Open Access

Open Journal of Semantic Web (OJSW)
Volume 6, Issue 1, 2019

http://www.ronpub.com/ojsw
ISSN 2199-336X

Integrity Proofs for RDF Graphs

Andrew Sutton, Reza Samavi

Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4L8, Canada, {suttonad, samavir}@mcmaster.ca

Abstract

Representing open datasets with the RDF model is becoming increasingly popular. An important aspect of this data model is that it can utilize the methods of computing cryptographic hashes to verify the integrity of RDF graphs. In this paper, we first develop a number of metrics to compare the state-of-the-art integrity proof methods and then present two new approaches to generate an integrity proof of RDF datasets: (i) semantic-based and (ii) structure-based. The semantic-based approach leverages timestamps (or other inherent notions of ordering) as an indexing key to construct a sorted Merkle tree variation, where timestamps are semantically extractable from the dataset. The structure-based approach utilizes the redundant structure of large RDF datasets to compress the dataset statements prior to generating a variation of a Merkle tree. We provide a theoretical analysis and an experimental evaluation of our two proposed methods. Compared to the Merkle and sorted Merkle tree, the semantic-based approach achieves faster querying performance for large datasets. The structure-based approach is well suited when RDF datasets contain large amounts of semantic redundancies. We also evaluate our methods' resistance to adversarial threats.

Type of Paper and Keywords

Regular research paper: *integrity proof, cryptographic hashing, Merkle tree, RDF, linked data, semantic web*

1 Introduction

The Resource Description Framework (RDF) [35], developed by the World Wide Web Consortium (W3C), is an effective data model for the Semantic Web. With the Linked Data initiative [15], RDF has become increasingly popular to represent open datasets and facilitate data interoperability and aggregation in big data analytics environments. The simplicity and intrinsic flexibility of the RDF data model, the lightweight reasoning support of RDFS (RDF-schema) [33], and availability of query languages (e.g., SPARQL [36]) led to a variety of Linked Data based proposals such as scalable solutions on privacy auditing [27, 28]. As for any data model, a desirable feature for RDF is having efficient methods to generate integrity proofs of RDF statements, particularly when the size of the dataset is incrementally growing. For example, the Linked Data based log designed for encoding privacy auditing constantly grows and the integrity proof needs to be recomputed to guarantee that the log is tamper-proof and/or supports non-repudiation [31]. In this paper, we investigate the state-of-the-art methods of generating cryptographic hashes that can be used as integrity proofs for RDF datasets. We then propose two approaches for computing a cryptographic hash of RDF datasets based on the *semantic* or *structure* of the underlying dataset. Our semantic-based approach provides a more efficient method of computing a cryptographic hash for the special case of growing RDF datasets where the dataset statements carry some notion of ordering (e.g., statements are timestamped). Alternatively, the structure-based method forms a compressed version of the dataset to provide a cryptographic hash with less processing effort.

Current approaches to generate integrity verification

(cryptographic hash) for RDF datasets are either based on incremental methods where a commutative operation (e.g., concatenation or multiplication) is applied to the hash values of data items (e.g., individual RDF statements or subgraphs) to produce a hash value for the whole dataset [22, 3, 29, 13] or based on constructing different variations of a Merkle tree [23], which is a rooted binary hash tree where nodes are labeled with cryptographic hashes and the root provides a hash for the whole dataset [23, 4, 5, 20]. The hash value of an RDF graph generated using each of the proposed methods carries different properties. For example, one method may generate the same hash value when the order of statements in a dataset changes and another might be order sensitive. The methods may also differ in their running time to generate the integrity proof of a graph or in supporting an efficient method for pinpointing a specific RDF statement that contributes to a different hash value for the entire graph.

The gap that this research intends to fill is to first define a set of properties that can be used to compare and select the appropriate method of generating the integrity proof of an RDF graph depending on the requirements which the proof needs to satisfy. Second, we propose an algorithm to generate an integrity proof specifically in the context of privacy auditing (or other similar logging contexts) where the RDF statements in the log are required to be timestamped. Our method is an extension of a sorted Merkle tree in which the key is not externally generated, as is typically the case for sorted Merkle trees, but is exploited from the semantics of the RDF dataset (e.g., the keys can be retrieved through SPARQL queries). Although our algorithm is limited for special cases where a key is inherently present in the dataset, it provides the advantage of avoiding the additional cost of pre-processing for sorting or indexing the RDF dataset prior to generating an integrity proof. This is desirable particularly in privacy auditing cases where the RDF dataset is incrementally growing. Third, we propose an algorithm to generate a structural integrity proof based on compressing the semantic redundancies in the dataset. This algorithm is limited in its application to highly redundant datasets, however large RDF datasets often result in high semantic redundancies, which our algorithm leverages to reduce the cost of computing hashes of the redundant data [10, 6].

The work presented in the paper is an extension of the work that has been published in [32]. The contributions uniquely reported in this paper include an expanded set of integrity proof properties, a structure-based integrity proof approach, and an experimental evaluation of the semantic- and structure-based methods. We also extend the related work and discuss new directions for future research.

The rest of the paper is structured as follows. In Section 2, a comparative analysis of existing methods of generating integrity proofs is performed and common properties among these methods are presented. In Section 3, the algorithm for generating a semantic-based integrity proof is described. Section 4 presents the structure-based integrity proof algorithm. In Section 5, we compare our proposed algorithms with nine other integrity proof methods. We experimentally evaluate our semantic- and structure-based methods in Section 6. The related work is studied in Section 7 and we provide a number of future research directions and conclude in Section 8.

2 Integrity Proof Properties

In this section, we perform a comparative analysis between nine methods of generating a cryptographic hash value for a set of data items, which can be used as the integrity proof for the dataset. We selected the methods based on their popularity and relevance in the literature. We investigate these methods in terms of eight common properties as described below. For the remainder of the paper, we interchangeably refer to RDF statements (i.e., triples or quads) or RDF graphs as data items and a collection of statements or graphs as a dataset.

Preserving integrity proof independent to data order. This property determines if the method for computing a hash value of the entire dataset is independent from the order of the data items. It is often the case that the same set of data items are ordered in different sequences. For example, querying a database may provide the same set of data to different users, but the order of the data in the query result may be different for each user. In certain scenarios, such as when calculating digital signatures, it is undesirable for the same set of data to generate a unique hash value for each possible data item sequence order. Rather, we would like the hash value to remain *data order independent* so that the same dataset, regardless of the order of its data items, generates the same hash value (assuming that the data items themselves have not been modified).

Support for random access to a hash value. For this property, we are interested to investigate if the hash method supports random access (or provides a notion of indexing) to the hash value of a specific data item. When some data items are modified in a dataset (intentionally or maliciously) and in turn the computed hash value of the entire dataset captured the integrity violation, the ability to efficiently pinpoint hash values of which data items contributed to the inconsistency of the integrity proof is a desirable property. For example,

in Linked Data-based auditing, where privacy events are represented by RDF graphs, we need to determine which graph was modified so that auditors can focus their investigation on the modified statements. Alternatively, a lack of random access forces an auditor to examine the entire dataset to pinpoint altered subgraphs.

Pre-processing and running time. In order to achieve properties such as calculating data order independent hash values, some methods require data pre-processing, such as running the data through a sorting algorithm. Other methods require dataset compression in order to reduce the amount of hashing calculations. The additional pre-processing effort will affect the overall efficiency of the hash calculation method. Since each hash calculation method requires different amounts of pre-processing to achieve desirable properties (such as data order independence), the runtime of each method is an important factor to consider. Importantly, determining the computational cost of achieving different hash calculation properties must be examined. We break down the runtime property to analyze and compare the complexities of the pre-processing step (if applicable), insertion (the incremental step), generation (computing the integrity proof for the entire dataset at once), and query (locating a specific data item hash value in the dataset integrity proof).

Incrementality. Suppose we have a dataset, D, and a hash of the dataset H. If D is modified to D' (e.g., by inserting a new data item and the original dataset remains unchanged), the property of incrementality states that we can update the hash H to H' proportional to the amount of change that was made to produce D' (rather than entirely recomputing H' from scratch) [1, 2]. This property is particularly useful for the case of computing the hash of RDF datasets that contain common subjects and predicates as well as for incrementally growing RDF datasets (e.g., audit log data, where new log events are added to the log). Support for incrementality increases the efficiency in computing the hash for incremental changes to the underlying dataset.

Compression. When computing the integrity proof of an RDF dataset, a common approach is to compute individual statement hashes and combine the hash values into the dataset integrity proof. Especially for large RDF datasets, performing compression prior to computing the dataset integrity proof may reduce the overall hash computation runtime. Importantly, compression reduces the semantic redundancies in the dataset, the overall size of the dataset, and the number of hash calculations required. For this property, we study if the method incorporates compression in its integrity proof computation and if the inclusion of compression outweighs the additional pre-processing effort.

Proof of membership and non-membership. The last property we investigate is if the hash computation method allows proofs of membership and/or proofs of non-membership. A proof of membership means that there is way to determine if a data item is or is not at a given position in the set, without having to store or retrieve the entire dataset. Alternatively, a proof of non-membership means that we can produce evidence, such as a position or a path, that a given data item is not present in the dataset, without storing or retrieving the entire dataset.

In the following subsections, we evaluate the state-of-the-art methods of computing hash values that can be applied to RDF datasets in terms of the above properties. The results of our analysis are summarized in Table 1. Columns in this table are ordered according to the appearance of properties. Rows in Table 1 are ordered according to the algorithms and methods described below.

