On Hashing Graphs

Ashish Kundu[1] and Elisa Bertino[2]

[1] IBM T J Watson Research Center, New York, USA
[2] Department of Computer Science and CERIAS, Purdue University, West Lafayette, USA

Abstract. Collision resistant one-way hashing schemes are the basic building blocks of almost all crypto-systems. Use of graph-structured data models are on the rise – in graph databases, representation of biological and healthcare data as well as in modeling systems for representing system topologies. Therefore, the problem of hashing graphs with respect to crypto-systems needs to be studied and addressed. The traditional Merkle Hash technique cannot be applied as it is because graphs are more complex data structures than trees. 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 leakage-free hashing schemes for graphs, (3) we describe a hashing scheme for hashing directed and undirected graphs that uses Merkle hash technique, (4) and a hashing scheme that uses structural information instead of Merkle hash technique, (5) we define leakage-free hashing schemes for graphs. Our constructions use graph traversal techniques and are highly efficient with respect to updates to graphs: they require as little as two ($O(1)$) hash values to be updated to refresh the hash of the graph, while the Merkle Hash Technique and Search DAG schemes for trees and DAGs respectively require as many as $O(|V|)$ and $O(|V| + |E|)$.

1 Introduction

One of the fundamental building blocks of modern cryptography is hash functions. 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 [7, 15]. Often it is required to check if a given graph has been updated or not or whether two graphs are identical[3]. Hashing is used to evaluate these operations. Authenticating the graph-structured data is often required as a security property in databases [14, 13]. Authentication schemes use hashing as a basic building block.

Hashing graphs pose several challenges: graphs have several nodes and edges, with each node having multiple incoming edges and outgoing edges. The edges maybe directional and there maybe cycles in the graphs. There are four types of graphs that are of interest in this paper: (a) trees, (b) directed acyclic graphs, (c) graphs with cycles and

[3] Identical graphs are isomorphic graphs, but the reverse is not true [4].

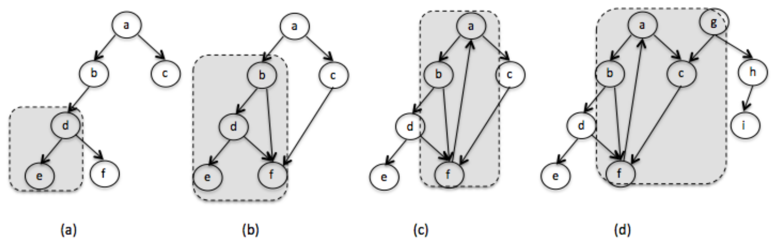


Fig. 1. Graphs: (a) Tree, (b) DAG, (c) Graph with cycles, (d) Graph with multiple sources (vertices with no incoming edge) and cycles. The shadowed with dotted boundary are the subgraphs that a user may receive.

(d) graphs with multiple sources. In the Figure 1, these types of graphs are seen. From (a) onwards, the complexity of hashing increases because the complexity of the graph increases from (a) to (b) and so on.

Implementing a random oracle as a hash function for graphs require that the same hash value is computed universally by any instance of that implementation of the random oracle. In case of hashing schemes such as SHA-1, SHA-2, a message is either shared completely or not shared at all with a user. In contrast, when graphs are used, a user may receive part(s) of a graph (subgraph(s)). Moreover, when taking updates into consideration, parts of the graph maybe updated and other parts may not be updated; the challenge is how does the hash of a graph is computed for an updated graph and what is its cost? Can we compute the updated hash value incrementally? Moreover, graph models are used for representation of sensitive data as well (such as healthcare, biological data). Traditional hashing schemes leak information (SHA1, SHA2 leak information about length of the plaintext [1]). The challenge is how to hash graphs so that no information is leaked at least in the context of probabilistic polynomial adversaries [9]. The Merkle hash technique (MHT) is used to compute hashes for trees [12] and has been extended for directed acyclic graphs [11]. However, not all the above challenges are addressed by the Merkle hash technique and its extensions. Cryptographic hashing techniques for cyclic graphs have not been well-studied. 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, the cost of computing the hashes on updates using MHT is quite high: any modification of the value of a node or insertion or deletion of a node leads to as many updates as the depth of the node is in a tree. Can we reduce some of these costs not only for trees but also for general graphs? 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. Encryption is too heavyweight and has different security properties than the properties required for hash functions. Therefore, there is a need for perfectly collision-resistant hash functions for graphs. In addition the formalization of such secure hashing schemes for graphs is an essential requirement besides their constructions.

