

3-2014

# Security of Graph Data: Hashing Schemes and Definitions

Muhammad U. Arshad  
*Purdue University, marshad@purdue.edu*


Ashish Kundu  
*Purdue University*

Elisa Bertino  
*Purdue University, bertino@cs.purdue.edu*

Krishna Madhavan  
*Network for Computational Nanotechnology, Purdue University*

Arif Ghafoor  
*Purdue University School of Electrical and Computer Engineering*

Follow this and additional works at: <http://docs.lib.purdue.edu/ccpubs>

 Part of the [Engineering Commons](#), [Life Sciences Commons](#), [Medicine and Health Sciences Commons](#), and the [Physical Sciences and Mathematics Commons](#)

---

Arshad, Muhammad U.; Kundu, Ashish; Bertino, Elisa; Madhavan, Krishna; and Ghafoor, Arif, "Security of Graph Data: Hashing Schemes and Definitions" (2014). *Cyber Center Publications*. Paper 627.  
<http://dx.doi.org/10.1145/2557547.2557564>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

# Security of Graph Data: Hashing Schemes and Definitions

Muhammad U. Arshad<sup>†1</sup>, Ashish Kundu<sup>‡2</sup>, Elisa Bertino<sup>†3</sup>, Krishna Madhavan<sup>†4</sup>, Arif Ghafoor<sup>†5</sup>

<sup>†</sup>Purdue University  
West Lafayette, Indiana, USA  
{<sup>1</sup>marshad, <sup>3</sup>bertino, <sup>4</sup>cm, <sup>5</sup>ghafoor}@purdue.edu

<sup>‡</sup>IBM T J Watson Research Center  
Yorktown Heights, New York, USA  
<sup>2</sup>akundu@us.ibm.com

## ABSTRACT

Use of graph-structured data models is on the rise – in graph databases, in representing biological and healthcare data as well as geographical data. In order to secure graph-structured data, and develop cryptographically secure schemes for graph databases, it is essential to formally define and develop suitable collision resistant one-way hashing schemes and show that they are efficient. The widely used Merkle hash technique is not suitable as it is, because graphs may be directed acyclic ones or cyclic ones. In this paper, we are addressing this problem. Our contributions are: (1) define the practical and formal security model of hashing schemes for graphs, (2) define the formal security model of perfectly secure hashing schemes, (3) describe constructions of hashing and perfectly secure hashing of graphs, and (4) performance results for the constructions. Our constructions use graph traversal techniques, and are highly efficient for hashing, redaction, and verification of hashes graphs. We have implemented the proposed schemes, and our performance analysis on both real and synthetic graph data sets support our claims.

## Categories and Subject Descriptors

D.4.6 [Security and Protection]: *Cryptographic controls*

## Keywords

Integrity; Hash Functions; Perfectly Secure Hash Functions; Graphs; Privacy

## 1. INTRODUCTION

One of the fundamental building blocks of modern cryptography is hash function. Hash functions are used towards verification of data integrity as well as message authentication codes and digital signature schemes. Traditional hash functions handle messages as bit strings.  $\mathcal{H} : \{0, 1\}^* \rightarrow \{0,$

$1\}^n$  defines a hash function that takes a bitstring of any size as input and outputs a hash-value of size  $n$ -bits. Integrity protection of data includes not only data objects that are bit strings but also data objects that are graphs.

Some of the several contexts where graph-structured data models are widely used are healthcare, biological, geographical and location data as well as financial databases [6, 18]. Often it is required to check if a given graph has been updated or not or whether two graphs are identical<sup>1</sup>. Hashing is used to carry out these operations [12]. The problem of hashing graphs is of great immediate practical value and just not of conceptual value – programmers are looking for techniques that they can implement in order to compute hash values of directed graphs<sup>2</sup>. Moreover, authenticating graph-structured data is often a crucial security requirement for databases [17]. Hashing is used as a basic building block for authentication schemes based on hash-and-sign paradigm. In this paper, we have focused on hashing schemes (integrity) and not on authenticity.

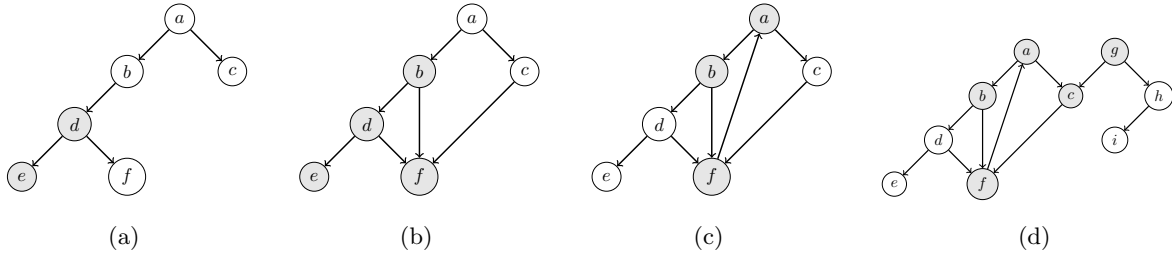
In case of hashing schemes such as SHA-1, SHA-2, a message is either shared completely or not shared at all with the user. In contrast, when graphs are used, a user may receive part(s) of a graph (subgraph(s)). Graph models are often used for representing sensitive data as well (such as healthcare, biological data). Traditional hashing schemes leak information (for SHA1 and SHA2 leak information about the length of the plaintext [2]). The challenge is how to hash graphs so that no information is leaked at least in the context of probabilistic polynomial adversaries [8]. The Merkle hash technique (MHT) has been proposed as an approach for computing hashes for trees [15] and has been extended for directed acyclic graphs [14].

Cryptographic hashing techniques for cyclic graphs have not been well-studied in the existing literature. There are schemes for hashing trees and directed acyclic graphs: Merkle hash technique, and Search DAG, which is based on MHT. Existing schemes rely on the fact that updates do not change a tree into a non-tree graph and a DAG to a cyclic graph. That restriction is quite strong – it limits the operations that can be carried out on a graph database. Moreover, it is desirable to have one hashing scheme that is applicable to any type of graph, which helps simplifying security of graph

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
*CODASPY'14*, March 3–5, 2014, San Antonio, Texas, USA.  
Copyright 2014 ACM 978-1-4503-2278-2/14/03 ...\$15.00.  
<http://dx.doi.org/10.1145/2557547.2557564>.

<sup>1</sup>Identical graphs are isomorphic graphs, but the reverse is not true [4].

<sup>2</sup><http://stackoverflow.com/questions/14532164/hash-value-for-directed-acyclic-graph>



**Figure 1: Graphs: (a) Tree, (b) DAG, (c) Graph with cycles, (d) Graph with multiple sources (vertices with no incoming edge). The shadowed with dotted boundary are the subgraphs that a user may receive.**

databases; without such a generic yet efficient scheme, graph security techniques need to determine the type of the graph being hashed and then determine which scheme would be applicable. Such a process would not only add cost of software engineering and maintenance for such additional schemes and logic, but also add cost in terms of performance and latency to the security schemes using hashing.

In many application domains, such as health care and military, an additional requirement is to maintain the secrecy of the content from which its hash value: by secrecy, we mean that the hash value of a content does not leak any information of the content. Canetti et al. [3] refer to this as perfectly secure hashing. As in the MHT, when a subtree is shared with a user, the user also receives a set of Merkle hash values for some of the nodes that are not in the shared subtree. If the hash function used is not perfectly collision-resistant, the hash values could lead to leakage of information about the unshared nodes, which needs to be prevented. In addition the user may learn the existence of nodes that he/she is not allowed to access; such knowledge may lead to inferences that may then result in privacy or confidentiality breaches [11]. Encryption is too heavyweight and has different security properties than the properties required for hash functions.

Therefore, in case of graph data, there is a need to define the formal notions of hashing and perfectly secure hashing schemes, and develop constructions for such definitions that are efficient and practical to be used by real-world graph database systems. In addition, the formalization of such secure hashing schemes for graphs is essential towards analyzing different constructions with respect to their security.

In this paper, we make the following contributions: (1) we define the formal security model of hashing schemes for graphs, (2) we define the formal security model of perfectly secure hashing schemes, (3) we have proposed efficient generic constructions of hashing and perfectly secure hashing schemes that can be applied to any types of graphs, and (4) we have carried out experiments, and our performance results for the constructions corroborate that our schemes are highly efficient and can be used in real-world graph databases and graph security schemes.

## 2. RELATED WORK AND BACKGROUND

There are two techniques in the literature that come closest to the constructions presented in this paper: The Merkle hash technique [15] (MHT) and the Search-DAG technique [14] (SDAG).

Integrity assurance of tree-structured data is primarily carried by the Merkle hash technique [15]. We would describe the issues associated with MHT in the next section. Martel et al. [14] proposed SDAG – an authentication technique for directed acyclic graphs referred to as “Search DAGs” in third party distribution frameworks. Their technique uses MHT. The SDAG technique only covers DAGs, not the general directed graphs, which is of greater interest in this paper. Martel et al. do not define security of hashing of directed graphs nor they address perfectly secure hashing of graphs. Kundu et al. [10] have defined the security of leakage-free signatures for graphs, which however does not cover the hashing schemes.

Throughout the paper, we have cited other related works as and when needed.