2.1 Linked Data Graph Digests

We first discuss two similar methods for computing the digest of Linked Data (RDF) graphs proposed by Melnik [22] and Carroll [3]. A statement in an RDF graph is composed of a subject, predicate, object triple. For each statement in a graph, Melnik [22] computes a hash value of a statement's subject, predicate, and object, and concatenates the three digests to produce the statement's digest. Upon computing a digest for each statement, the set of statement digests are sorted, concatenated together, and hashed to produce the digest of the graph. Similarly, Carroll [3] uses a sort function on the statements, concatenates the sorted statements, and computes the digest. Since both methods use a sort function, the computed hash value is independent to the order of the data items, as the same hash value will be computed for different sequences of the same data. Using concatenation means that there is no indexing ability for these methods since there is no key associated with each data item and the position of each data item in the resulting hash is dependent on the sort function.

Both methods require the use of a sort function, such as merge sort, which results in a pre-processing complexity of $O(nlog(n))$. Incrementality is not supported by these methods since there are sorting functions applied to the data prior to computing the dataset hash. Since these methods do not support insertion and the sort function is integrated into the hash computation, the generation runtimes for both methods are $O(nlog(n))$. The lack of incrementality means that an inserted data item affects the data order and requires

Table 1: Comparative analysis of integrity proof methods for RDF datasets

					Runtime Complexities									
	Algorithm	**Ordering**	**Indexing**	**Pre-processing**	**Pre-processing**	**Insertion**	**Generation**	**Query**	**Incrementality**	**Membership**	**Non-membership**	**Compression**	**Referencing**
1	Melnik [22]	Y	N	Y	$O(nlog(n))$	N/A	$O(nlog(n))$	N/A	N	N	N	N	§ 2.1
2	Carroll [3]	Y	N	Y	$O(nlog(n))$	N/A	$O(nlog(n))$	N/A	N	N	N	N	§ 2.1
3	Sayers *et al.* [29]	Y	N	N	N/A	$O(1)$	$O(n)$	N/A	Y	N	N	N	§ 2.2
4	Fisteus *et al.* [13]	Y	N	N	N/A	$O(1)$	$O(n)$	N/A	Y	N	N	N	§ 2.2
5	Merkle Tree [23]	N	Y	N	N/A	$O(log(n))$	$O(n)$	$O(n)$	N	Y	N	N	§ 2.3
6	Sorted Merkle Tree	Y	Y	Y	$O(nlog(n))$	N/A	$O(n)$	$O(n)$	N	Y	Y	N	§ 2.4
7	Position Merkle Tree [20]	N	Y	Y	$O(1)$	$O(log(n))$	$O(n)$	$O(n)$	Y	Y	N	N	§ 2.5
8	History Merkle Tree [4]	N	Y	N	N/A	$O(log(n))$	$O(n)$	$O(log(n))$	Y	Y	N	N	§ 2.6
9	Merklix Tree [5]	Y	Y	Y	$O(k)$	$O(log(n))$	$O(n)$	$O(log(n))$	Y	Y	Y	N	§ 2.7
10	Timestamp Tree [32]	Y	Y	N	N/A	$O(log(n))$	$O(n)$	$O(log(n))$	Y	Y	Y	N	§ 3
11	Compression Tree	N	Y	Y	$O(n)$	$O(log(n))$	$O(n)$	$O(log(n))$	Y	Y	Y	Y	§ 4

the dataset hash to be entirely recomputed. All of the data are concatenated when computing the hash value, which means there is no way to provide a proof of membership or non-membership without supplying the whole dataset. These comparisons are summarized in the first two rows of Table 1.

2.2 Incremental Cryptography

Sayers *et al.* [29] and Fisteus *et al.* [13] utilize incremental cryptography to compute the digest of Linked Data graphs. Incremental cryptography is the process of incrementally applying a combining function that is both commutative and associative to a set of data item hash values to produce a hash value for the whole dataset. Sayers [29] uses multiplication, whereas Fisteus [13] employs exclusive-or as the combining operation. The hash value of a graph is computed by hashing the statements of the graph and multiplying (or XORing) each statement digest together modulo a large prime number [29, 16]. Both methods can compute the same hash value for a set of data independent to the data order because of the combining operation.

However, the consequence of using the combining operation is that the position of a data item in the resulting hash value will be lost, so these methods do not support proof of membership and non-membership. Since each of these methods utilize incremental cryptography, they do not need to perform pre-processing to the data (for example, sorting), which is an improvement over the previous two methods, and results in a generation runtime of $O(n)$. Each of these methods are based on incremental cryptography, which supports incrementality. Insertions can be easily applied by hashing the statement and applying the combining operation to the statement and dataset hashes, which results in an insertion complexity of $O(1)$. The third and fourth rows of Table 1 summarize these comparisons.

2.3 Merkle Tree

A Merkle tree [23] is used for the integrity verification of an ordered set of data items. Formally, a Merkle tree is a rooted binary hash tree defined as an undirected graph $MT = (V, E)$, where V is a finite set of vertices (or nodes) and E is a finite set of edges. The vertex set of MT contains three types of elements, *root*, *internal*, and *leaf*. An example Merkle Tree is shown in Fig. 1. The *root* node has no parents and is defined as $r = h(r.leftChild||r.rightChild)$ and h is a cryptographic hash function (e.g., SHA-256). *Internal* nodes have a parent and are defined as $n_i = h(n_i.leftChild||n_i.rightChild)$. A *leaf* node has a parent and is defined as $l_i = h(data_i)$, where $data_i$ is a data item. Each node in the tree is labeled with a hash value. The label of the root h_r, is used to verify the integrity of the data items that a Merkle tree is generated for. The integrity of a data item can then be verified by reconstructing a portion of the tree up to the root and then comparing the original root hash with the recomputed hash. The root hash of a Merkle tree is dependent on the order of the data items in its leaf nodes. Each sequence of a set of data items results in a unique Merkle tree and in turn a different hash value for the root node.

To make the Merkle tree order independent, we need to either sort the data items prior to constructing the tree (Section 2.4) or determine an item's position while constructing the tree (Section 2.7). Furthermore, a Merkle tree is designed to statically generate an integrity proof of a dataset, so the entire tree must be reconstructed for incremental updates. The generation runtime is $O(n)$ as the integrity proof is generated

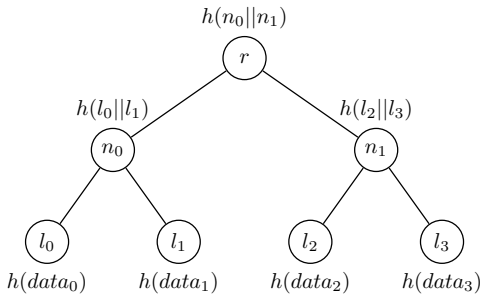


Figure 1: Merkle tree MT

at once for the entire dataset. However, there are incremental Merkle trees that support data item hash value insertions [14]. In this case, the insertion runtime is $O(log(n))$ since only a subset of the tree needs to be updated upon each insertion. Querying a hash value in a Merkle tree results in a runtime of $O(n)$ due to the lack of indexing keys (i.e., in a worst case scenario, the query must traverse the whole tree). The fifth row of Table 1 summarizes the Merkle tree properties.

2.4 Sorted Merkle Tree

If the set of data contains key-value pairs then a sorted Merkle tree can be constructed. Formally, if we have a dataset $D = \{(k, d)|k \in K, d \in D\}$, where each data item $d \in D$ has an associated key k, then we can construct a Merkle tree where the data items are sorted based on their keys. Contrary to a Merkle tree where the resulting root hash is dependent on the order of the data items, a sorted Merkle tree allows the root hash to be independent of the data order. Since each data item is associated with a key, no matter what the initial order of the data is, a sort function can be applied to the keys to order the data into a specific sequence and then construct a Merkle tree on this ordered sequence of data. However, the sort function requires additional pre-processing to get the data into an ordered sequence.

The asymptotic runtimes of a sorted Merkle tree are equivalent to a Merkle tree except additional pre-processing due to sorting requires a worst case runtime of $O(nlog(n))$. Similar to a Merkle tree, a sorted Merkle tree allows for proofs of membership without revealing or storing the entire dataset as well as lacks support for incrementality. However, unlike a Merkle tree, a sorted Merkle tree can provide proofs of non-membership by producing a path in the tree where the data item should be. Since the data items are sorted, the correct position of the data item is known. The sixth row in Table 1 summarizes the properties for a sorted Merkle tree.

2.5 Position-Aware Merkle Tree

Mao *et al.* [20] present a modified Merkle tree called a Position-aware Merkle tree (PMT) where each node in a Merkle tree can keep track of its relative position to its parent node. A node n_i in a PMT records its position in the tree and is defined as a 3-tuple $(n_i.p, n_i.r, n_i.v)$, where $n_i.p$ is n_i's relative position to its parent node, $n_i.r$ is the number of n_i's leaf nodes, and $n_i.v$ is the value of n_i [20]. A PMT allows the generation of an integrity authentication path for data verification by directly computing the root of the tree without querying the whole tree structure. Although this approach makes each node cognisant of its position in the overall tree structure, it does not utilize the underlying semantics of the data to position the data items in the tree.