In this paper, we make the following contributions: (1) we define the formal security model of collision-resistanthashing schemes for graphs, (2)we define the formal security model of perfectly collision-resistant hashing schemes for graphs, (3) we describe a hashing scheme for hashing directed and undirected graphs that uses the MHT, (4) and a hashing scheme that uses structural information instead of the MHT, (5) we propose a perfectly collision-resistant hashing scheme for graphs.

2 Related Works and Background

There are two techniques in the literature that come closest to the constructions presented in this paper: Merkle hash technique [12] (MHT) and the Search-DAG technique [11] (SDAG). Integrity assurance of tree-structured data is primarily carried by the Merkle hash technique [12]. This scheme requires information about other nodes and edges (extraneous information) in order to verify the integrity of a subtree, which is why it leaks. We would describe the issues associated with MHT in the next section, and the following section, we would describe the SDAG technique. Martel et al. [11] proposed SDAG – an authentication technique for directed acyclic graphs referred to as "Search DAGs" in third party distribution frameworks. Their technique uses Merkle hash technique. The SDAG technique only covers DAGs, not the general directed graphs, which is of greater interest in this paper.

2.1 Hashing Trees with Merkle Hash Technique

The Merkle hash technique [12] 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 $mh(x)$ = $\mathcal{H}(mh(y_1)\|mh(y_2)\|\ldots\|mh(y_m))$, where y_1, y_2, $\ldots$,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 Merkle hash 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 $mh(d)$ = $\mathcal{H}(mh(e) \parallel mh(f))$. 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 $\mathcal{H}(mh(b) \parallel mh(c))$, respectively.

In order to account for the contents in non-leaf nodes, two simple variants of the Merkle hash technique can be used to compute the MH of a non-leaf node from the MH of its children and the contents (or hash of the content) of the non-leaf node itself. Suppose x to be a non-leaf (thus a root/intermediate) node in T. The MH of x is defined as follows: $mh(x)$ = $\mathcal{H}(\mathcal{H}(c_x)\|mh(y_1)\|mh(y_2)\|\ldots\|mh(y_m))$.

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 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_δ (Consider the shadowed subtree in Figure 1:

1. With respect to each node in T_δ, MH of its siblings[4] that are in T but not in T_δ. For example, in Figure 1, $mh(f)$ (with respect to e) is sent.
2. With respect to each node x in T_δ, the MH of each sibling of each ancestor of x, if that sibling is not in T_δ. In our example, $mh(a)$ and $mh(b)$ are also sent.
3. With respect to each node in T_δ, the hash of the content of each of its ancestor. In our example, $\mathcal{H}(c_a)$ and $\mathcal{H}(c_b)$ are sent to the user.
4. The structural order between a node in T_δ and its sibling(s) that are not in T_δ, and the structural order between the sibling nodes that are not in T_δ. In our example, the order between e and f, and the order between b and c are sent to the user.
5. Parent-child/ancestor-descendant relationship(s) between a node in T_δ and another node not in T_δ (such as the relationship between b and d), and those between the nodes that are not in T_δ (such as 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_δ). In our example, a is the root of the tree and this fact is conveyed to the user.

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.

2.1.2 Privacy Attacks The attacks on the Merkle hash technique are based on the set of *verification objects* sent to the user. By exploiting the knowledge of these information, inference attacks described below can be carried out on the MHT.

– The first attack exploits the information (1), (2), and(3). By comparing the Merkle hash of a node e in the shared subtree with the MH of another node f received, the user can infer whether contents of e is same as that of f and if the subtree with root e is identical to the subtree with root f.
– The second attack exploits the information (1), (2), (3), and (6). By comparing the Merkle hashes of two nodes (c and f) that are received, the user can infer whether the contents of c is same as that of f.