### 2.1 Hashing Trees with Merkle Hash Technique

The Merkle hash technique [15] works bottom-up. For a node  $x$  in tree  $T(V, E)$ , it computes a Merkle hash (MH)  $mh(x)$  as follows: if  $x$  is a leaf node, then  $mh(x) = \mathcal{H}(c_x)$ ; else  $mh(x) = \mathcal{H}(mh(y_1) \| mh(y_2) \| \dots \| mh(y_m))^3$ , where  $y_1, y_2, \dots, y_m$  are the  $m$  children of  $x$  in  $T$  in that order from left to right. For example, consider the tree in Figure 1(a). The MH for this tree is computed as follows. The MH of  $e$  and  $f$  are computed as  $\mathcal{H}(c_e)$  and  $\mathcal{H}(c_f)$ , respectively, which are then used to compute the MH of  $d$  as  $mh(d) = \mathcal{H}(mh(e) \| mh(f))$ . The MH of  $b$  is computed as  $\mathcal{H}(mh(d))$ . Similarly, the MH of  $c$  and  $a$  are computed as  $\mathcal{H}(c_c)$  and  $\mathcal{H}(mh(b) \| mh(c))$ , respectively.

By using such a technique, only the contents of the leaf nodes can be authenticated. In case the non-leaf (intermediate or root) nodes have contents, the computation of the MH of non-leaf nodes  $x$  involves the MH of all of its children, and either (a) the content of  $x$ , or (b) hash of the content of  $x$  (used in 1(a)). Suppose  $x$  to be a non-leaf node in  $T$ . The MH of  $x$  is defined as follows:  $mh(x) = \mathcal{H}(\mathcal{H}(c_x) \| mh(y_1) \| mh(y_2) \| \dots \| mh(y_m))$ .

Consider again the tree in Figure 1(a). MH of  $d$ ,  $b$  and  $a$  are computed respectively as  $\mathcal{H}(\mathcal{H}(c_d) \| mh(e) \| mh(f))$ ,  $\mathcal{H}(\mathcal{H}(c_b) \| mh(d))$ , and  $\mathcal{H}(\mathcal{H}(c_a) \| mh(b) \| mh(c))$ .

#### 2.1.1 Integrity Verification

Let  $T_\delta$  be a subtree of tree  $T$  as shown shadowed to be shared with a user. The following set of *verification objects*  $\mathcal{VO}$  is also sent to the user, for integrity verification of  $T_\delta$ .

<sup>3</sup>Henceforth,  $\mathcal{H}$  denotes a one-way, collision-resistant hash function, and  $\|$  stands for concatenation

Such *auxiliary information* constitutes the information leakages by MHT (Figure 1). The user then computes the MH of the whole tree using such information (the subtree and auxiliary information) and verifies the hash of the original tree using this MH.

1. Let  $x$  be a node in  $T_\delta$ . The MH of each sibling of  $x$  that is in  $T$  but not in  $T_\delta$ . (e.g.,  $mh(f)$  w.r.t.  $e$ )
2. The MH of each sibling  $y$  of each ancestor  $x$ , such that  $y$  is not in  $T_\delta$ . (e.g.,  $mh(c)$  and  $mh(f)$  w.r.t.  $e$ )
3. The hash of the content of each ancestor of  $x$ . (e.g.,  $\mathcal{H}(c_a)$  and  $\mathcal{H}(c_b)$  w.r.t.  $x$ )
4. (i) The structural order between a node  $x$  in  $T_\delta$  and its sibling(s) that are not in  $T_\delta$ , and (ii) the structural order between the sibling nodes that are not in  $T_\delta$ . (e.g., (i) the order between  $e$  and  $f$ , and (ii) the order between  $b$  and  $c$ .)
5. (i) Parent-child/ancestor-descendant relationship(s) between a node in  $T_\delta$  and another node not in  $T_\delta$  and (ii) those between the nodes that are not in  $T_\delta$ . (e.g., the relationships (i) between  $b$  and  $d$ , (ii) between  $a$  and  $b$ , and between  $a$  and  $c$ .)
6. The fact that a given node is the root of  $T$  (even if it is the root of  $T_\delta$ ).

The user then computes the MH of the whole tree using such information (the subtree and  $\mathcal{VO}$ ) and compares it with the received signed MH of the root. If they are equal, the integrity of the subtree is validated. Moreover, this process verifies the integrity of the subtree against the original tree.

## 2.2 Hashing Graphs

In graphs, nodes may have multiple incoming edges, there may be no roots or source nodes; perhaps, there may be a number of roots or source nodes. The major challenges in hashing graphs arise out of the fact that (i) a graph may have cycles, (ii) changes to the nodes affect hashes of multiple other nodes, and (iii) a graph can be shared with a user in parts or in terms of subgraphs. For example, in Figure 1, subgraphs that are shared are shown as shadowed regions on each of the graphs.

Consider the DAG in Figure 1(b). Node  $f$  has three incoming edges. If we use the SDAG technique, then the hash of node  $f$  influences the hashes of three other nodes from which these nodes originate:  $d$ ,  $b$ , and  $c$ . While that is acceptable, any update to  $f$  affects the hashes of all nodes other than  $e$ . In general in the SDAG technique, an update to a node in a DAG affects  $|V| + |E|$  nodes.

Hashing graphs with cycles is more complex because such graphs cannot be ordered topologically [4]. For the graphs in Figure 1 (c) and (d), the cycles involve nodes  $a$ ,  $b$ ,  $c$  and  $f$ . The issue is then how to hash such graphs so that the cyclic structure is also preserved in the hash – dissimilar structures should not have the same hash even if the contents may be identical.

## 2.3 Formal Security Models

Existing approaches [15, 14] do not formally define the notion of graph hashing, nor they define how to construct perfectly collision-resistant graph hash functions. Canetti et

al. [2, 3] developed the notion of perfectly collision-resistant hash functions for messages that are bit-strings and are not graphs nor any structured/semi-structured objects. Canetti et al. [3] also proposed that the *verify* method should be included in the definition of random oracles. In this paper, we also propose a similar *hash-verify* method as well as a *hash-redact* method that is specific for structured/semi-structured data.

## 3. TERMINOLOGY

*Trees and Graphs:* A directed graph  $G(V, E)$  is a set of nodes (or vertices)  $V$  and a set of edges  $E$  between these nodes:  $e(x, y)$  is an edge from  $x$  to  $y$ ,  $(x, y) \in V \times V$ . Undirected graphs can be represented as directed graphs. Therefore, in what follows we consider only the case of directed graphs and we will use the term graph with the meaning of directed graph. A node  $x$  represents an atomic unit of data, which is always shared as a whole or is not shared at all. A source is a node that does not have any incoming edge. A node  $x$  is called the ancestor of a node  $y$  iff there exists a path consisting of one or more edges from  $x$  to  $y$ . Node  $x$  is an immediate ancestor, also called parent, of  $y$  in  $G$  iff there exists an edge  $e(x, y)$  in  $E$ . Nodes having a common immediate ancestor are called siblings. Let  $G(V, E)$  and  $G_\delta(V_\delta, E_\delta)$  be two graphs. We say that  $G_\delta(V_\delta, E_\delta)$  is a *redacted subgraph* of  $G(V, E)$  if  $G_\delta(V_\delta, E_\delta) \subseteq G(V, E)$ .  $G_\delta(V_\delta, E_\delta) \subseteq G(V, E)$  if and only if  $V_\delta \subseteq V$  and  $E_\delta \subseteq E$ . Also  $G_\delta(V_\delta, E_\delta) \subset G(V, E)$  if and only if  $V_\delta \cup E_\delta \subset V \cup E$ . A redacted subgraph  $G_\delta(V_\delta, E_\delta)$  is derived from the graph  $G(V, E)$  by redacting the set of nodes  $V \setminus V_\delta$  and the set of edges  $E \setminus E_\delta$  from  $G$ . A directed tree  $T(V, E)$  is a directed graph with the following constraint: removal of any edge  $e(x, y)$  from  $E$  leads to two disconnected trees with no edge or path between nodes  $x$  and  $y$ . As in the case of graphs, a redacted subtree of tree  $T(V, E)$  denoted by  $T_\delta(V_\delta, E_\delta)$  is such that  $T_\delta(V_\delta, E_\delta) \subseteq T(V, E)$ .  $T_\delta(V_\delta, E_\delta) \subseteq T(V, E)$  denotes that  $V_\delta \subseteq V$  and  $E_\delta \subseteq E$ . Redacted subgraph  $T_\delta(V_\delta, E_\delta)$  is derived from the tree  $T(V, E)$  by redacting the set of nodes  $V \setminus V_\delta$  and the set of edges  $E \setminus E_\delta$  from  $T$ . Two trees/graphs/forests with the same nodes and edges, but *different ordering* between at least one pair of siblings are different trees/graphs/forests.

## 4. REVIEW OF STANDARD HASHING SCHEMES

In this section, we review the standard definition of hashing schemes (adopted from [8]). A standard hashing scheme  $\Pi$  is a tuple  $(\mathbf{Gen}, \mathcal{H})$ . ( $\mathbf{s}$  is not a secret key in standard cryptographic sense.)