Similar to a Merkle tree, a PMT does not provide data order independent integrity proofs of datasets. Since each node is aware of its position in the tree, data items can be accessed through the positioning scheme. Proofs of membership can be achieved since a data item can be checked if it is or is not at a given position due to the node position data. Unlike a Merkle tree, a PMT requires some additional pre-processing to record the 3-tuple position index for each node, which only takes $O(1)$ time. Incrementality is supported since the new data can be inserted into the tree. The runtime complexities for insertion, generation, and query are similar to a Merkle Tree. The seventh row of Table 1 summarizes the position-aware Merkle tree properties.

2.6 History-Based Merkle Tree

Crosby *et al.* propose a tree-based history data structure for tamper-evident logging called a History-Based Merkle tree [4]. The history-based tree is an append-only tree where loggers incrementally add log events to the tree and consistency proofs are generated to prove that each addition to the tree has not altered past additions. However, the addition of data items to the tree does not preserve the semantic order of the data. If log events are added to the tree out of sequence, then the resulting root hash will not be representative of the specific order of the events. Therefore, the root hash of the tree is dependent on the order of the data items.

Similar to the Merkle tree, a history-based tree supports indexing and proofs of membership since paths in the tree for each data item can be produced. Unlike a Merkle tree, a history tree supports incrementality since a log event can be incrementally added to the tree, which means the entire hash tree does not need to be recomputed. Given the index of a data item, the history tree supports $O(log(n))$ random access [4]. The insertion and generation runtimes are similar to

the previous Merkle tree based approaches. These comparisons are shown in the eighth row of Table 1.

2.7 Merklix Tree

A Merklix tree [5] (also known as a Merkle Patricia Tree [9]) is a binary tree with Merkle and radix tree properties. A radix tree is a search tree where keys are strings defined by the position of nodes in the tree and all children of a node share a common prefix of the key. Similar to a Merkle tree, a Merklix tree labels non-leaf nodes with the cryptographic hash of their children and leaf nodes with the cryptographic hash of a data item. Unlike a Merkle tree, a Merklix tree uses a radix tree structure to store elements in the tree by using a key, where each sub-tree shares a common prefix in their key [5].

A Merklix tree is also a rooted binary hash tree defined as an undirected graph $MXT = (V, E)$. An example Merklix Tree is shown in Fig. 2. The *root* and *leaf* nodes are defined the same way as for the Merkle tree. However, the *internal* nodes of a Merklix tree are defined as the ordered pair $n_i = (hash, key)$, where $hash = h(n_i.leftChild||n_i.rightChild)$ and key is the common prefix shared among the hash values of n_i's children. Assuming that the hash values are in binary, node n_0 is the subtree of nodes whose hash values have a common prefix of 0 (e.g., $h_{l_0} =$ **0**01101001... and h_{l_1} = **0**11001110...). Similarly, node n_1 is the subtree of nodes whose hash values have a common prefix of 1 (e.g., h_{l_2} = **1**0110010... and h_{l_3} = **1**1100110...).

Note that the order of the data items in Fig. 2 differs from the order shown in Fig. 1, however, the computed hash value of the root for a Merklix tree will always be the same for each possible sequence of the nodes' data items (assuming the data items have not changed) since the data items will be sorted into the correct sequence based on their hash values. Another advantage of Merklix tree is its support for proof of non-membership. Since the position of a data item in the tree is known based on the key, we can determine if a data item was part of the dataset used to construct the tree by producing a path in the tree that would lead to that data item [5].

Although a Merklix tree provides integrity verification proofs of unordered data, additional computation is required to determine the key and an element's position. Determining an element's position involves computing and comparing common prefixes, which requires $O(k)$ worst case time, where k is the length of the key. Due to the use of indexing keys, a Merklix tree supports $O(log(n))$ query time. Furthermore, in certain situations where maintaining the order of the data is important, a Merklix tree does not guarantee order preservation. For example, in security logs, preserving the order

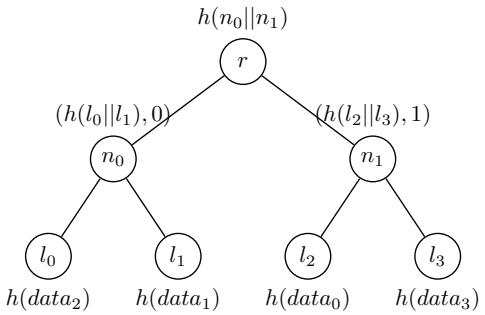


Figure 2: Merklix tree MXT

of events is critical when conducting forensic analysis and maintaining log provenance. Since a Merklix tree employs a radix tree structure, the data items will be positioned based on their hash values rather than some underlying semantic of the dataset. Due to the radix tree structure, a Merklix tree allows the insertion of data in $O(log(n))$ time, which supports incrementality. The ninth row of Table 1 shows these properties.

3 Semantics-Based Integrity Proof

In this section we recapitulate our algorithm proposed in [32] to construct a sorted Merkle tree for incrementally growing RDF datasets based on a key that is semantically extractable from the RDF dataset. Although not all RDF datasets carry a notion of a key in their dataset, the assumption of finding a key, in special cases, is reasonable since there are a number of semantic databases built on Linked Data principles in which capturing provenance assertions of an individual or collection of RDF statements is necessary. For example, in privacy auditing, a privacy log event is designed as a named graph [34] that the provenance assertions about the event include the necessary timestamp of its publication [28]. This timestamp can be a perfect candidate to be used as the key for the log event (an RDF named graph). We can leverage this existing feature of the dataset to create a sorted Merkle tree without additional pre-processing effort to generate a key or to sort the dataset based on the generated key.

A timestamp tree is an incrementally growing (i.e., append-only) sorted Merkle tree that uses pre-determined timestamp data as the key for data items in the tree. Formally, a timestamp tree is a rooted binary hash tree defined as an undirected graph $TT = (V, E)$. Similar to the trees discussed in Section 2, the vertex set of TT contains three types of elements, *root*, *internal*, and *leaf*, where each type is defined as an ordered pair $(hash, timestamp)$. For a *root* or *internal* node n_i, $hash = h(n_i.leftChild||n_i.rightChild)$

Algorithm 1: Timestamp tree insertion algorithm

Data: Previous timestamp tree: tt_{i-1}, RDF data: d_i
Result: Timestamp tree: tt_i
1 $TS_{d_i} \leftarrow$ extractTimestamp(d_i) ;
2 $position_i \leftarrow$ compareTimestamp(TS_{d_i}, tt_{i-1}) ;
3 $newLeaf_i \leftarrow$ insertData($h(d_i)$, TS_{d_i}, tt_{i-1}, $position_i$) ;
4 $tt_i \leftarrow$ recomputeParentHashes($newLeaf_i$, tt_{i-1}) ;
5 **return** tt_i

and $timestamp = n_i.rightChild.timestamp$. For a *leaf* node, $hash = h(data_i)$ and $timestamp = data_i.timestamp$.

We now illustrate the operations in the construction of TT through the sub-figures of Fig. 3, where node updates are highlighted. The construction of a timestamp tree includes one `generate` operation followed by $n-1$ `insert` operations (formally described in Alg. 1), where n is the number of data items at a given time.

Initially, the `generate`(d_0) operation starts the construction of a timestamp tree as shown in Fig. 3a, where d_0 is the first data item. The timestamp TS_{d_0} is extracted from d_0 and will be used to determine d_0's position in the tree. Since this is the first stage in the tree generation, there is no position to determine and d_0 is inserted as the root of the tree and is labeled with hash of d_0, $h(d_0)$, and TS_{d_0}. There are no other nodes in the tree so there are no parent node hashes to recompute.

At time T_1, a new data item d_1 is available and is inserted in the tree with the `insert`(tt_0, d_1) operation in Alg. 1 (Fig. 3b). Data item d_1's timestamp TS_{d_1} is extracted (line 1) and is compared with each node's timestamp in the tree at time T_0 (tt_0) to determine d_1's position (line 2). Since there is only one node in the tree, TS_{d_1} is compared with TS_{d_0}. Assuming that $TS_{d_1} > TS_{d_0}$, d_1 is inserted to the right of d_0 as a leaf l_1 labeled with the hash of d_0, $h(d_0)$, and TS_{d_0} (line 3). Since a new node was inserted in the tree, the root of the tree must be recomputed. In this case the root from Fig. 3a becomes leaf l_0 and a new root r_1 labeled with the hash of it's children, $h(l_0||l_1)$, and the rightmost child's timestamp TS_{d_1} is computed (line 4). Data item d_2 is available at time T_2 and is inserted in the tree (Fig. 3c). The extracted timestamp TS_{d_2} is compared with each node of the tree from T_1 starting at the root r_1. Assuming that $TS_{d_2} > TS_{d_1}$, we know that all children of r_1 contain timestamps before TS_{d_2} so a new root r_2 is created where the previous root r_1 becomes the left child of r_2 and d_2 becomes the right child.

Finally, at time T_3, data item d_3 is inserted in the tree (Fig. 3d). Again, the comparison with the extracted timestamp TS_{d_3} is performed on r_2 from T_2. Assuming that $TS_{d_3} > TS_{d_2}$, d_3 will be inserted to the right of d_2. Since d_2 is a leaf l_2 at T_2, a new internal node n_1 is created with d_2 as it's left child and d_3 as it's right child. Node n_1 is labeled with the hash of it's children, $h(l_2||l_3)$, and the left child's timestamp TS_{l_3} and is the right child of the new root r_3. The root r_3 updates it's hash value to be $h(n_0||n_1)$ and timestamp to be the right child's timestamp, TS_{n_1}. The process of inserting new data items through timestamp extraction, comparison, insertion, and node re-computation continues as data becomes available.