2.1.3 Inefficiency of Updates using MHT Consider that there is a change to the node e. The change affects the hash values of e, d, b, and a. A single change led to four changes. In general in MHT, a change in a node may lead to changes in $O(n)$ nodes, with n nodes in the tree. This leads to several performance issues especially in memory integrity, integrity in databases and indices.

2.2 Hashing Graphs

Graphs are more complex structures than trees. Nodes may have multiple incoming edges, there maybe no roots or source nodes; perhaps, there maybe a number roots or source nodes. The major challenges in hashing graphs arise out of the fact that (i) a

[4] Nodes that are siblings in a tree have a common parent.

graph may have cycles, (ii) changes to the nodes affects 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 the figure, 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 SDAG technique, then hash of node f influences 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 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. How can we 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 maybe identical.

2.3 Formal Security Models

Existing works [12, 11] do not formally define the notion of graph hashing, nor they define nor construct perfectly collision-resistant hash functions. Canetti et al. [1, 2] 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. [2] also proposed that *verify* method should be included in the definition of random oracles. In this paper, we have also proposed 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 an 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 [9]). A standard hashing scheme Π is a tuple $(\texttt{Gen}, \mathcal{H})$. ($\texttt{s}$ is not a secret key in standard cryptographic sense.)

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

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

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

If $\mathcal{H}^s$ is defined only for inputs $x \in \{0,1\}^{l'(\lambda)}$ and $l'(\lambda) > l(\lambda)$, then we say that $(\texttt{Gen},\mathcal{H})$ 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 [9]. 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 [9]). In the rest of the paper, $\mathcal{A}$ is a probabilistic polynomial time (PPT) adversary.

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

1. Key $(\texttt{pk, sk}) \leftarrow \texttt{Gen}(1^\lambda)$
2. $\mathcal{A}$ is given $\texttt{s}$ and outputs x and x'. (If Π is a fixed-length hash function then 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 $\mathcal{H}^{\texttt{s}}(x) = \mathcal{H}^{\texttt{s}}(x')$. In such a case, we say that $\mathcal{A}$ has found a collision for $\mathcal{H}^{\texttt{s}}$; else the output of the experiment is 0.

Definition 2. *A hash function $\Pi = (\texttt{Gen}, \mathcal{H})$ is collision-resistant if for all probabilistic polynomial adversaries $\mathcal{A}$ there exists a negligible function $\texttt{negl}$ such that*

$$\texttt{Pr}(\texttt{H} - \texttt{Coll}_{\mathcal{A},\Pi}(\lambda) = 1) \leq \texttt{negl}(\lambda)$$

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 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 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 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 Merkle hash technique, 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 probabilistic polynomial-time algorithms and one deterministic algorithm $g\Pi$ = (`gGen`, $g\mathcal{H}$, `ghRedact`, `ghVrfy`) satisfying the following requirements:*

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

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

HASH-REDACTION: *The redaction algorithm `ghRedact` is a probablistic 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 $\mathcal{G}_\delta$.*

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

HASH-VERIFY: `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:* $b \leftarrow \mathtt{ghVrfy}^{\mathtt{s}}(gH, \mathcal{G}, \mathcal{VO})$

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 λ, any key $\mathtt{s} \leftarrow \mathtt{gGen}(\lambda)$, and any $gH \leftarrow g\mathcal{H}^{\mathsf{s}}(G(V, E))$, $\mathcal{VO} \leftarrow \mathtt{ghRedact}(\{G\}, G)$, $\mathtt{ghVrfy}^{\mathsf{s}}(gH, \{G(V, E)\}, \mathcal{VO})$ always outputs 1.