*Definition 1. (Standard hashing scheme)* A hashing scheme  $\Pi$  consists of two probabilistic polynomial-time algorithms  $\Pi = (\mathbf{Gen}, \mathcal{H})$  satisfying the following requirements:

**KEY GENERATION:** The probabilistic key generation algorithm  $\mathbf{Gen}$  takes as input a security parameter  $1^\lambda$  and outputs a key  $\mathbf{s}$ :  $\mathbf{s} \leftarrow \mathbf{Gen}(1^\lambda)$ .

**HASH:** There exists a polynomial  $l$  such that  $\mathcal{H}$  takes a key  $\mathbf{s}$  and a string  $x \in \{0, 1\}^*$  and outputs a string  $\mathcal{H}^{\mathbf{s}}(x) \in \{0, 1\}^{l(\lambda)}$  (where  $\lambda$  is the value of the security parameter implicit in  $\mathbf{s}$ ).

If  $\mathcal{H}^s$  is defined only for inputs  $x \in \{0,1\}^{l'(\lambda)}$  and  $l'(\lambda) > l(\lambda)$ , then we say that  $(\mathbf{Gen}, \mathcal{H})$  is a fixed-length hash function for inputs of length  $l'(\lambda)$ .

## 4.1 Security of Hash Functions

The strongest form of security for hash functions include three properties: (1) collision resistance, (2) second pre-image resistance, and (3) pre-image resistance. Collision resistance is the strongest notion and subsumes second pre-image resistance, which in turn subsumes pre-image resistance [8]. In other words, any hash function that satisfies (2) satisfies also (3) but the reverse is not true; and any hash function that satisfies (1) satisfies also (2) (and transitively (3)) but the reverse is not true. In the rest of the paper, security of hash functions refer to the strongest property, i.e., collision resistance.

For the traditional hash functions defined in the previous section, we are now reviewing the collision-finding experiment (adapted from [8]). In the rest of the paper,  $\mathcal{A}$  is a probabilistic polynomial-time (PPT) adversary.

### Collision-finding Experiment: H-Coll $_{\mathcal{A},\Pi}(\lambda)$

1. Key  $\mathbf{s} \leftarrow \mathbf{Gen}(1^\lambda)$
2.  $\mathcal{A}$  is given  $\mathbf{s}$  and outputs  $x$  and  $x'$ . (If  $\Pi$  is a fixed-length hash function for inputs of length  $l'(\lambda)$  then  $x, x' \in \{0,1\}^{l'(\lambda)}$ )
3. The output of the experiment is 1 if and only if  $x \neq x'$  and  $\mathcal{H}^s(x) = \mathcal{H}^s(x')$ . In such a case, we say that  $\mathcal{A}$  has found a collision for  $\mathcal{H}^s$ ; else the output of the experiment is 0.

*Definition 2.* A hash function  $\Pi = (\mathbf{Gen}, \mathcal{H})$  is collision-resistant if for all PPT adversaries  $\mathcal{A}$  there exists a negligible function<sup>4</sup>  $\mathbf{negl}$  such that

$$\Pr[\text{H-Coll}_{\mathcal{A},\Pi}(\lambda)] \leq \mathbf{negl}(\lambda)$$

## 5. COLLISION-RESISTANT HASHING OF GRAPHS

The standard definition of hashing schemes cannot be applied directly to graphs because the standard definition operates on messages  $x \in \{0,1\}^*$  and each message is shared either fully or not shared at all with a user. In contrast, a graph  $G(V, E)$  is a set of nodes and edges, where each node may be represented by  $x$ , and a user may have access to one or more subgraphs instead of the complete graph.

As discussed earlier, a hashing scheme for graphs requires a key generation algorithm and a hash function algorithm. We would also need a method to verify hashes for a graph as well as its subgraphs. We refer to this algorithm as the “hash-verification” method. This is because as mentioned earlier, users may receive entire graphs or subgraphs, and need to verify their integrity. To that end, if the hash function used to compute the hash of a graph is a collision-resistant hash function, then the user would need certain extra information along with the proper subgraphs. Otherwise, the hash value computed by the user for the received

<sup>4</sup>A function  $\epsilon(k)$  is negligible in cryptography if for every polynomial  $p(\cdot)$ , there exists an  $N$  such that for all integers  $k > N$  it holds that  $\epsilon(k) < \frac{1}{p(k)}$  ([8]:Definition 3.4).

subgraphs would not match the hash value of the graph unless there is a contradiction to the premise that the hash function is collision-resistant. We refer to this extra information as “verification objects”  $\mathcal{VO}$  ???. Computation of the  $\mathcal{VO}$  for a subgraph with respect to a graph is carried out by another algorithm also part of the definition of the hashing scheme for graphs. We call this algorithm as “hash-redaction” of graphs.

The conceptualization of these two algorithms for verification and redaction described in the previous paragraph is already in use by schemes such as the MHT, but have not been formalized. As essential components of our formalization of the notion of graph hashes, we need to formalize these methods. Such formalization is also essential for a correct design and rigorous analysis of the protocols that realize these definitions. The definition of the graph hashing schemes is as follows.

### *Definition 3. (Collision-resistant graph hashing scheme)*

A hashing scheme  $g\Pi$  consists of three PPT algorithms and one deterministic algorithm  $g\Pi = (\mathbf{gGen}, g\mathcal{H}, \mathbf{ghRedact}, \mathbf{ghVrfy})$  satisfying the following requirements:

**KEY GENERATION:** The probabilistic key generation algorithm  $\mathbf{gGen}$  takes as input a security parameter  $1^\lambda$  and outputs a key  $\mathbf{s}$ :  $\mathbf{s} \leftarrow \mathbf{gGen}(1^\lambda)$ .

**HASHING:** The hash algorithm  $g\mathcal{H}$  takes a key  $\mathbf{s}$  and a graph  $G(V, E)$  and outputs a string  $g\mathcal{H}^s(G(V, E)) \in \{0,1\}^{l(\lambda)}$ , where  $l$  a polynomial, and  $\lambda$  is the value of the security parameter implicit in  $\mathbf{s}$ .

**HASH-REDACTION:** The redaction algorithm  $\mathbf{ghRedact}$  is a probabilistic algorithm that takes  $G(V, E)$  and a set of subgraphs  $\mathcal{G}_\delta$  (such that each  $G_\delta \in \mathcal{G}_\delta$ ,  $G_\delta \subseteq G(V, E)$ ) as inputs and outputs a set  $\mathcal{VO}_{\mathcal{G}_\delta, G(V, E)}$  of verification objects for those nodes and edges that are in  $G(V, E)$  but not in any of the subgraphs in  $\mathcal{G}_\delta$ .

$$\mathcal{VO}_{\mathcal{G}_\delta, G(V, E)} \leftarrow \mathbf{ghRedact}(\mathcal{G}_\delta, G(V, E))$$

**HASH-VERIFY:**  $\mathbf{ghVrfy}$  is a deterministic algorithm that takes a hash value  $gH$ , a set of graphs  $\mathcal{G}$ , and a set of verification objects  $\mathcal{VO}$ , and returns a bit  $b$ , where  $b = 1$  if the hash value  $gH$  is a valid hash for  $\mathcal{G}$  and  $\mathcal{VO}$ , and  $b = 0$  otherwise:  $b \leftarrow \mathbf{ghVrfy}^s(gH, \mathcal{G}, \mathcal{VO})$

## 5.1 Correctness

A hashing scheme for graphs is correct if the following properties hold.

**Hashing Correctness (Empty redaction):** For any graph  $G(V, E)$ , any positive integer value of  $\lambda$ , any key  $\mathbf{s} \leftarrow \mathbf{gGen}(\lambda)$ , and any  $gH \leftarrow g\mathcal{H}^s(G(V, E))$ ,  $\mathcal{VO} \leftarrow \mathbf{ghRedact}(\{G\}, G)$ ,  $\mathbf{ghVrfy}^s(gH, \{G(V, E)\}, \mathcal{VO})$  always outputs 1.

**Hash-Redaction Correctness:** For any graph  $G(V, E)$ , any positive integer value of  $\lambda$ , any key  $\mathbf{s} \leftarrow \mathbf{gGen}(\lambda)$ , any set  $\mathcal{G}_\delta$  of subgraphs  $G_\delta(V_\delta, E_\delta) \subseteq G(V, E)$  such that the union of all the subgraphs in  $\mathcal{G}_\delta$  results in a graph that is a proper subgraph of  $G$ , and  $gH \leftarrow g\mathcal{H}^s(G)$ ,  $\mathcal{VO} \leftarrow \mathbf{ghRedact}(\mathcal{G}_\delta, G)$ ,  $\mathbf{ghVrfy}^s(gH, \mathcal{G}_\delta, \mathcal{VO})$  always outputs 1.

## 5.2 Security of Hash Functions

The strongest form of security for hash functions for graphs also includes three properties: (1) collision resistance, (2) second pre-image resistance, and (3) pre-image resistance. As earlier, collision resistance is the strongest notion and subsumes second pre-image resistance, which in turn subsumes pre-image resistance.