Throughout the provided tree generation example, it is assumed that $TS_{d_1} < TS_{d_2} < ... < TS_{d_n}$. This assumption is valid in situations where out of sequence data are considered invalid, such as in security and privacy logs. In such datasets, the new data item is always appended to the tree as the rightmost leaf node since the timestamps are sequentially ordered from left to right and the current rightmost leaf of the tree has the latest timestamp. In this case, computation of the hash value of the root node in each increment of a data item is limited to only recomputing the hash values of the rightmost path of the tree. For this reason, we store the right child's timestamp at each parent to provide a timestamp comparison upper bound so that we can immediately insert a new data item as the rightmost leaf node of the tree upon performing a timestamp comparison with the root.

However, if in a rare case, insertion of out-of-sequence nodes is a requirement (for example, due to system latency where appending out of sequence data items is necessary), this algorithm supports this type of insertion with the additional cost of comparing timestamps and following a path to a leaf of the tree depending on where in the tree the out-of-sequence data item ends up.

4 Structure-Based Integrity Proof

The caveat of the method described in the previous section is that the semantics we are extracting from the data might be considered confidential [19, 24]. For example, the timestamps extracted from the audit log events can be queried in the timestamp tree. These timestamps provide temporal information relating to the events, which can help an adversary determine when specific events occur in a system (e.g., when access requests and responses are performed). Rather than rely on the *semantics* of the data to generate an integrity proof, we can generate an integrity proof based on the *structure* of the data. In this section, we demonstrate how the structure of the RDF datasets can be exploited to generate structure-based integrity proofs.

The underlying data model of Linked Data is RDF with the *subject-predicate-object* triple structure.

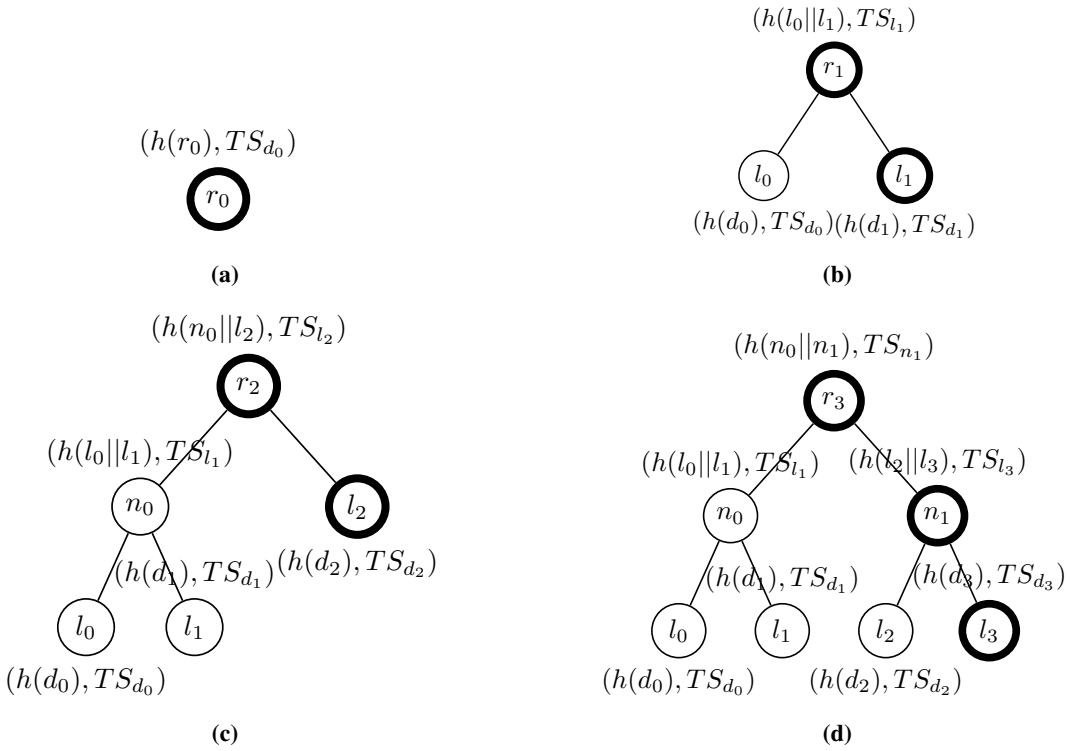


Figure 3: Timestamp tree TT Generation

Often, large datasets contain semantic redundancies, i.e., multiple repetitions in the *subjects* and *predicates* of triples. The graph representation includes redundancy in the form of repeated nodes and edges that follow power law distributions [10, 6]. Due to large portions of RDF datasets containing semantic redundancies, we can perform compression techniques in order to reduce the size of the dataset (i.e., reduce the number of statements to hash) prior to generating an integrity proof. Computing an integrity proof of a full sized dataset will result in multiple identical hash values being computed because of the semantic redundancies. It is important to note that, semantic-based integrity proof methods, such as the timestamp tree in Section 3, operate on a graph level since they rely on a group of statements (i.e., named graph) with an associated timestamp to order the data in the tree. However, a structure-based integrity proof provides more granularity by operating on a triple level (i.e., hashing individual statement components).

We leverage the RDF compression approach described in [10] to perform dataset compression prior to generating an integrity proof. The method in [10] states that a set of triples $\{(s, p_1, o_{11}), ..., (s, p_1, o_{1n_1}), ..., (s, p_2, o_{21}), ..., (s, p_2, o_{2n_2}), ..., (s, p_k, o_{kn_k})\}$ would be written as the adjacency list $s \rightarrow [(p_1, ObjList_1), ..., (p_k, ObjList_k)]$. We adapt the adjacency list to a k-ary hash tree, called a Compression Tree, so that auditing queries can be performed on the tree (e.g., querying a specific triple or triple element).

Formally, a compression tree (Fig. 4) is a rooted k-ary hash tree defined as an undirected graph $CT = (V, E)$. The $internal$ node type of the vertex set V is defined as the pair $(tripleComponentHash, childrenHash)$. The element $tripleComponentHash = h(subject)$, $h(predicate)$ or $h(object)$, where h is a cryptographic hash function, and for node n_i, $childrenHash = h(n_i.child_1||...||n_i.child_k)$. The $root$ node represents the integrity proof of the dataset and is defined as $childrenHash$ and the $leaf$ nodes are defined as the hash of the object, $h(object)$. Since the compression tree is based on the RDF triple, the tree has a maximum height of 3, where all nodes at depth 1 are common subjects, all nodes at depth 2 are common predicates, and all nodes at depth 3 are objects. The compression tree is designed for highly redundant data, where there are large amounts of common subjects and predicates. We present the generation algorithm (Alg. 2) in Section 4.1 and the insertion algorithm (Alg.3) in Section 4.2.

4.1 Compression Tree Generation

Alg. 2 takes an RDF dataset, D, as input and outputs a compression tree ct. In line 1, we perform RDF

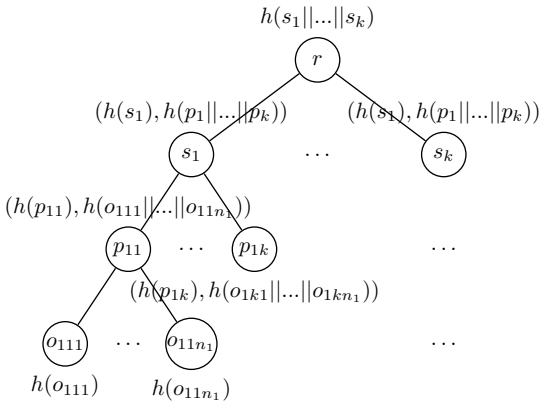


Figure 4: Compression Tree CT

compression on the dataset D. Using the method described in [10], the algorithm removes all semantic redundancies from the dataset, where common subjects and predicates are reduced to one instance. After performing the compression, we have the compressed dataset d (composed of a number of adjacency lists). In line 2, we iterate over the compressed dataset d and compute the hash (using SHA-256) of all elements in the adjacency lists, which outputs the hashed dataset TH.

It is important to note that there will be no collisions in the hash function output (assuming we use a strong cryptographic hash function) since we have reduced all redundant triple components to one instance (i.e., there will not be two subjects that hash to the same value). Lines 3-5 construct the compression tree. In line 3, we initialize the tree with a root node. Then in line 4, for each adjacency list in TH, we insert a triple to the tree. For example, in Fig. 4, we first create the root r, then the nodes of the triple s_1, p_{11}, o_{111} are added to the tree, followed by objects o_{112} to o_{11n_1} and predicates p_2 to p_k. This process is repeated for all triples in TH up to (and including) the final subject s_k (line 4 is essentially converting the set of adjacency lists to a k-ary tree). After the tree has been constructed in line 4, in line 5 we compute the hash of each node's children (starting from the object nodes up to the root node) to form the hash tree.