Hash-Redaction Correctness: For any graph $G(V, E)$, any positive integer value of λ, any key $\mathtt{s} \leftarrow \mathtt{gGen}(\lambda)$, 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 $gH \leftarrow g\mathcal{H}^{\mathsf{s}}(G)$, $\mathcal{VO} \leftarrow \mathtt{ghRedact}(\mathcal{G}_\delta, G)$, $\mathtt{ghVrfy}^{\mathsf{s}}(gH, \mathcal{G}_\delta, \mathcal{VO})$ 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: $\mathtt{GH\text{-}Coll}_{\mathcal{A},g\Pi}(\lambda)$

1. Key $\mathtt{s} \leftarrow \mathtt{gGen}(1^\lambda)$
2. $\mathcal{A}$ is given $\mathtt{s}$ and outputs (a) $G(V, E)$ and $G'(V', E')$, and (b) $\mathcal{VO}_{\mathcal{G}_\delta, G(V,E)} \leftarrow \mathtt{ghRedact}(\mathcal{G}_\delta, G(V, E))$ and $\mathcal{VO}'_{\mathcal{G}'_\delta, G'(V',E')} \leftarrow \mathtt{ghRedact}(\mathcal{G}'_\delta, G'(V', E'))$
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 $\mathcal{H}^{\mathsf{s}}$; else the output of the experiment is 0.
   (a) $G(V, E) \neq G'(V', E')$ and $gH = gH'$, where $gH \leftarrow g\mathcal{H}^{\mathsf{s}}(G)$, and $gH' \leftarrow g\mathcal{H}^{\mathsf{s}}(G'(V', E'))$.
   (b) $G(V, E) \neq G'(V', E')$ and $\mathtt{ghVrfy}^{\mathsf{s}}(gH', \mathcal{G}_\delta, \mathcal{VO}) = g\mathcal{H}^{\mathsf{s}}(gH, \mathcal{G}'_\delta, \mathcal{VO}')$.

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

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

6 Perfectly Collision-resistant Hashing of 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 [1] 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 [2], 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 [2].

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 Merkle hash technique, 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, 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 a 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 $g\Pi$ consists of three probabilistic polynomial-time algorithms and one deterministic algorithm $g\Pi$ = (*`gGen`*, $g\mathcal{H}$,* `ghRedact`*,* `ghVrfy`*) satisfying the following requirements:*

KEY GENERATION: *The probabilistic key generation algorithm* `pgGen` *takes as input security parameter 1^λ and outputs a key* `s`*.*

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

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

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

HASHING: *There exists a polynomial l such that the hash algorithm $pg\mathcal{H}$ takes a key* `s` *and a graph $G(V,E)$ and outputs a string $pgH \leftarrow pg\mathcal{H}^{\mathtt{s},\mathtt{r}}(G(V,E)) \in \{0,1\}^{l(\lambda,\lambda_r)}$.*

HASH-REDACTION: *The redaction algorithm* `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 $\mathcal{G}_\delta$.*

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

HASH-VERIFY: `pghVerify` *is a deterministic algorithm that takes a hash value $pg\mathcal{H}$, 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.*

$$b \leftarrow \texttt{pghVerify}^{\mathrm{s},\mathrm{r}}(pgH, \mathcal{G}, \mathcal{VO})$$

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: $\texttt{PGH-Coll}_{\mathcal{A},pg\Pi}(\lambda, \lambda_r)$

1. Key $\mathrm{s} \leftarrow \texttt{pgGen}(1^\lambda)$
2. Randomizer $\mathrm{r} \leftarrow \texttt{pgRandom}(1^{\lambda_r})$
3. $\mathcal{A}$ is given s, r; $\mathcal{A}$ outputs (a) $G(V,E)$ and $G'(V',E')$, and (b) $\mathcal{VO}_{\mathcal{G}_\delta, G(V,E)} \leftarrow \texttt{pghRedact}(\mathcal{G}_\delta, G(V,E))$ and $\mathcal{VO}'_{\mathcal{G}'_\delta, G'(V',E')} \leftarrow \texttt{pghRedact}(\mathcal{G}'_\delta, G'(V',E'))$
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 $\mathcal{H}^{\mathrm{s}}$; else the output of the experiment is 0.
   (a) $G(V,E) \neq G'(V',E')$ and $pgH = pgH'$, where $pgH \leftarrow pg\mathcal{H}^{\mathrm{s},\mathrm{r}}(G)$, and $pgH' \leftarrow pg\mathcal{H}^{\mathrm{s},\mathrm{r}}(G'(V',E'))$.
   (b) $G(V,E) \neq G'(V',E')$ and $\texttt{pghVerify}^{\mathrm{s},\mathrm{r}}(pgH', \mathcal{G}_\delta, \mathcal{VO}) = pg\mathcal{H}^{\mathrm{s},\mathrm{r}}(pgH, \mathcal{G}'_\delta, \mathcal{VO}')$.