**Collision-finding Experiment:**  $\text{GH-Coll}_{\mathcal{A},g\Pi}(\lambda)$

1. Key  $\mathbf{s} \leftarrow \text{gGen}(1^\lambda)$
2.  $\mathcal{A}$  is given  $\mathbf{s}$  and outputs (a)  $G(V, E)$  and  $G'(V', E')$ , and (b)  $\mathcal{VO}_{\mathcal{G}_\delta, G(V, E)} \leftarrow \text{ghRedact}(\mathcal{G}_\delta, G(V, E))$  and  $\mathcal{VO}'_{\mathcal{G}'_\delta, G'(V', E')} \leftarrow \text{ghRedact}(\mathcal{G}'_\delta, G'(V', E'))$
3. The output of the experiment is 1 if and only if any of the following is true: in such a case, we say that  $\mathcal{A}$  has found a collision for  $\mathcal{H}^s$ ; else the output of the experiment is 0.
  - (a)  $G(V, E) \neq G'(V', E')$  and  $gH = gH'$ , where  $gH \leftarrow g\mathcal{H}^s(G)$ , and  $gH' \leftarrow g\mathcal{H}^s(G'(V', E'))$ .
  - (b)  $G(V, E) \neq G'(V', E')$  and  $\text{ghVrfy}^s(gH', \mathcal{G}_\delta, \mathcal{VO}) = g\mathcal{H}^s(gH, \mathcal{G}'_\delta, \mathcal{VO}')$ .

*Definition 4.* A hash function  $g\Pi = (\text{gGen}, g\mathcal{H}, \text{ghRedact}, \text{ghVrfy})$  is collision-resistant if for all PPT adversaries  $\mathcal{A}$  there exists a negligible function  $\text{negl}$  such that

$$\Pr[\text{GH-Coll}_{\mathcal{A},g\Pi}(\lambda) = 1] \leq \text{negl}(\lambda)$$

## 6. PERFECTLY COLLISION-RESISTANT HASHING SCHEME FOR GRAPHS

In this section, we formalize the notion of perfectly one-way (i.e., collision-resistant) hash functions for graphs.

Hash functions used in practice do not hide information about the message being hashed. Canetti [2] showed that there is the need for a hash function that is perfectly one-way, i.e., for which it is hard to find a collision. SHA1, MD5 do not satisfy the “perfectly one-way” property. In this paper, we refer to “perfectly one-way” hash functions as “perfectly collision-resistant” hash functions. Before we define what perfectly collision-resistant hash functions are for graphs, we discuss why such a notion is necessary and what the leakages are.

Standard hashing schemes may leak information about the image being hashed. Even though such schemes are one-way in a computational sense (informally speaking, to find a collision one needs to solve a hard problem or do intractable amount of work), the hash value  $\mathcal{H}(x)$  of image  $x$  reveals some information about  $x$ . Such information leaks in the reverse direction –  $\mathcal{H}(x)$  to  $x$  makes this function “not perfectly one way”; such leakage may allow the attacker to construct pre-images and second-preimages with less work than what was defined by the random oracle model. In the case of sensitive data, such leakages via hash values lead to another security issue: leakage of sensitive information. As Canetti et al. describe in [3], if  $x$  represents a confidential information,  $\mathcal{H}(x)$  may leak the length of  $x$  and bits of  $x$ , which is a serious security breach.

The formal definition of hashing schemes does not capture the requirement of non-leakage of information about pre-images. Canetti has introduced a formal definition for  $x \in$

$\{0, 1\}^*$  and several constructions for perfectly one-way hash functions [3].

However, for graphs no such notion has been defined. Graphs are often used to represent sensitive data, and it is thus essential to hide all the information contained in the nodes. There is another reason for the need of perfectly collision-resistant hash functions: the standard definition operates on messages  $x \in \{0, 1\}^*$  and each message is either fully shared or not shared at all with a user. In contrast, a graph  $G(V, E)$  is a set of nodes and edges, where each node may be represented by  $x$ , and a user may have access to one or more subgraphs instead of the complete graph. As in the MHT, when a subtree is shared with a user, the user also receives a set of MH values for some of the nodes that are not in the shared subtree. If the hash function used is not perfectly collision-resistant, then the hash values could lead to leakage of information about the unshared nodes, which needs to be prevented. Encryption is too heavyweight and has different security properties than those required for hash functions. We thus need perfectly collision-resistant hash functions for graphs.

In the previous section, we formally defined the notion of collision-resistant graph hashing schemes. The definition of a perfectly collision-resistant graph hash function is identical to this definition but includes an extra element: a key that is used towards making the scheme perfectly collision-resistant (as well as hiding).

*Definition 5. (Perfectly collision-resistant graph hashing scheme)* A hashing scheme  $pg\Pi$  consists of three probabilistic polynomial-time algorithms and one deterministic algorithm  $pg\Pi = (\text{pgGen}, pg\mathcal{H}, \text{pghRedact}, \text{pghVerify})$  satisfying the following requirements:

**KEY GENERATION:** The probabilistic key generation algorithm  $\text{pgGen}$  takes as input security parameter  $1^\lambda$  and outputs a key  $\mathbf{s}$ .

$$\mathbf{s} \leftarrow \text{pgGen}(1^\lambda).$$

**RANDOMIZER GENERATION:** The probabilistic randomizer generation algorithm  $\text{pgRandom}$  takes as input security parameter  $1^{\lambda_r}$  and outputs a uniformly chosen random  $\mathbf{r} \in \{0, 1\}^{\lambda_r}$ .

$$\mathbf{r} \leftarrow \text{pgRandom}(1^{\lambda_r}).$$

**HASHING:** There exists a polynomial  $l$  such that the hash algorithm  $pg\mathcal{H}$  takes keys  $\mathbf{s}$  and  $\mathbf{r}$  along with a graph  $G(V, E)$  as input and outputs a string  $pgH$ .

$$pgH \leftarrow pg\mathcal{H}^{\mathbf{s}, \mathbf{r}}(G(V, E)) \in \{0, 1\}^{l(\lambda, \lambda_r)}.$$

**HASH-REDACTION:** The redaction algorithm  $\text{ghRedact}$  is a probabilistic algorithm that takes  $G(V, E)$  and a set of subgraphs  $\mathcal{G}_\delta$  (such that each  $G_\delta \in \mathcal{G}_\delta$ ,  $G_\delta \subseteq G(V, E)$ ) as inputs and outputs a set  $\mathcal{VO}_{\mathcal{G}_\delta, G(V, E)}$  of verification objects for those nodes and edges that are in  $G(V, E)$  but not in any of the subgraphs in  $\mathcal{G}_\delta$ .

$$\mathcal{VO}_{\mathcal{G}_\delta, G(V, E)} \leftarrow \text{pghRedact}(\mathcal{G}_\delta, G(V, E)).$$

**HASH-VERIFY:**  $\text{pghVerify}$  is a deterministic algorithm that takes a hash value  $pgH$ , a set of graphs  $\mathcal{G}$ , and a set of verification objects  $\mathcal{VO}$ , and returns a bit  $b$ , where  $b = 1$  meaning valid if the hash value  $pgH$  is a

valid hash for  $\mathcal{G}$  and  $\mathcal{VO}$ , and  $b = 0$  meaning invalid hash value.

$$b \leftarrow \text{pghVerify}^{s,x}(\text{pg}H, \mathcal{G}, \mathcal{VO})$$

## 6.1 Correctness

A perfectly collision-resistant hashing scheme for graphs is correct if the following properties hold: Hashing Correctness (Empty redaction) and Hash-Redaction Correctness. These two properties have definitions similar to the definitions of such properties for the collision-resistant hash function for graphs in section 5.1.

## 6.2 Security

There are two security requirements: (1) collision-resistance, and (2) semantically perfect one-wayness of graph hash functions. The collision-resistance experiment is similar to the one defined earlier in Section 5.2.

**Collision-finding Experiment:**  $\text{PGH-Coll}_{\mathcal{A}, \text{pg}\Pi}(\lambda, \lambda_r)$

1. Key  $\mathbf{s} \leftarrow \text{pgGen}(1^\lambda)$
2. Randomizer  $\mathbf{r} \leftarrow \text{pgRandom}(1^{\lambda_r})$
3.  $\mathcal{A}$  is given  $\mathbf{s}, \mathbf{r}$ ;  $\mathcal{A}$  outputs (a)  $G(V, E)$  and  $G'(V', E')$ , and (b)  $\mathcal{VO}_{\mathcal{G}_\delta, G(V, E)} \leftarrow \text{pghRedact}(\mathcal{G}_\delta, G(V, E))$  and  $\mathcal{VO}'_{\mathcal{G}'_\delta, G'(V', E')} \leftarrow \text{pghRedact}(\mathcal{G}'_\delta, G'(V', E'))$
4. The output of the experiment is 1 if and only if any of the following is true: in such a case, we say that  $\mathcal{A}$  has found a collision for  $\mathcal{H}^{s,x}$ ; else the output of the experiment is 0.
  - (a)  $G(V, E) \neq G'(V', E')$  and  $\text{pg}H = \text{pg}H'$ , where  $\text{pg}H \leftarrow \text{pg}\mathcal{H}^{s,x}(G)$ , and  $\text{pg}H' \leftarrow \text{pg}\mathcal{H}^{s,x}(G'(V', E'))$ .
  - (b)  $G(V, E) \neq G'(V', E')$  and  $\text{pghVerify}^{s,x}(\text{pg}H', \mathcal{G}_\delta, \mathcal{VO}) = \text{pg}\mathcal{H}^{s,x}(\text{pg}H, \mathcal{G}'_\delta, \mathcal{VO}')$ .