4.2 Compression Tree Insertion

To support incrementality, the compression tree also provides an insertion method. The input to Alg. 3 is the compression tree that the new data will be inserted to, ct_{i-1}, and the RDF triple, t_i, that will be inserted in the tree. First, we decompose the triple into its subject, predicate, and object components and compute the hash (using SHA-256) of each component in lines 1, 2, and 3, respectively. In line 4, we search the subject

Algorithm 2: Compression tree generation algorithm

Data: RDF dataset: D
Result: Compression tree: ct
1 $d \leftarrow$ RDFCompression(D) ;
2 $TH \leftarrow$ computeTripleHashes(d) ;
3 $ct \leftarrow$ createRootNode() ;
4 $ct \leftarrow$ addTripleNodes(TH, ct) ;
5 $ct \leftarrow$ computeChildrenHashes(ct) ;
6 **return** ct

Algorithm 3: Compression tree insertion algorithm

Data: Previous compression tree: ct_{i-1}, RDF triple: t_i
Result: Compression tree: tt_i
1 $sh_i \leftarrow$ computeSubjectHash(t_i) ;
2 $ph_i \leftarrow$ computePredicateHash(t_i) ;
3 $oh_i \leftarrow$ computeObjectHash(t_i) ;
4 $subjPosition_i \leftarrow$ compareSubjectHash(sh_i, ct_{i-1}) ;
5 **if** $subjPosition_i$ == $NULL$ **then**
6 $\quad ct_{i-1} \leftarrow$ insertSubject(sh_i, ct_{i-1}) ;
7 $\quad subjPosition_i \leftarrow$ updateSubjPosition() ;
8 $predPosition_i \leftarrow$ comparePredicateHash(ph_i, ct_{i-1}, $subjPosition_i$) ;
9 **if** $predPosition_i$ == $NULL$ **then**
10 $\quad ct_{i-1} \leftarrow$ insertPredicate(ph_i, ct_{i-1}, $subjPosition_i$) ;
11 $\quad predPosition_i \leftarrow$ updatePredPosition() ;
12 $ct_{i-1} \leftarrow$ insertObject(oh_i, ct_{i-1}, $predPosition_i$) ;
13 $ct_i \leftarrow$ recomputeChildrenHashes(ct_{i-1}) ;
14 **return** ct_i

nodes in the compression tree ct_{i-1} (i.e., the nodes at depth 1) for an equivalent subject hash to sh_i. If sh_i does not exist in the tree, a new node is inserted as the rightmost child of the root (line 6), otherwise sh_i exists and we do not add a new node. The position of the newly inserted node (or the existing node's position) is recorded in $subjPosition_i$ (line 7). In lines 9-13, we perform a similar procedure for the predicate. However, we use the recorded subject position, $subjPosition_i$, to reduce the scope of the predicate hash comparison search to the children of the subject node. We assume that the triple t_i has not been inserted into the tree previously (i.e., the dataset follows proper semantics and does not have multiple identical triples). Therefore, in line 14, we simply insert the object hash oh_i as a child of the predicate node at $predPosition_i$. Finally, since the compression tree is a hash tree, we recompute all of the nodes' children hashes along the inserted path. Performing the process in Alg. 3 will either: i) add a new object node to an existing $subject$-$predicate$ path; (ii) add a new object and predicate node to an existing $subject$ node; or (iii) add a new $subject$-$predicate$-$object$ path to the tree (of depth 3).

5 COMPARATIVE ANALYSIS

In this section we use the same properties discussed in Section 2 to compare our semantic- and structure-based methods with other integrity proof approaches. The tenth and eleventh rows of Table 1 summarize the properties of the timestamp tree (semantic-based) and the the compression tree (structure-based), respectively.

Data order independence. A timestamp tree produces an integrity proof independent of the data order since a data item can be inserted into the tree based on underlying data semantics (i.e., timestamps). These semantics follow a total ordering (e.g., timestamps are $ts_1 < ts_2 < ... < ts_n$), which means the data items (e.g., log events) will always be inserted in a strict order. Therefore a timestamp tree produces the same integrity proof regardless of the initial data sequence. On the other hand, the compression tree, in its current form, does not support data order independence. However, the tree can be modified to leverage incremental cryptography to achieve data order independence as proposed in [29, 1]. Rather than using concatenation to create a hash of the children of the subject and predicate nodes (as is typical for the Merkle tree data structure), we can combine the children hashes through a commutative and associative operation, thus maintaining the same dataset integrity proof regardless of the data order.

Random access. Since the data items are ordered in a specific way, a timestamp tree supports random access (indexing) to a hash value since each node in the tree is indexed with a timestamp key related to the data. The compression tree is a k-ary tree with indexes based on the component hashes of triples in the dataset, which supports random access to a hash value. A compression tree provides more granularity in its random access support than the timestamp tree since we can query hash values on the triple level rather than the graph level. A timestamp tree cannot support triple level granularity random access since the semantic required for indexing is based on metadata describing a set of statements. For example, each log event in an audit log is defined as a set of statements (i.e., a graph) with an associated publication timestamp, which is used for indexing. Therefore, the timestamp tree supports random access to hashes of graphs (i.e., graph level random access) based on metadata that carry a notion of ordering (e.g., timestamp), whereas the compression tree provides random access to hashes of triple components.

Pre-processing and running time. Leveraging the semantics of the data in the timestamp tree approach to order the items means that there is no pre-processing cost associated with applying a sort function to the data or computing an external key for indexing. However, the timestamp keys still need to be extracted through a query, although the query runtime is $O(1)$. Unlike the timestamp tree, the compression tree requires pre-processing (i.e., compression) to be done to the dataset. Therefore, since we rely on compression to be applied to the data, the pre-processing step runs in $O(n)$ times, which means the generation time for the compression tree is also $O(n)$. The running time for querying and inserting a hash value into both the timestamp and compression trees is only dependent on a subset of the tree, which results in a runtime of $O(log(n))$. These runtimes are similar to the asymptotic insertion time for Merkle-based binary hash trees (see the experiment in Section 6 for tree generation and query runtimes).

Incrementality. Both the timestamp and compression trees support incrementality through their respective insertion operations. When new data is available, the data can be added to the trees to update the integrity proof of the dataset without requiring the entire tree (and integrity proof) to be recomputed. Insertions to the tree only rely on updating the nodes along the insertion path.

Compression. A unique property of the compression tree is that it integrates dataset compression into the integrity proof computation. Unlike the other methods outlined in Table 1 (including the timestamp tree), the compression tree reduces the overall size of the dataset and the total number of hashing operations. This is particularly beneficial for large datasets with many semantic redundancies (see the experiment in Section 6).

Membership and non-membership proofs. The timestamp tree supports proofs of membership and non-membership since paths to leaves in the tree can be determined based on the timestamp indexing. The total ordering of the timetamp semantics means that paths can be produced to where log events should be located. Similarly, proofs of membership and non-membership are supported in the compression tree based on the triple hash indexing. Since the nodes of the tree are derived from the RDF triple, determining where a specific triple component should be in the tree can be easily performed.

Summary. Although the timestamp and compression trees are based on the binary hash tree, there are cases when exploiting the semantics is more beneficial than the structure and vice versa. The semantic-based approach is beneficial when the dataset contains groups of statements with associated metadata (e.g., log events with publication timestamps) since queries can be performed on the tree to verify the integrity of a specific timestamped event. However, since the data semantics are stored in the tree and accessible through queries, there is a potential for private

```
1  @prefix l2tap:<http://purl.org/l2tap#> .
2  @prefix scip:<http://purl.org/scip#> .
3  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
4  _:privacy-graph-header {
5     _:logevent a l2tap:PrivacyEvent ;
6        l2tap:memberOf _:log1 ;
7        l2tap:eventParticipant _:researcher-2 ;
8        l2tap:publicationTimestamp _e1-t1 ;
9        l2tap:eventData _:privacy-graph-body .
10    _:e1-t1 a tl:Instant ;
11       tl:atDateTime "2018-01-29T12:00:00Z"^^xsd:dateTime ;
12       tl:onTimeline _:tlphysical .
13    _:privacy-graph-body a rdfg:Graph . }
14 _:privacy-graph-body {
15    _:requests-req1 a scip:AccessRequest ;
16      scip:dataRequestor _:researcher-2 ;
17      scip:dataSender _:researcher-1 ;
18      scip:dataSubject _:patient-1234 ;
19      scip:requestedDataItem _:patient-1234-CTScan ;
20      scip:requestedPurpose _:purposes-treatment ;
21      scip:requestedPrivilege _:Use .
22    _:researcher-2 scip:requestorRole _:Principle-Investigator .
23    _:researcher-1 scip:senderRole _:Researcher . }
```

Listing 1: Typical privacy log event graph

information leakage. Utilizing the data structure in a compression tree is suitable for computing an integrity proof without revealing information about the underlying dataset, while also allowing random access to a statement component hash value (i.e., hash of a subject, predicate, or object). A compression tree is most beneficial for large redundant datasets since the integrated compression eliminates the need to perform many repetitive hash calculations.

6 Experimental Evaluation

In this section, we report the experimental evaluation of our semantic-based approach with Merkle tree variations in terms of generation and query runtimes. We describe the dataset used in the experiments in Section 6.1 and our experimental setup in Section 6.2. We report the results of two sets of experiments for the semantic- and structure-based integrity proof methods in Sections 6.3 and 6.4, respectively. Additionally, we evaluate the runtime of our structure-based method to determine the amount of redundancies that need to be present in the dataset to benefit from compression. We also include an evaluation of the memory used during the compression process. Finally, in Section 6.5, we discuss how our methods can withstand types of security-related threats.

6.1 Dataset

For both the semantic- and structure-based integrity proof experiments we leveraged the Linked Data Log to Transparency, Accountability, and Privacy (L2TAP) privacy audit log framework [28] to generate synthetic privacy audit log events. An example log event is shown in Listing 1. The events in an L2TAP log are composed of a header representing provenance semantics (lines 4-13), and a body describing privacy semantics (lines 14-23). For the semantic-based integrity proof experiment, we constructed our privacy audit log dataset by incrementally generating log events (Listing 1) over time so that each event has a different timestamp, resulting in our dataset containing 1 million events. The publication timestamp for each event is captured in the event header in line 11 (this is extracted in the timestamp tree generation). For the structure-based integrity proof experiment we leverage a fixed-size subset of our dataset (1 million triples). We vary a subset of the 1 million triples (i.e., 0% to 100%) to be redundant triples (i.e., equivalent subjects and predicates).