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. Iff 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: $\texttt{PGH-Priv1}_{\mathcal{A},pg\Pi}(\lambda,\lambda_r)$

1. Key $\texttt{s} \leftarrow \texttt{pgGen}(1^{\lambda})$
2. Randomizers $\texttt{r}_1$, $\texttt{r}_2 \leftarrow \texttt{pgRandom}(1^{\lambda_r})$
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: $pgH_0 \leftarrow \text{pg}\mathcal{H}(\texttt{r}_0, G_0(V_0, E_0))$. $pgH_1 \leftarrow \text{pg}\mathcal{H}(\texttt{r}_b, G_b(V_b, E_b))$.
6. $\mathcal{A}$ is given $\texttt{s}$, pgH_0 and pgH_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: $\texttt{PGH-Priv2}_{\mathcal{A},pg\Pi}(\lambda,\lambda_r)$

1. Compute Key $\texttt{s} \leftarrow \texttt{pgGen}(1^{\lambda})$
2. Compute randomizer $\texttt{r} \leftarrow \texttt{pgRandom}(1^{\lambda_r})$
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 $G(V, E)$: $pgH \leftarrow \text{pg}\mathcal{H}^{\text{s,r}}(G(V, E))$.
4. Toss an unbiased coin; if it returns head then bit $b = 1$, else $b = 0$.
5. Compute the following: $\mathcal{VO}_0 \leftarrow \texttt{pghRedact}(\mathcal{G}_{\delta 0}, G(V, E))$ and $\mathcal{VO}_1 \leftarrow \texttt{pghRedact}(\mathcal{G}_{\delta 1}, G(V, E))$
6. $\mathcal{A}$ is given $\texttt{s}$, pgH, $\mathcal{VO}_0$ 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 $pg\Pi$ = (*pgGen*, $pg\mathcal{H}$,* pghRedact*,* pghVerify*) is collision-resistant if for all probabilistic polynomial adversaries $\mathcal{A}$ there exists a negligible function* $\texttt{negl}$ *such that*

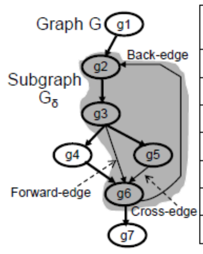
$$\texttt{Pr}((\texttt{PGH}-\texttt{Coll}_{\mathcal{A},pg\Pi}(\lambda,\lambda_r) = 1)) \leq \texttt{negl}(\lambda,\lambda_{\texttt{r}})$$

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

$$\texttt{Pr}((\texttt{PGH}-\texttt{Priv1}_{\mathcal{A},pg\Pi}(\lambda,\lambda_r) = 1) \vee (\texttt{PGH}-\texttt{Priv2}_{\mathcal{A},pg\Pi}(\lambda,\lambda_r) = 1)) \leq \frac{1}{2} + \texttt{negl}(\lambda,\lambda_{\texttt{r}})$$

7 Construction of Collision-resistant 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 $g\Pi$ and highly efficient updating hashes for updates to the nodes. Our construction exploits a specific property of graph traversals, and defines a new type of trees “efficient-tree” that are used to represent graphs.



Node	Post-order	Pre-order
g1	7	1
g2	6	2
g3	5	3
g4	3	4
g5	4	7
g6	2	5
g7	1	6

Fig. 2. A graph with depth-first tree in bold.

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 [10, 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.

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] [8]. 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).