The following experiment involves the adversary who can learn information about the graphs applied with hash. In the first game, the adversary is challenged to determine the graph that has been hashed without the knowledge of the graphs themselves. The adversary is given two hash values computed either for the same graph or for two distinct graphs. If the hash function is not perfectly collision-resistant hash function then the adversary can determine whether the two hash values correspond to one graph or the two graphs with a probability non-negligibly greater than  $\frac{1}{2}$ .

**Privacy experiment-1:**  $\text{PGH-Priv1}_{\mathcal{A}, \text{pg}\Pi}(\lambda, \lambda_r)$

1. Key  $\mathbf{s} \leftarrow \text{pgGen}(1^\lambda)$
2. Randomizers  $\mathbf{r}_1, \mathbf{r}_2 \leftarrow \text{pgRandom}(1^{\lambda_r})$
3. Any two random graphs  $G_0(V_0, E_0)$ , and  $G_1(V_1, E_1)$  that differ only at the contents of one or more nodes, drawn uniformly from  $\mathcal{G}$ .
4. Toss an unbiased coin; if it returns head then bit  $b = 1$ , else  $b = 0$ .
5. Compute the following:  $\text{pg}H_0 \leftarrow \text{pg}\mathcal{H}(\mathbf{r}_0, G_0(V_0, E_0))$  and  $\text{pg}H_1 \leftarrow \text{pg}\mathcal{H}(\mathbf{r}_1, G_1(V_1, E_1))$ .
6.  $\mathcal{A}$  is given  $\mathbf{s}, \mathbf{r}, \text{pg}H_0$  and  $\text{pg}H_1$ ;  $\mathcal{A}$  outputs a bit  $b'$ .

7. The output of the experiment is 1 if and only if any of  $b = b'$ .

The following experiment is for privacy, but is with respect to the hash-redaction algorithm.

**Privacy experiment-2:**  $\text{PGH-Priv2}_{\mathcal{A}, \text{pg}\Pi}(\lambda, \lambda_r)$

1. Compute Key  $\mathbf{s} \leftarrow \text{pgGen}(1^\lambda)$
2. Compute randomizer  $\mathbf{r} \leftarrow \text{pgRandom}(1^{\lambda_r})$
3. Draw a random graph  $G(V, E)$ . Determine any two sets of subgraphs  $\mathcal{G}_{\delta_0}, \mathcal{G}_{\delta_1} \subseteq G(V, E)$ . Compute the hash of  $G(V, E)$ :  $\text{pg}H \leftarrow \text{pg}\mathcal{H}^{s,x}(G(V, E))$ .
4. Toss an unbiased coin; if it returns head then bit  $b = 1$ , else  $b = 0$ .
5. Compute the following:  $\mathcal{VO}_0 \leftarrow \text{pghRedact}(\mathcal{G}_{\delta_0}, G(V, E))$  and  $\mathcal{VO}_1 \leftarrow \text{pghRedact}(\mathcal{G}_{\delta_1}, G(V, E))$
6.  $\mathcal{A}$  is given  $\mathbf{s}, \mathbf{r}, \text{pg}H, \mathcal{VO}_0$  and  $\mathcal{VO}_1$ ;  $\mathcal{A}$  outputs a bit  $b'$ .
7. The output of the experiment is 1 if and only if any of  $b = b'$ .

*Definition 6.* A hash function  $\text{pg}\Pi = (\text{pgGen}, \text{ph}\mathcal{H}, \text{pghRedact}, \text{pghVerify})$  is collision-resistant if for all PPT adversaries  $\mathcal{A}$  there exists a negligible function  $\text{negl}$  such that

$$\Pr[\text{PGH-Coll}_{\mathcal{A}, \text{pg}\Pi}(\lambda, \lambda_r) = 1] \leq \text{negl}(\lambda, \lambda_r)$$

*Definition 7.* A hash function  $\text{pg}\Pi = (\text{pgGen}, \text{ph}\mathcal{H}, \text{pghRedact}, \text{pghVerify})$  is perfectly collision-resistant if for all PPT adversaries  $\mathcal{A}$  there exists a negligible function  $\text{negl}$  such that

$$\Pr[(\text{PGH-Priv1}_{\mathcal{A}, \text{pg}\Pi}(\lambda, \lambda_r) = 1)$$

$$\vee (\text{PGH-Priv2}_{\mathcal{A}, \text{pg}\Pi}(\lambda, \lambda_r) = 1)] \leq \frac{1}{2} + \text{negl}(\lambda, \lambda_r)$$

## 7. CONSTRUCTION OF HASHING SCHEME FOR GRAPHS

In this section, we propose a construction of collision-resistant hashing scheme for general graphs that is applicable to trees, DAGs and graphs with cycles. The scheme is secure with respect to  $\text{pg}\Pi$ . Our construction exploits a specific property of graph traversals, and defines a new type of trees “efficient-tree” that are used to represent graphs.

### 7.1 Graph Traversal

A graph  $G(V, E)$  can be traversed in a depth-first manner or breadth-first manner [4]. Post-order, pre-order, and in-order graph traversals are defined in [9, 4]. While post-order and pre-order traversals are defined for all types of trees, in-order traversal is defined only for binary trees. In each of these traversals, the first node visited is assigned 1 as its *visit count*. For every subsequent vertex visited, the *visit count* is incremented by 1 and is assigned to the vertex. This sequence of numbers is called the sequence of post-order (PON), pre-order (RON), or in-order (ION) numbers for the tree  $T$ , depending on the particular type of traversal. Figure 2 shows the traversal numbers and the DFT for a graph.

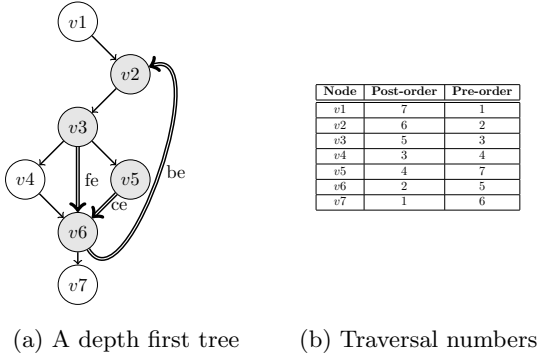


Figure 2: A graph with depth-first tree.

*Properties of traversal numbers:* The post-order number of a node is smaller than that of its parent. The pre-order number of a node is greater than that of its parent. The in-order number of a node in a binary tree is greater than that of its left child and smaller than that of its right child. A specific traversal number of a node is always smaller than that of its right sibling. The following lemma provides the basis for using traversal numbers in hash computation.

LEMMA 1. *The pair of post-order and pre-order number for a node in a non-binary tree correctly and uniquely determines the position of the node in the structure of the tree, where the position of a node is defined by its parent and its status as the left or right child of that parent.*

PROOF. From the post-order and pre-order traversal sequences of the vertices, it is possible to *uniquely* re-construct a non-binary tree [5] [7]. Thus from these sequences or from their counterparts, for a node, it is possible to correctly identify its parent and its status as left or right child of that parent in the tree. □

## 7.2 Graphs

In our scheme we are going to use the notion of post-order and pre-order numbers. However, in order to do that, we need to represent the graph as a tree. To that end, what we do is: we carry out a depth-first search (DFS) of the graph  $G(V, E)$ , which gives us one or more depth-first trees (DFT).

The various types of edges in a graph are defined below using the notion of traversal numbers. An example of depth-first tree and types of its edges is given in Figure 2 with the post- and pre-order numbers for each node being given in the table in the figure. Edge  $e(v_3, v_6)$  is a forward-edge, edge  $e(v_5, v_6)$  is a cross-edge and edge  $e(v_6, v_2)$  is a back-edge. However, DFTs do not capture the edges that are called forward-edges, cross-edges and back-edges.

*Definition 8.* Let  $\tau$  be the depth-first tree (DFT) of a directed graph  $G = (V, E)$ . Let  $x, y \in V$ , and  $e(x, y) \in E$ . Let  $p_x$  and  $r_x$  refer to post-order number and pre-order number of node  $x$ , respectively. With respect to the DFT  $\tau$ ,  $e(x, y)$  is a (1) tree-edge, iff  $p_x > p_y$ , and  $r_x < r_y$ ; (2) forward-edge, iff there is a path from  $x$  to  $y$  consisting of more than one tree-edges,  $p_x > p_y$ , and  $r_x < r_y$ ; (3) cross-edge, iff  $p_x > p_y$ , and  $r_x > r_y$ ; (4) back-edge, iff  $p_x < p_y$ , and  $r_x > r_y$ .

*Efficient-Tree Representation of a Graph:* In the efficient tree representation of a graph,  $G(V, E)$  we represent forward-, cross-, and back-edges by a special node called edge-node which contains information about both the source and target vertex pre- and post-processing numbers (for that specific edge). Moreover, edge-node has an incoming edge from the node the specific edge originates from. The edge-nodes do not have any outgoing edges. Once the post-order and pre-order numbers have been assigned to the nodes, we can dismantle the structure of the DFT, because the traversal numbers can be used to re-construct the DFT again (Lemma 1 and Definition 8). Figure 3 shows such tree-representations of graphs.

