_p^* \), the probability that a computationally bounded adversary can find...
for another set of node labels such that $\text{acc}(N) = \text{acc}(N')$ is neg($\lambda$).

**Definition 8 (Collision Resistant Edge Label Acc).**
Given a set of edge labels $E = \{e_1, e_2, ..., e_n\}$ where $e_i \in \mathbb{Z}_p^*$ and the bilinear parameter $\text{pub} = (p, G, G_T, e, g)$ chosen according to security parameter $\lambda$ and the elements of $G_T \{g, g^s, g^{s^2}, ..., g^{s^q}\}$ for some randomly chosen $s \in \mathbb{Z}_p^*$, the probability that a computationally bounded adversary can find another set of edge labels $E'$ such that $\text{acc}(E) = \text{acc}(E')$ is neg($\lambda$).

**Definition 9 (Collision Resistant Adj. Node Acc).**
Given a set of outgoing vertex identifiers values $ON_{v_j} = \{a_1, a_2, ..., a_n\}$ where $a_i \in \mathbb{Z}_p^*$, the bilinear parameter $\text{pub} = (p, G, G_T, e, g)$ chosen according to security parameter $\lambda$ and the elements of $G_T \{g, g^s, g^{s^2}, ..., g^{s^q}\}$ for some randomly chosen $s \in \mathbb{Z}_p^*$, the probability that a computationally bounded adversary can find another set of adjacent vertices of $ON_{v_j}'$ such that $\text{acc}(ON_{v_j}) = \text{acc}(ON_{v_j}')$ is neg($\lambda$). Similarly for incoming adjacent edges.