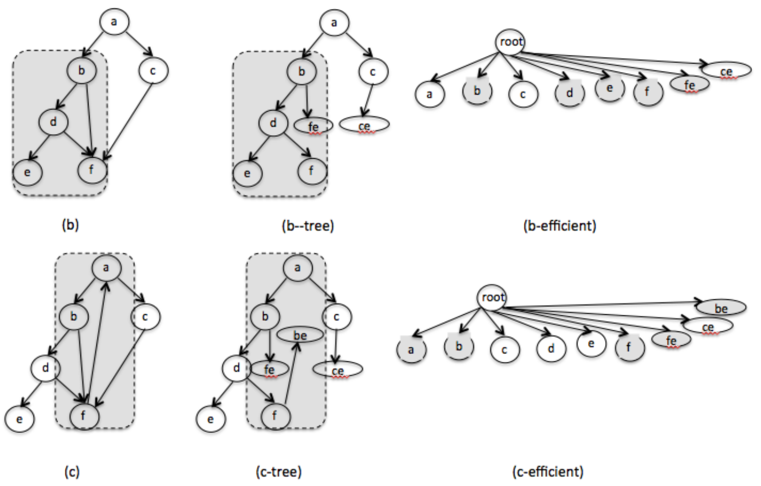


Fig. 3. Tree representation of the running examples from Figure 1: (b-tree) and (c-tree) are tree-representations of the graphs in (b) and (c) – each node contains its (post-order, pre-order) number. (b,c-efficient) are one-level tree representations of the graphs exploiting the post-order and pre-order 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.

The various types of edges in a graph are defined below using the notion of traversal numbers. An example of depth-first tree, types of edges are given in Figure 2 with the post- and pre-order numbers for each node being given in the table in the figure. Edge $e(g_3, g_6)$ is a forward-edge, edge $e(g_5, g_6)$ is a cross-edge and edge $e(g_6, g_2)$ is a back-edge. However, DFTs do not capture the edges that are called forward-edges, cross-edges and back-edges.

Definition 8. *Let τ 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 o_x and q_x refer to post-order number and pre-order number of node x, respectively. With respect to the DFT τ, $e(x, y)$ is a (1) tree-edge, iff $o_x > o_y$, and $q_x < q_y$; (2) forward-edge, iff there is a path from x to y consisting of more than one tree-edges, $o_x > o_y$, and $q_x < q_y$; (3) cross-edge, iff $o_x > o_y$, and $q_x > q_y$; (4) back-edge, iff $o_x < o_y$, and $q_x > q_y$.*

Efficient-Tree Representation of a Graph: We represent each such edge by a special node called edge-node and the special node has an incoming edge from the node the specific edge originates from. The special nodes do not have any outgoing edges. Once post-order and pre-order numbers are 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). The rest of the edges can be re-constructed from the edge-nodes. Figure 3 shows such tree-representations of graphs.

7.3 Construction of Collision-resistant Hash Scheme for Graphs

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

g$\mathcal{H}$:

1. Input: a graph $G(V, E)$and its efficient tree-representation.
2. Sort the source nodes of the graphs in the non-decreasing order of their contents or label.
3. Carry out depth-first traversal of the graph $G(V, E)$ on the first source node x in the sorted order. If there are no source nodes in $G(V, E)$, choose x randomly.
4. If node y is visited in DFS, assign its (post-order, pre-order) number to it.
5. If the edge $e(y, z)$ is not a tree-edge and is cross-edge, then create an edge-node yz having the following content $ce((p_y, r_y), (p_z, r_z))$; if forward-edge: $fe((p_y, r_y), (p_z, r_z))$, and back-edge: $be((p_y, r_y), (p_z, r_z))$
6. Add an edge from y to the new node yz, and remove the edge $e(y, z)$.
7. Remove x from the sorted order of source nodes if exists.
8. If there are nodes in $G(V, E)$ that are yet to be visited, then repeat from 2.
9. Compute hash of each node x as follows: $H_y \leftarrow \mathcal{H}((p_y, r_y)||y)$.
10. To each edge-node yz, assign $H_{yz} \leftarrow \mathcal{H}(H_y||H_z)$.
11. Compute the hash of graph $gHG(V, E)$ as follows:

$$gH_{G(V,E)} \leftarrow \mathcal{H}(H_{y1}||H_{y2}||\dots||H_{ym})$$

where yi 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.