We understand the limitation of using synthetic data for experimental evaluation. However, the validity of experiments designed in this paper is independent to the content of RDF graphs since we made minimal assumptions about the datasets used for the experiments: i.e., presence of a notion of ordering (timestamp) for the semantic-based experiments and having varying levels of

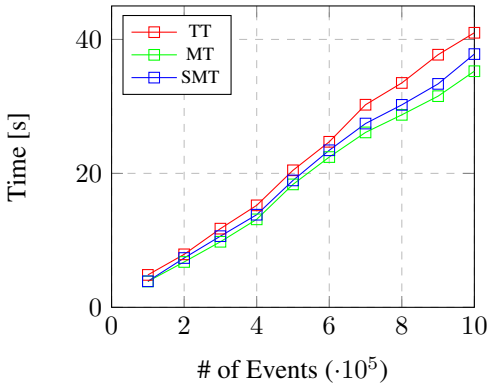


Figure 5: Tree generation runtime

redundancy (10%, 20%, 30%, etc.) for the structure-based experiments. Therefore, any RDF dataset with these characteristics, regardless of its content, would qualify for our experiments.

6.2 Test Environment

Prior to running the experiment, we generated increasing amounts of the privacy log event in Listing 1 with sequential timestamps. The event graph hash computation, tree generation, and hash query operations were run on a MacBook Pro with a 2.9 GHz Intel Core i7 processor and 16 GB of memory. The semantic- and structure-based integrity proof methods were implemented in Java and the *System.nanoTime()* Java method was used to measure the elapsed execution time of the experiments. The memory usage was measured using the Java runtime *totalMemory()* and *freeMemory()* methods. The recorded time does not take into account the time to generate the audit log events (these were pre-computed before the experiment). To account for variability in the testing environment, each reported elapsed time is the average of twenty independent executions.

6.3 Semantic-Based Experiment

In the context of privacy auditing, it is assumed that all events in a privacy log have an associated publication timestamp. We want to demonstrate how leveraging the underlying semantics of the data can improve the Merkle tree-based integrity proof methods. Specifically, we are interested in evaluating two aspects of the methods: *integrity proof generation* and *data hash query* runtimes. In our semantic-based integrity proof experimental evaluation, we compare our timestamp tree approach (Section 3) with the standard Merkle tree (Section 2.3) and the sorted Merkle tree (Section 2.4).

Integrity proof generation. We opted to perform static tree generation rather than incremental tree generation. Static tree generation constructs the tree at once for n data items, whereas incremental generation constructs a new tree for each data item $1...n$. Although we have shown the incremental generation approach in Alg. 1, we can easily adapt this algorithm for statically generating the tree (i.e., rather than inserting a single data item each iteration, we run the algorithm over the whole dataset at once). In a privacy auditing context, incremental generation is leveraged for when the integrity proof method requires real-time updates, where new log events can be inserted to the tree as they become available. On the other hand, static generation is leveraged for retrospective auditing, where we have a set of generated log events and we want to compute an integrity proof for that dataset. We decided to evaluate the static tree generation since it better highlights the differences in overall runtimes between the three tree-based methods for datasets of increasing sizes.

Fig. 5 depicts the results of the integrity proof generation runtime experiment comparing the timestamp tree (TT), Merkle tree (MT), and sorted Merkle tree (SMT) for log datasets of increasing sizes (100,000 to 1,000,000 log events). The standard Merkle tree constructs a binary hash tree from the data with no pre-processing steps involved in the construction. A sorted Merkle tree extends this process to include a sorting step prior to constructing the tree. This sorting process requires a key to be generated to order the data. A timestamp tree extracts the key used to sort the data from the data (i.e., timestamp) and additionally carries the timestamp ordering semantics throughout the entire tree. In this experiment, the data are ordered sequentially based on their timestamps prior to constructing the tree (as is typically the case for audit log events). As shown in Fig. 5, the Merkle tree has the lowest generation runtime since it simply computes the hash of each data item and constructs the tree. Since the log events are generated in order prior to constructing the tree, the sorting step for the sorted Merkle tree negligibly adds to the processing time. The timestamp tree has the additional step of extracting the timestamp from the event and storing that information in the nodes of the tree, which yields a slight increase in runtime over the other two methods. Based on the results, it can be seen that all three methods have similar generation runtimes and that the timestamp extraction and comparison for the timestamp tree negligibly adds to the runtime.

Data hash query. The ability to query the integrity proof of a privacy audit log for a specific event is beneficial for auditors when conducting an investigation. Specifically, an auditor may ask the question of "did this

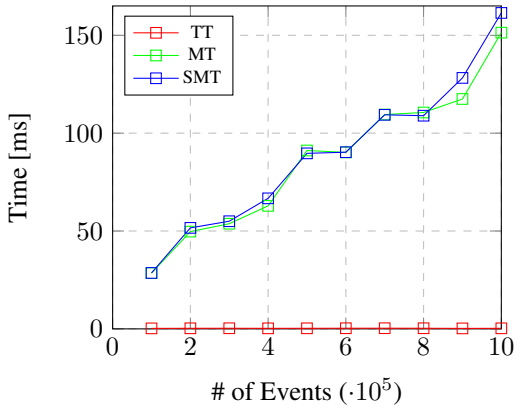


Figure 6: Data hash query runtime

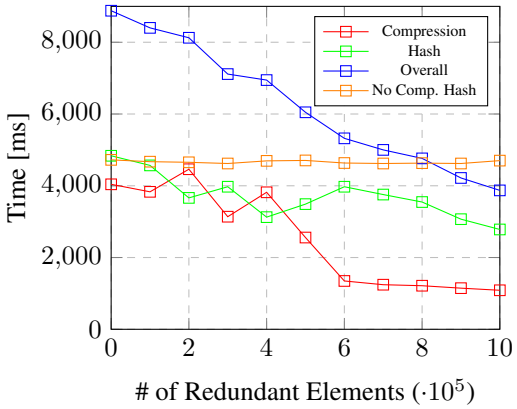


Figure 7: Data compression runtime

event produce a hash value that was part of the dataset integrity proof?". We evaluate the three methods on their ability and effectiveness at supporting such a query. After the tree is generated, we compute the hash of a data item and query the tree for that data item. A typical auditing query would want to check if the tree contains the hash of the latest event in the log. To evaluate the query execution runtime for each tree method, we perform a black-box query, where the auditor running the query is not aware of the underlying tree ordering or semantics (i.e., does not know the internal structure of the timestamp tree, sorted Merkle tree, etc.). Querying a timestamp tree is similar to a binary search through the tree (i.e., comparing timestamps at each node), whereas querying the Merkle tree methods resembles a depth first search (i.e., brute-force checking each path in the tree).

Fig. 6 shows the results of the data hash query runtime experiment. Although the sorted Merkle tree orders the data items prior to constructing the tree, under the assumption that the auditor is performing a black-box query, they cannot exploit the fact that the latest event is in the rightmost leaf position of the tree. Therefore, the query execution time is similar to the Merkle tree. Furthermore, the sorted Merkle tree does not carry the underlying data semantics throughout the tree (as opposed to the timestamp tree, which carries the timestamps in all tree nodes), so we cannot perform a binary tree search, rather we must rely on the depth first search method. Since the Merkle and sorted Merkle tree methods rely on depth first searching, the query must traverse $2 \cdot d - 1$ nodes, where d is the size of the dataset (i.e., the query must check all leaves to find the data hash). Conversely, the timestamp tree only requires traversing h nodes, where h is the height of the tree (since the query is performing binary search based on the timestamps). Since $h \ll 2 \cdot d - 1$ and h grows slowly, the timestamp tree query runtime is sub-linear compared with the Merkle and sorted Merkle trees. For example, for a dataset of size 30,000 the timestamp tree query takes about 0.32 ms, whereas the Merkle and sorted Merkle trees take around 54 ms. If we increase the dataset to 40,000 the timestamp tree remains at around 0.32 ms, whereas the Merkle trees increase to 64 ms.

Although the timestamp tree does not improve over the Merkle tree generation performance, the additional timestamp semantics greatly reduces the query runtime. While the sorted Merkle tree produces identical integrity proofs independent to the order of data, similar to the timestamp tree, we cannot leverage this property when querying, especially in a black-box query scenario, since the tree does not carry the semantics of the data ordering throughout the tree. When comparing the three tree-based integrity proof methods against the generation and query runtime aspects, the timestamp tree is a more suitable structure for audit log based tasks.