## 7.3 Construction of Collision-resistant Hashing Scheme for Graphs

The construction of the collision-resistant hash functions for graphs  $gH$  is given below.  $\mathcal{H}$  refers to a standard hash function as defined by II. The last statement in  $g\mathcal{H}$  computes the hash of a tree as shown in Figure 3 (b-efficient) and (c-efficient).

Using the proposed schemes different hash values are generated for isomorphic graphs [4] i.e., if the origin or DFS-tree changes the resulting hash value also changes.

$g\mathcal{H}$ : *Input:* a graph  $G(V, E)$  .

1. Sort the source nodes of the graphs in the non-decreasing order of their contents or label.
2. Let  $x$  be the first source node in the sorted order. If there are no source nodes in  $G(V, E)$  choose  $x$  randomly.
3. Carry out the depth-first traversal of the graph  $G(V, E)$  from source node  $x$  as follows.
  - (a) If node  $y$  is visited in DFS, assign its (post-order, pre-order) numbers to it:  $(p_y, r_y)$ .
  - (b) Let there be an edge  $e(y, z)$  such that  $e(y, z) \in E$  and  $z \in V$ . If  $e(y, z)$  is not a tree-edge, then create an edge-node  $yz$  having the following content:
    - If cross-edge:  $ce((p_y, r_y), (p_z, r_z))$ ;
    - If forward-edge:  $fe((p_y, r_y), (p_z, r_z))$ ;
    - If back-edge:  $be((p_y, r_y), (p_z, r_z))$ .
  - (c) Add an edge from  $y$  to the new node  $yz$ , and remove the edge  $e(y, z)$ .
4. Remove  $x$  from the sorted order of source nodes if exists.
5. If there are nodes in  $G(V, E)$  that are not yet visited, then repeat from 2.
6. Compute hash of each node  $y$  as follows:

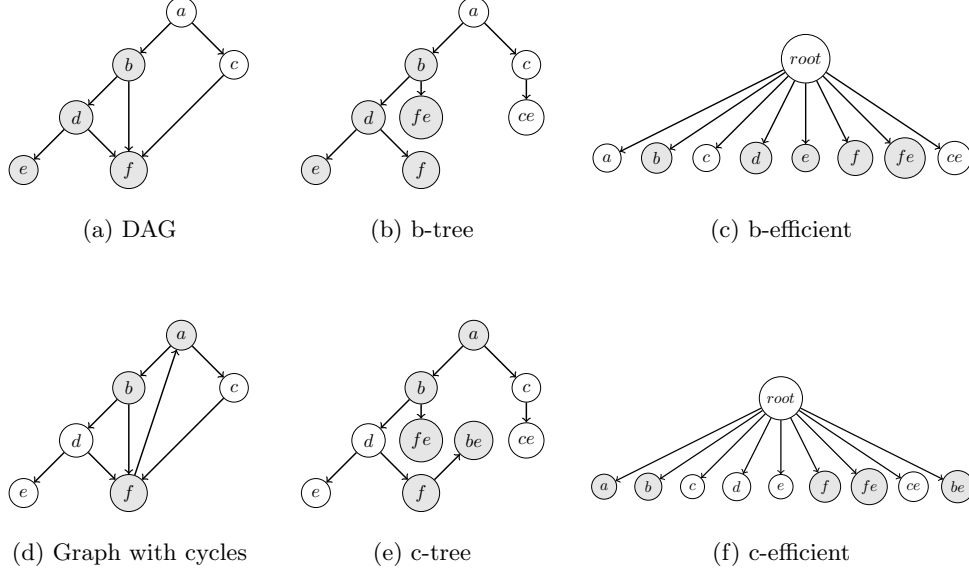
$$H_y \leftarrow \mathcal{H}((p_y, r_y) || y).$$

7. To each edge-node  $yz$ , assign  $H_{yz} \leftarrow \mathcal{H}(H_y || H_z)$ .
8. Compute the hash of graph  $gH_{G(V, E)}$  as follows:

$$gH_{G(V, E)} \leftarrow \mathcal{H}(H_{y_1} || H_{y_2} || \dots || H_{y_m})$$

where  $y_i$  refers to the  $i$ 'th node in the increasing order of the post-order numbers of the nodes, and of the





**Figure 3:** Tree representation of the running examples from Figure 1: (b-tree) and (c-tree) are tree-representations of the graphs in (a) and (d) – each node contains its (post-order, pre-order) number. (b, c-efficient) are one-level tree representations of the graphs exploiting the post-order and pre-order numbers of graph traversals. Each node contains their numbers. *fe*, *ce*, and *be* represent forward-edges, cross-edges and back-edges, respectively. Each of these nodes contain the post- and pre-order numbers of the origin and target of the edges and the hashes of their contents.

originating nodes of the edge-nodes, and  $m$  is the total number of nodes in the efficient-tree including original nodes and the edge-nodes.

**ghRedact:**

1. Input: a set of subgraphs  $\mathcal{G}_\delta$  that contains the efficient tree representations of the subgraphs, and the efficient tree-representation of graph  $G(V, E)$ .
2. Given  $\mathcal{G}_\delta$ , and  $G(V, E)$ , determine the set  $V \setminus \text{excluded}$  and  $E \setminus \text{excluded}$  consisting of excluded nodes and edge-nodes, respectively, in the efficient tree-representation of the graph.
3.  $\mathcal{VO} \leftarrow \emptyset$
4.  $\mathcal{VO} \leftarrow \mathcal{VO} \cup ((p_y, r_y), Hy)$ , where  $y \in V \setminus \text{excluded}$ .
5.  $\mathcal{VO} \leftarrow \mathcal{VO} \cup (\tau((p_y, r_y), (p_z, r_z), Hyz))$ , where  $\tau$  defines the type of the edge-node: *fe*, *ce*, and *be*, and  $yz \in E \setminus \text{excluded}$ .

**ghVrfy:**

1. Input: a set of subgraphs  $\mathcal{G}'_\delta$  that contains the efficient tree representation of the subgraphs, a set of verification objects  $\mathcal{VO}$ , and the hash of the complete graph  $gH$ .
2. Sort the received nodes, edge-nodes in the increasing order of the post-order numbers of the nodes or origins of the edge-nodes.

3. For each node  $x$  that is in a subgraph  $G_\delta \in \mathcal{G}'_\delta$ , compute  $H_x$ .
4. For each edge-node  $xy$  that is in a subgraph  $G_\delta \in \mathcal{G}'_\delta$ , compute  $H_{xy}$ .
5. Compute the hash  $gH' \leftarrow \mathcal{H}(H_{x1} || H_{x2} || \dots || H_{xm'})$ , where  $x_i$  refers to the  $i$ 'th node in the increasing order of the post-order numbers of the nodes, and of the originating nodes of the edge-nodes, and  $m'$  is the total number of nodes in the efficient-tree including original nodes and the edge-nodes.
6. Iff  $gH' = gH$ , return 1, else return 0.

## 7.4 Construction of Perfectly Collision-resistant Hashing Scheme for Graphs

In order to construct perfectly collision-resistant hash functions for graphs, we need to handle two cases of use of hashes: (1) redaction is not necessary, and (2) redaction is necessary. The reason is: redaction leads to use of non-empty verification objects that contain post-order and pre-order numbers, which would leak information (as shown in previous Section). When redaction is to be supported, the verification object needs to protect the information about the graph, and should not leak it to the verifier via the hash value. As in standard hashing for graphs described earlier, the verification object contains the post-order and pre-order number of each node that is not in the redacted graph and its hash. Post-order and pre-order numbers leak information about the number of nodes in the graph, and other information such as cycles or whether there are different types of edges

in the graph. In order to make the hash scheme perfectly collision-resistant one, then it has to prevent leakage of such information.

There are use-cases that do not need redaction, and there are other use-cases that need redaction. If redaction is not essential for hashing, then the computation of hash has to make sure that the hash value  $pgH_{G(V,E)}$  does not leak any information about the graph  $G(V, E)$ , which is defined in the Section 6.2; it should be secure against the Privacy Experiment - 1 in Section 6.2. If redaction is necessary, in order to prevent any leakage by the redacted hash value of a sub-graph  $G_\delta(V_\delta, E_\delta)$ , the verification object must also not leak any information; the hash function should be secure against both privacy experiments: Privacy Experiment - 1 and 2.

## 7.5 Perfectly Collision-resistant Hashing Without Redaction

In this section, we would discuss what changes we would make to the hashing scheme described in the previous section in order to make it perfectly-secure without supporting redaction. In what follows, we have referred to the steps of the previous section 7.

For hashing:

1. In Step 3(a), generate a unique random number  $\mu(x)$  for each node/edge-node  $x$ .
2. In Step 6 for computing hash of a node, use the random number: compute hash

$$H_y \leftarrow \mathcal{H}((p_y, r_y) || \mu(y) || y).$$

3. In Step 7 for computing hash of an edge-node  $yz$ , compute hash

$$H_{yz} \leftarrow \mathcal{H}(\mu(yz) || H_y || H_z).$$

4. Hash-value of the graph is:  $(H_{G(V,E)}, \sigma)$ , where  $\sigma$  is an ordered list of the random numbers computed during the process of hashing, and the order is the order in which the nodes are sorted for hashing.