`ghRedact`:

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

`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 xi 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.

Cost: Each of the schemes g$\mathcal{H}$, `ghRedact`, and `ghVrfy`, perform 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: $O(|V| + |E|)$.

Updates using Efficient-Trees: Updates to a node in a graph when applied in its tree-representation, leads to only a constant number $O(1)$ (two) of updates to hash values. In contrast, as we saw earlier, when a node is updated, MHT requires $O(|V|)$ number of hash values to be updated. SDAG scheme requires $O(|V| + |E|)$ number of hash values to be updated. For example, consider the Figure 3. Consider that the node f is updated. SDAG requires 4 hash values to be updated. However, if we use the efficient-tree representation of the DAG, then the number of hashes updated is 2: one for the node f and another for the root. This is true for any node at any depth in a graph.

7.4 Constructions of Perfectly Collision-resistant Hashing Scheme for Graphs

We are going to use the GGM trees to generate random numbers using the length-doubling pseudo-random functions. Due to space constraints, we refer the reader to [6, 3] for a discussion on these trees.

The random numbers are used with each node, and edge-node towards computing the perfectly one-way hashes of the contents of the nodes and edge-nodes. Any scheme from Canetti's schemes can be used. The random numbers are then supplied as part of the hash values for the nodes or edge-nodes. Then hash computation, hash-redaction and hash-verification proceeds as in the previous section.

7.5 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.*

8 Conclusions and Future Works

Graphs are widely used to specify and represent data. Verifications of integrity and whether two graphs are identical or not are often required in computing that involves 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 with respect to updates to graphs: they require as little as two ($O(1)$) hash values to be updated to refresh the hash of the graph, while the Merkle Hash Technique and Search DAG schemes for trees and DAGs respectively require as many as $O(|V|)$ and $O(|V| + |E|)$. Moreover, our proposed schemes are also as efficient as these schemes: both of them require a single traversal on the graph. We have also proposed the first perfectly collision-resistant hashing schemes for graphs. We are in the process of implementing these schemes in C++.

References

1. Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *CRYPTO*, pages 455–469, 1997.
2. Ran Canetti, Daniele Micciancio, and Omer Reingold. Perfectly one-way probabilistic hash functions (preliminary version). In *STOC*, pages 131–140, 1998.
3. Ee-Chien Chang, Chee Liang Lim, and Jia Xu. Short redactable signatures using random trees. In *Proceedings of the The Cryptographers' Track at the RSA Conference 2009 on Topics in Cryptology*, CT-RSA '09, pages 133–147, Berlin, Heidelberg, 2009. Springer-Verlag.
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 *Proceedings of the 8th International Symposium on Parallel Processing*, pages 362–367, Washington, DC, USA, 1994. IEEE Computer Society.
6. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.
7. H. V. Jagadish and Frank Olken. Database management for life sciences research. *SIGMOD Rec.*, 33(2):15–20, June 2004.
8. V. Kamakoti and C. Pandu Rangan. An optimal algorithm for reconstructing a binary tree. *Inf. Process. Lett.*, 42(2):113–115, 1992.
9. J. Katz and Y. Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman & Hall/CRC, 2007.
10. D. E. Knuth. *The Art of Computer Programming*, volume 1. Pearson Education Asia, third edition, 2002.
11. Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39:21–41, 2004. 10.1007/s00453-003-1076-8.
12. R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.
13. Kai Samelin, Henrich Phls, Arne Bilzhause, Joachim Posegga, and Hermann de Meer. Redactable signatures for independent removal of structure and content. In Mark Ryan, Ben Smyth, and Guilin Wang, editors, *Information Security Practice and Experience*, volume 7232 of *Lecture Notes in Computer Science*, pages 17–33. Springer Berlin / Heidelberg, 2012.
14. Man Lung Yiu, Eric Lo, and Duncan 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.
15. Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow.*, 2(1):718–729, August 2009.