6.4 Structure-Based Experiment

Although the structure-based method is based on the Merkle binary hash tree, we did not perform an experimental comparison of the compression tree with the methods used in the semantic-based experiment. The purpose of the structure-based experiment is to determine if performing compression reduces the runtime for hashing the dataset and where a breakpoint occurs. Alternatively, the semantic-based approach is a modification of an existing approach (sorted Merkle tree), so we wanted to compare our semantic approach with the Merkle tree and demonstrate that the semantics in the tree benefits an auditor's query. Furthermore, the two approaches operate at different levels of granularity; the semantic approach operates on the graph level, whereas the structure approach operates at the triple level. Therefore, in our structure-based experiment, we evaluate the runtime of our compression method and the memory overhead incurred during the compression

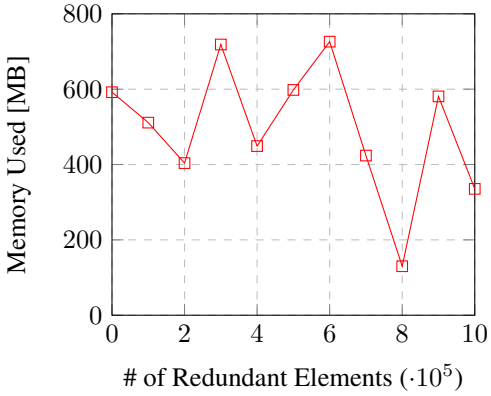


Figure 8: Memory usage during compression process (fixed dataset size)

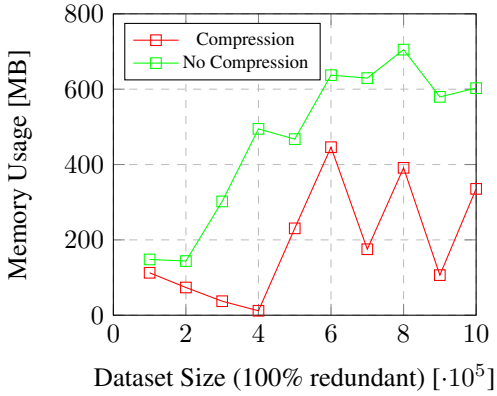


Figure 9: Comparison of memory usage between compression and non-compression processes (variable dataset size)

process.

Compression. The results of the structure-based integrity proof experiment are shown in Fig. 7. We first computed the integrity proof of the dataset (i.e., hashing each triple component) with no compression as the number of redundant triples increases. Since no compression is performed, the hashing runtime is constant for all redundancy sizes and provides a baseline to compare the compression runtime with. We then performed the same experiment, however we apply dataset compression to reduce the number of redundant triples prior to hashing the triples. This process is broken down into compression, hashing, and the overall computation (compression and hashing) and we report each elapsed execution time in Fig. 7. We want to experimentally verify where a breakpoint is to determine how many redundancies must be present in the dataset to benefit from performing compression prior to hashing the dataset. From our reported results, performing compression prior to hashing will reduce the overall integrity proof runtime if the dataset contains at least 80% redundant elements. Fig. 7 shows that the overall computation (compression and hashing) intersects the (no compression) hash computation when the dataset contained 80% redundancies. After this point, the overall computation runtime was lower than the hash computation without compression.

Despite requiring at least 80% redundancies in the dataset to benefit from applying compression prior to generating an integrity proof, there are still cases where datasets contain redundancies of this size. Large RDF datasets follow power law distributions in the amount of redundant triple components in the dataset. For example, the Linked Data log for privacy auditing [28] allows participants to encode privacy policies and obligations related to the sharing of data. Often, there are many clauses that need to be encoded in the log, which results in a large amount of semantic redundancies on both the subject and predicate level (i.e., there are many obligations associated with a single privacy policy). Especially for large audit logs, there will be many redundancies of this nature and applying compression to the dataset would reduce the integrity proof generation runtime.

Memory. A limitation of the structure-based method is that the data structure needs to be loaded into the main memory in order to generate the integrity proof. Due to this limitation, the structure-based approach may not be able to process very large datasets (depending on the memory allocation and size). Fig. 8 and 9 depict the memory usage during the compression and integrity proof process. In Fig. 8 we ran the compression process over the same dataset as reported in Fig. 7 (i.e., fixed dataset size with increasing amounts of redundant statements). Although the memory usage reciprocates between some values, there is an overall trend in declining memory usage as the dataset redundancies increase. For datasets with many redundancies, the memory usage decreases since there are less statement hashes to process (due to the compression). Furthermore, in Fig. 9 we compare the memory usage between the compression and non-compression methods. For this experiment, we leveraged a variable dataset size composed entirely of redundant triples. It can be seen that the compressed version requires less memory than the non-compressed version, however, the memory usage begins to increase as the dataset size increases. Therefore, for very large datasets ($\gg 10^5$), there may be problems processing the dataset if there is limited memory available.

6.5 Threat Model Evaluation

When the items in a dataset have a temporal aspect, as in the case of auditing, an adversary may change the order of log events to avoid the detection of malicious activity. For example, at time t_0 a log recorded the authorization for a process p_0. At time t_1, an adversary performs an action and is recorded in the log. At time t_2, a benign entity performs another action and is recorded in the log. Suppose a security incident occurs and upon investigation, is determined to have occurred at time t_1 (and unknowingly to the investigators, was ultimately caused by the adversary's actions at time t_1). By attempting to reorder the data items in the timestamp tree to hide malicious activity, the adversary makes the appearance that the benign entity's actions caused the security incident. A timestamp tree should be able to detect and prevent the malicious attempt to incorrectly insert and change the order of the leaves.

In order to prevent an adversary from reordering the data items in a timestamp tree, we can utilize a chaining mechanism that takes advantage of the underlying data order semantics to detect and prevent this malicious behavior. When a new data item is being inserted into the tree, the data item leaf node's label is modified to additionally contain the hash of the previous root of the tree. Originally, a data item leaf in a timestamp tree is labeled with the pair $(h(d_i), TS_{d_i})$. The modified label contains the triple $(h(d_i), TS_{d_i}, h(r_{i-1}))$, where r_{i-1} is the previous root of the tree. By adding the hash of the previous tree root to the leaf node, an adversary could not successfully insert a data item in the incorrect location. Since the tree is incrementally growing over time, the adversary cannot retrospectively calculate the correct $h(r_{i-1})$ for the previous tree root. Of course for this method to work, all tree root hashes must be published, for example in a blockchain [31, 21], so that they can be used for integrity verifications.

The chaining method previously described can also be applied at the data item generation level. For example, in a privacy auditing scenario, where there are multiple participants contributing to the log, each participant can link together their generated log events. Specifically, if a participant is inserting event data d_i into the tree and they have previously generated data d_{i-1} and d_{i-2}, the leaf node for d_i would be labeled with the pair $(h(d_i||d_{i-1}||d_{i-2}), TS_{d_i})$. Therefore, an adversary could not alter a participant's event data in the tree since they do not possess all of a participant's log events.

Contrary to the semantic-based integrity proof method, the structure-based method does not reveal any private information about the dataset. From a semantic perspective there is no adversarial threat to the data structure. However, leveraging compression techniques adds another layer of complexity to adversaries attempting to reveal plaintext data. Cryptanalysis relies on exploiting redundancies in the plaintext, which means large RDF datasets that contain many redundancies helps adversaries reduce the possible number of plaintext [30]. When encrypting (or computing digital signatures), compression resolves the risk of adversaries exploiting dataset redundancies to reveal the plaintext triples.

7 Related Work

A number of related work has been described in the comparative analysis of Section 2. In this section, we discuss the additional related work in auditing, Linked Data-based digital signatures, RDF graph compression techniques, and RDF triple indexing.

Google's DeepMind Health Verifiable Data Audit project is developing a real-time auditing platform for health research data that uses an append-only log that records data usage events in a Merkle tree structure [14, 7]. This approach can be extended to capture the time of any data interaction, which can be utilized in the log tree structure such as in our method. Lindqvist presents a protocol for producing verifiable privacy-preserving membership proofs of Merkle trees [19]. In the context of audit logs, Lindqvist's protocol allows auditors to query the logs for audit proofs of log entries while preserving the privacy of unrelated entries. Since our timestamp tree is an extension of a sorted Merkle tree, where the leaves are ordered, there is the potential of leaking information about adjacent leaves when querying the tree [19, 24]. Supplementing our approach with Lindqvist's protocol has potential for preventing undesirable data leakage.

Kleedorfer *et al.* propose a Linked Data-based messaging system where conversations can be verified through digital signatures [18]. This approach preserves the integrity of conversations by chaining message signatures, where all messages and signatures are defined as RDF graphs. The graph hashes used for the digital signatures are calculated using incremental cryptography. However, this approach does not exploit the fact that the message graphs contain timestamp information for each message, which our approach can use to semantically preserve the order of the messages. Kasten *et al.* provide a framework for signing graph data where a number of graph hashing methods for the signatures are discussed [16]. Similarly, this approach does not leverage available data in the graph datasets to construct a hash value to achieve sorted Merkle tree-like properties, which is desirable for preserving the data order.

A number of approaches to RDF compression

are discussed in [10], including direct compression, adjacency lists representations, and dictionary+triples decomposition. Direct compression applies zip based compression to an RDF dataset, whereas dictionary+triples decomposition requires constructing a catalog of the information entities in an RDF graph with high levels of compression [11]. We leverage the adjacency list approach rather than direct or dictionary compression since using dedicated data structures (i.e., adjacency lists) efficiently compresses the dataset [10]. Fernandez *et al.* extend the dictionary+triples decomposition approach and propose a new representation format called Header-Dictionary-Triples (HDT), where the header includes metadata describing the RDF dataset, the dictionary organizes all RDF graph identifiers, and the set of triples comprises the structure of the RDF graph [11, 12]. An HDT representation of an RDF graph provides a compact dataset for the exchange and publication of huge RDF datasets. Pan *et al.* propose a method to reduce the number of triples in RDF documents and serialise the graph based on the structure of the RDF document [25]. Khatchadourian builds a representation of an RDF dataset by generating a structural summary of the graph [17]. However, this approach focuses on capturing the semantics of the dataset rather than capturing the exact signature of the dataset (as is applicable in our case).