For verification:

1. A verifier receiver the hash value that has  $H_{G(V,E)}$ , and  $\sigma$ ;
2. Verifier sorts the nodes as in Step 2, and computes the hash of each of the node by using the random number for that node in  $\sigma$ : it concatenates the random value while computing the hash.

Use of the random numbers help this updated scheme support the Privacy Experiment - 1.

## 7.6 Perfectly Collision-resistant Hashing With Redaction

In order to develop a hashing scheme for graphs that supports both Privacy Experiment - 1 and 2, we need to ensure that the post-order and pre-order numbers do not leak any information. In that case, we recommend use of order-preserving encryption to encrypt the post-order numbers and pre-order numbers, and use the encrypted values in place of the corresponding plaintext values [16] [1]. Use of such encrypted values have been supported in the notion of randomized traversal numbers [13].

Computation of hash of a graph and redacted graph would use encrypted randomized traversal numbers, and the random numbers, which support Privacy Experiments - 1 and 2, respectively. The rest of the scheme is as described in the previous section 7.5.

## 7.7 Security

The following lemma states the security of the proposed constructions for  $g\Pi$  and  $pg\Pi$ .

LEMMA 2. *Under the random-oracle model, the  $g\Pi$  is a collision-resistant hash function for graphs.*

LEMMA 3. *Under the random-oracle model, the  $pg\Pi$  is a perfectly collision-resistant hash function for graphs.*

Due to space constraints we are unable to provide detailed proofs, which will be added in a technical report as an extension of this paper. However, the proof of Lemma 2 follows from the properties of post-order and pre-order numbers and the random oracle hypothesis. Post-order and pre-order numbers prevent collision for graphs that have same set of nodes and edges but having the possibility that the order between siblings may have been changed, which would be invalidated by these numbers. The hash function being used to hash nodes and edge-nodes implement the random oracle and thus they are collision resistant. Any collision found by  $g\Pi$  implies that our claims about the post-order and pre-order numbers and the hash functions are invalid.

Similarly, for Lemma 3, security depends on the security of (1) traversal numbers, (3) order-preserving encryption and (2) random oracle premise. In fact collision resistance is dependent on 1 and 3, while privacy properties of the hash values depend on the security properties of the encryption function. We would specify the proofs in a technical report.

## 8. PERFORMANCE RESULTS

In the following sections, we analyze the complexity and performance of the proposed schemes.

*Cost:* Each of the schemes  $g\mathcal{H}$ ,  $ghRedact$ , and  $ghVrfy$ , performs a traversal on the graph in the worst case (when the graph is shared as it is with no redaction). The cost of such traversal and each of these methods is:  $O(|V| + |E|)$ .

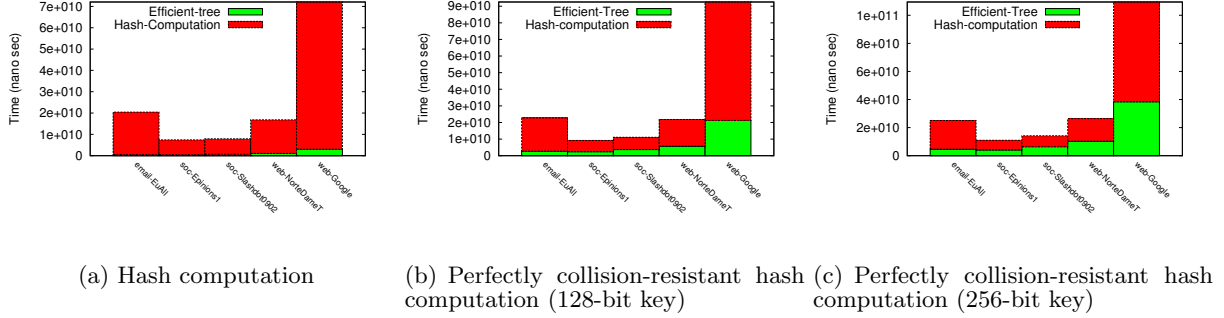
We have implemented the two schemes in Java 1.6 using JGraphT 0.8.3 (a free Java Graph Library) APIs. The experiments were carried out on a machine with following specification: Dell PowerEdge 2950, 64-bit Linux (Gentoo 2.6.32.20) running on dual 3.0GHz Intel Xeon E5450 processors with 32GB of RAM. We provided JVM with the following parameters: `-Xss1024m`, `-Xms2048m`, and `-Xmx4096m` to set the thread stack size and min/mix values for buffer space. For our experiments, we generated synthetic graph data sets using JGraphT's `RandomGraphGenerator` class and used real world directed graph data sets available at Stanford Large Network Dataset Collection<sup>5</sup>. Table 1 shows the specification of real world data sets as we used in our simulations. We ran all our simulations 10 times - per input instance - and report the average computation time for the results.

In our experiments, we have implemented the hashing scheme for graphs as in Section 7, and the perfectly collision-resistant hashing scheme without redaction as in Section 7.5.

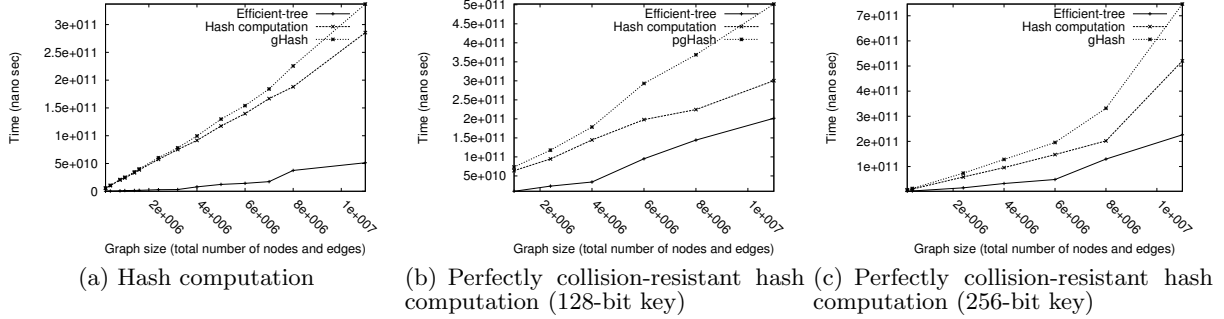
<sup>5</sup><http://snap.stanford.edu/data/>

**Table 1: Different graph data sets**

Graph	$ V $	$ E $	Description
soc-Epinions1	75,879	508,837	Who-trusts-whom network of Epinions.com
soc-Slashdot0902	82,168	870,161	Slashdot social network from February 2009
email-Enron	265,214	418,956	Email communication network from Enron
web-NorteDame	325,729	1,469,679	Web graph of Notre Dame
web-Google	875,713	5,105,039	Web graph from Google



**Figure 4: Time taken to compute hash of the graph using  $g\mathcal{H}$  scheme; Efficient-tree represents the time to build the efficient-tree; and the Hash computation represents the time to compute the  $gH_{G(V,E)}$  value using efficient-tree.**



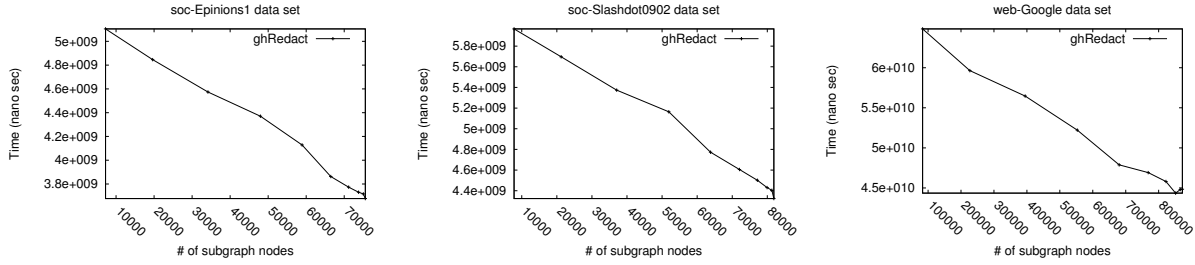
**Figure 5: Time taken to compute the hash of graph end-to-end; Efficient-tree represents the time to build the efficient-tree; Hash computation represents the time to compute the  $gH_{G(V,E)}$  value using efficient-tree; and gHash represents the total time to compute the graph hash i.e., efficient-tree and hash computation combined.**

Our simulation results report three different kind of results:

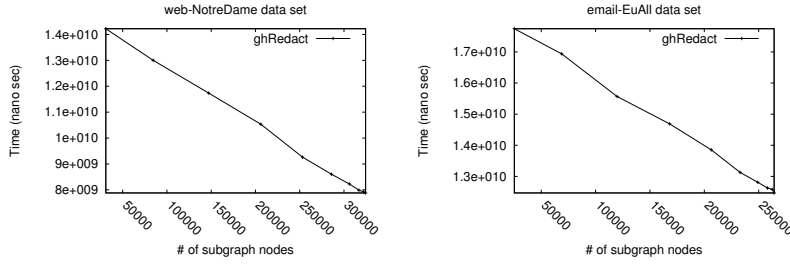
- For  $g\mathcal{H}$  function we report (a) the time to construct Efficient-tree from input graph and (b) the graph hash value computation,  $gH_{G(V,E)}$ , time for the two proposed hashing schemes.
- For  $ghRedact$  we report the time to compute the redacted graph vs number of nodes requested for  $ghRedact$ . Besides computing the redacted graph this process also involves generating the set of verification objects,  $\mathcal{VO}$  for both original nodes and edge-nodes in the redacted graph.
- In the final set of experimental results we plot  $ghVrfy$  computation time vs number of nodes requested by the user.