Leveraging the structural nature of RDF graphs for indexing is important for RDF data management system performance and scalability. Picalausa *et al.* present a framework for designing and using RDF structural indexes [26]. Erling discusses applying indexing schemes, column-wise compression and adopting column store techniques in the Virtuoso RDF management system [8].

8 Conclusions

In this paper we defined a set of properties that can be used to analyze appropriate methods of generating integrity verification proofs for RDF datasets. Current methods are based on incremental procedures that utilize a commutative operation or use constructions of Merkle trees to produce a hash value for a dataset. We proposed two tree-based approaches for generating an integrity proof of an RDF dataset: i) semantics-based and ii) structure-based. Our semantics-based integrity proof approach is an extension of a sorted Merkle tree, called a timestamp tree, where keys are semantically extractable from an RDF dataset (e.g., timestamps) and exploited to produce an integrity proof that achieves characteristics such as data order independent integrity proofs, random access, and proofs of membership and non-membership. The keys that are extracted from the dataset could be considered private information (e.g., timestamps), so we propose an alternative approach that leverages the structure of the RDF dataset. We apply compression techniques to the dataset to remove semantic redundancies prior to constructing a tree based on the RDF triple, called a compression tree. The paper includes an evaluation of the timestamp tree and compression tree generation algorithms compared to other integrity proof methods and an adversarial threat model. Based on the results, our methods are comparable to existing methods in terms of runtime growth and can resist adversaries that attempt to insert incorrect data to the tree.

There are a number of directions for future work for the semantic- and structure-based trees. First, in some application domains (e.g., in privacy audit logs) the nature of the chosen key used in a timestamp tree may be considered private information. Further work can be done to apply privacy-preserving techniques such as fully homomorphic encryption or zero knowledge proofs, where the exact timestamp is not revealed but the data insertion comparisons can still be performed over the encrypted data. Second, identical timestamps are stored in multiple tree nodes, resulting in redundant data in the tree. Node data optimization needs to be investigated to determine indexing schemes that allow the random access of data where a subset of the nodes carry a key rather than all nodes. Third, further Linked Data specializations can be applied to the compression tree. As described in [16], canonicalization and serialization functions should be applied to generalize the algorithm. Prior to performing any hash functions to the data, we can apply a canonicalization function to normalize the data followed by a serialization function to convert the canonicalized data into a sequential representation [16]. Furthermore, handling blank nodes and common or redundant URIs needs to be investigated in order to prevent unwanted collisions from the hash function.

Acknowledgements

Support from NSERC, SOSCIP and Vector Institute for Artificial Intelligence is acknowledged.

References

[1] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Annual International Cryptology Conference*, 1994, pp. 216–233.

[2] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography and application to virus protection," in *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, 1995, pp. 45–56.

[3] J. J. Carroll, "Signing RDF graphs," in *International Semantic Web Conference*, 2003, pp. 369–384.

[4] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in *USENIX Security Symposium*, 2009, pp. 317–334.

[5] Deadalnix's Den. Introducing Merklix tree as an unordered Merkle tree on steroid. https://www.deadalnix.me/2016/09/24/introducing-merklix-tree-as-an-unordered-merkle-tree-on-steroid/. Last accessed Jan 2018.

[6] L. Ding and T. Finin, "Characterizing the semantic web on the web," in *ISWC*, 2006, pp. 242–257.

[7] A. Eijdenberg, B. Laurie, and A. Cutter, "Verifiable data structures," November 2015. [Online]. Available: https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf

[8] O. Erling, "Virtuoso, a hybrid rdbms/graph column store." *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 3–8, 2012.

[9] Ethereum Wiki. Merkle patricia trie specification. Last accessed Jan 14, 2018. [Online]. Available: https://github.com/ethereum/wiki/wiki/Patricia-Tree

[10] J. D. Fernández, C. Gutierrez, and M. A. Martínez-Prieto, "RDF compression: basic approaches," in *Proceedings of the 19th international conference on WWW*, 2010, pp. 1091–1092.

[11] J. D. Fernández, M. A. Martínez-Prieto, and C. Gutierrez, "Compact representation of large rdf data sets for publishing and exchange," in *ISWC*, 2010, pp. 193–208.

[12] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary rdf representation for publication and exchange (HDT)," *Web Semantics: Science, Services and Agents on the WWW*, vol. 19, pp. 22–41, 2013.

[13] J. A. Fisteus, N. F. García, L. S. Fernández, and C. D. Kloos, "Hashing and canonicalizing notation 3 graphs," *Journal of Computer and System Sciences*, vol. 76, no. 7, pp. 663–685, 2010.

[14] Google DeepMind Health. Trust, confidence and verifiable data audit. https://deepmind.com/blog/trust-confidence-verifiable-data-audit/. Lastaccessed Jan 14, 2018.

[15] T. Heath and C. Bizer, "Linked data: Evolving the web into a global data space," *Synthesis Lectures on the Semantic Web: Theory and Tech.*, vol. 1, no. 1, pp. 1–136, 2011.

[16] A. Kasten, A. Scherp, and P. Schauß, "A framework for iterative signing of graph data on the web," in *ESWC*, 2014, pp. 146–160.

[17] S. Khatchadourian, "Bisimulation-based structural summaries of large graphs," Ph.D. dissertation, University of Toronto, Canada, 2016.

[18] F. Kleedorfer, Y. Panchenko, C. M. Busch, and C. Huemer, "Verifiability and traceability in a linked data based messaging system," in *SEMANTiCS*, 2016, pp. 97–100.

[19] A. Lindqvist, "Privacy preserving audit proofs," 2017.

[20] J. Mao, Y. Zhang, P. Li, T. Li, Q. Wu, and J. Liu, "A position-aware merkle tree for dynamic cloud data integrity verification," *Soft Computing*, vol. 21, no. 8, pp. 2151–2164, 2017.

[21] P. Marc, "Blockchain technology: Principles and applications," in *Research Handbook on Digital Transformations*, 2015.

[22] S. Melnik, "RDF API draft: Cryptographic digests of rdf models and statements," 2001. [Online]. Available: http://www-db.stanford.edu/~melnik/rdf/api.html#diges

[23] R. C. Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Security and Privacy*, 1980, pp. 122–122.

[24] R. Ostrovsky, C. Rackoff, and A. Smith, "Efficient consistency proofs for generalized queries on a committed database," in *International Colloquium on Automata, Languages, and Programming*, 2004, pp. 1041–1053.

[25] J. Z. Pan, J. M. G. Pérez, Y. Ren, H. Wu, H. Wang, and M. Zhu, "Graph pattern based rdf data compression," in *Joint International Semantic Technology Conference*, 2014, pp. 239–256.

[26] F. Picalausa, Y. Luo, G. H. Fletcher, J. Hidders, and S. Vansummeren, "A structural approach to indexing triples," in *ESWC*, 2012, pp. 406–421.

[27] R. Samavi and M. P. Consens, "L2TAP+SCIP: An audit-based privacy framework leveraging linked data," in *CollaborateCom*, 2012, pp. 719–726.

[28] R. Samavi and M. P. Consens, "Publishing privacy logs to facilitate transparency and accountability," *Journal of Web Semantics*, 2018.

[29] C. Sayers and A. H. Karp, "Computing the digest of an RDF graph," Mobile and Media Systems Laboratory, HP Laboratories, Palo Alto, USA, Tech. Rep. HPL-2003-235, 2004.

[30] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, 2007.

[31] A. Sutton and R. Samavi, "Blockchain enabled privacy audit logs," in *ISWC*, 2017, pp. 645–660.

[32] A. Sutton and R. Samavi, "Timestamp-based integrity proofs for linked data," in *Proceedings of the International Workshop on Semantic Big Data*, New York, NY, USA, 2018, pp. 1–6.

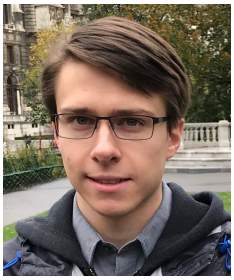
[33] W3C. RDF vocabulary description language 1.0: RDF schema. http://www.w3.org/TR/rdf-schema/. Last accessed Feb 5, 2018.

[34] W3C. RDFG: Named graph vocabulary. http://www.w3.org/2004/03/trix/rdfg-1/. Last accessed Feb 14, 2018.

[35] W3C. Semantic web standards - resource description framework (RDF). http://www.w3.org/RDF/. Last accessed Feb 5, 2018.

[36] W3C. SPARQL 1.1 query language. http://www.w3.org/TR/sparql11-query/. Last accessed Feb 5, 2018.

AUTHOR BIOGRAPHIES



Andrew Sutton recently completed his Masters (MSc) degree in Computer Science at McMaster University, Hamilton, Ontario, Canada. He received his Bachelor of Applied Science (BASc) in Honours Computer Science at McMaster University in 2016. His research interests include Information Security & Privacy, Cryptography, Blockchain, and Semantic Web.

Reza Samavi is currently an assistant professor in the Department of Computing and Software at McMaster University and a faculty affiliate with the Vector Institute for Artificial Intelligence. His research interests include Information Security & Privacy, Data Science, eHealth, Semantic Web and Linked Data. Reza received his PhD from University of Toronto and for his research on information privacy he received the Privacy Technologies Research Award from IBM and the Privacy By Design Research Award from the Information and Privacy Commissioner of Ontario.