*Time taken to build Efficient-tree:* This process involves running the Depth First Search (DFS) traversal algorithm on the input directed graph to assign both the pre- and post-processing numbers to nodes (i.e., determining the structural positions of nodes in a directed graph) and based-on these numbers classifying the graph edges as tree, forward, cross, and back edges. Additionally, this process also involves making the edge-node set for forward-, cross-, and back-edges. This is part of the one-time process of computing graph hash value,  $gH_{G(V,E)}$ .

*Time taken to compute  $gH_{G(V,E)}$  value:* This process requires computing the hash value of all the nodes in the Efficient-tree representation of  $G(V,E)$  by traversing the nodes in post-order fashion. Our implementation uses `TreeMap()` to keep a sorted listing of all the nodes (original nodes and edge-nodes) by post-order number. We use Java's `String-`

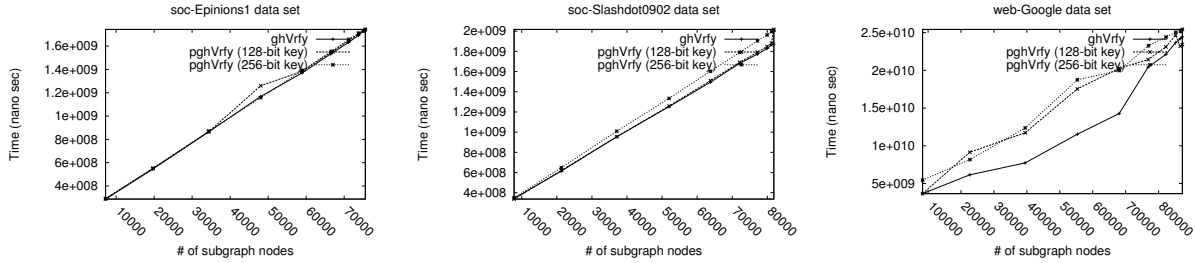


(a) Reduction time for soc-Epinions1 data set (b) Reduction time for soc-Slashdot0902 data set (c) Reduction time for web-Google data set

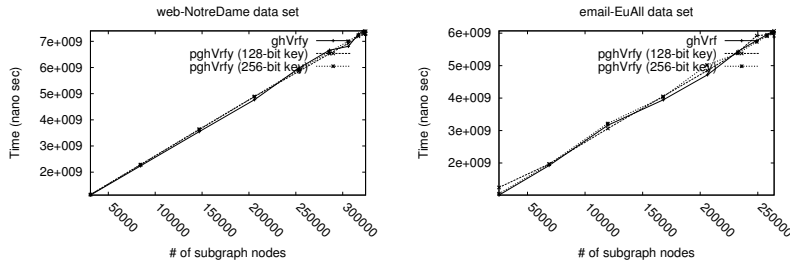


(d) Reduction time for web-NotreDame data set (e) Reduction time for email-EuAll data set

**Figure 6: ghRedact/pghRedact function computation time vs number of requested subgraph nodes**



(a) Verification time for soc-Epinions1 data set (b) Verification time for soc-Slashdot0902 data set (c) Verification time for web-Google data set



(d) Verification time for web-NotreDame data set (e) Verification time for email-EuAll data set

**Figure 7: ghVrfy/pghVrfy function computation time vs number of requested subgraph nodes**

Buffer class to maintain a running concatenation of hash values for all the nodes in the Efficient-tree.

From Figure 4 (a) we can see that the time to construct Efficient-tree is significantly less than the time to compute  $gH_{G(V,E)}$  value whereas it is less than the time to compute

$pgH_{G(V,E)}$  value ( Figure 4 (b) and (c)); We can also infer that the time to compute  $pgH_{G(V,E)}$  value with 128/256-bit key is more than the time to compute  $gH_{G(V,E)}$  value, as this process now involves generating a random key with 128/256-bit length. The results depict that the overall time

to compute  $g\mathcal{H}$  is between 70-100 seconds for web-Google data set with approximately six million nodes and edges combined signifying that our scheme is highly efficient. In fact we can make this more-efficient by by-passing  $O(n \log n)$  the sorting phase by placing the nodes in sorted-order during the DFS traversal once a node is finally visited. Figure 5 shows the end-to-end computation time for calculating the hash of graph for synthetic data.

*Time taken to compute  $ghRedact$ :* This operation is performed either by the data owner itself or by a third party data distributor on behalf of the user. From Figure 6 (a), (b), (c), (d), and (e) we can see that for all the graphs the  $ghRedact$  computation time decreases as we increase the number of nodes requested from the original graph. This is understandable that as we increase the size of the number of nodes requested the redacted graph size decreases and hence the computation required to generate the  $\mathcal{VO}$  set.

*Time taken to compute  $ghVrfy$ :* This function takes as input three parameters the original graph hash value  $gH_{G(V,E)}$ , the requested subgraph and the  $\mathcal{VO}$  set. Figure 7 (a), (b), (c), (d) and (e) shows the graph verification time vs the number of subgraph nodes returned. This function also incorporates computing the  $H_x$  and  $H_{xy}$  values for each node  $x$  and for each edge-node  $xy$ . We can see that as we increase the subgraph size the time to compute  $ghVrfy$  also increases; this is owing to the fact that this function has to compute the hash values for node and edge-nodes in the returned subgraph. Moreover, the time taken is more for the case of perfect graph hashing scheme when compared to the scheme  $g\mathcal{H}$ .

## 9. CONCLUSION AND FUTURE WORK

Graphs are widely used to specify and represent data. Verifications of integrity and whether two graphs are identical or not are often required by applications that involve graphs such as graph databases. Collision resistant one-way hashing schemes are the basic building blocks of almost all crypto-systems. In this paper, we studied the problem of hashing graphs with respect to crypto-systems and proposed two constructions for two notions of collision-resistant hash functions for graphs. We defined the formal security models of hashing schemes for graphs, and perfectly collision-resistant hashing schemes for graphs. We proposed the first constructions for general graphs that includes not only trees and graphs but also graphs with cycles and forests. Our constructions use graph traversal techniques and are highly efficient for hashing, redaction, and verification of graphs. Moreover, our proposed schemes require only a single traversal on the graph. To the best of our knowledge this is the first perfectly collision-resistant hashing schemes for graphs that are practical and are highly efficient for hashing, redaction, and verification of hashes graphs.

As a future work, we plan to take updates into consideration, parts of graph maybe updated and other parts may not be updated; the challenge is how to recompute hash of the updated graph with  $O(1)$  cost, which is currently logarithmic to linear (Merkle hash technique for trees requires  $O(n)$  – linear cost).

## 10. ACKNOWLEDGMENTS

This work is partially supported by NSF Grants IIS-0964639 and TUES-1123108.

## 11. REFERENCES

- [1] A. Boldyreva, N. Chenette, Y. Lee, and A. O’Neill. Order-preserving symmetric encryption. In *Advances in Cryptology-EUROCRYPT 2009*, pages 224–241. Springer, 2009.
- [2] R. Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *CRYPTO*, pages 455–469, 1997.
- [3] R. Canetti, D. Micciancio, and O. Reingold. Perfectly one-way probabilistic hash functions (preliminary version). In *STOC*, pages 131–140, 1998.
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
- [5] S. K. Das, K. B. Min, and R. H. Halverson. Efficient parallel algorithms for tree-related problems using the parentheses matching strategy. In *IPPS*, pages 362–367, Washington, DC, USA, 1994.
- [6] H. V. Jagadish and F. Olken. Database management for life sciences research. *SIGMOD Rec.*, 33(2):15–20, June 2004.
- [7] V. Kamakoti and C. P. Rangan. An optimal algorithm for reconstructing a binary tree. *Inf. Process. Lett.*, 42(2):113–115, 1992.
- [8] J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.
- [9] D. E. Knuth. *The Art of Computer Programming*, volume 1. Pearson Education Asia, third edition, 2002.
- [10] A. Kundu, M. J. Atallah, and E. Bertino. Leakage-free redactable signatures. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY ’12*, pages 307–316, New York, NY, USA, 2012. ACM.
- [11] A. Kundu and E. Bertino. Structural signatures for tree data structures. *PVLDB*, 1(1):138–150, 2008.
- [12] A. Kundu and E. Bertino. On hashing graphs. *IACR Cryptology ePrint Archive*, 2012:352, 2012.
- [13] A. Kundu and E. Bertino. Privacy-preserving authentication of trees and graphs. *International Journal of Information Security*, pages 1–28, 2013.
- [14] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39:21–41.
- [15] R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.
- [16] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. *IACR Cryptology ePrint Archive*, 2013:129, 2013.
- [17] M. L. Yiu, E. Lo, and D. Yung. Authentication of moving knn queries. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE ’11*, pages 565–576, Washington, DC, USA, 2011. IEEE Computer Society.
- [18] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow.*, 2(1):718–729, Aug. 2009.