Authentic Time-Stamps for Archival Storage

Alina Oprea and Kevin D. Bowers

RSA Laboratories, Cambridge, MA
{aoprea, kbowers}@rsa.com

Abstract. We study the problem of authenticating the content and creation time of documents generated by an organization and retained in archival storage. Recent regulations (e.g., the Sarbanes-Oxley act and the Securities and Exchange Commission rule) mandate secure retention of important business records for several years. We provide a mechanism to authenticate bulk repositories of archived documents. In our approach, a space efficient local data structure encapsulates a full document repository in a short (e.g., 32-byte) digest. Periodically registered with a trusted party, these commitments enable compact proofs of both document creation time and content integrity. The data structure, an append-only persistent authenticated dictionary, allows for efficient proofs of existence and non-existence, improving on state-of-the-art techniques. We give a rigorous security analysis of our solution and confirm through an experimental evaluation with the Enron email corpus its feasibility in practice. [1]

1 Introduction

Due to numerous regulations, including the recent eDiscovery laws, the Sarbanes-Oxley act and the Securities and Exchange Commission rule, electronic data must be securely retained and made available in a number of circumstances. One of the main challenges in complying with existing regulations is ensuring that electronic records have not been inadvertently or maliciously altered. Not only must the integrity of the records themselves be maintained, but also the integrity of metadata information, such as *creation time*. Often organizations might have incentives to modify the creation time of their documents either forward or backward in time. For example, document back-dating might enable a company to claim intellectual property rights for an invention that has been discovered by its competitor first. A party involved in litigation might be motivated to change the date on which an email was sent or received.

Most existing solutions offered by industrial products (e.g., [19]) implement WORM (Write-Once-Read-Many) storage entirely in software and use hard disks as the underlying storage media. These products are vulnerable to insider attacks with full access privileges and control of the storage system that can easily

[1] This is the full version of the paper appearing in the 14th European Symposium on Research in Computer Security (ESORICS) 2009.

compromise the integrity of data stored on the disk. Sion [39] proposes a solution based on secure co-processors that defends against document tampering by an inside adversary at a substantial performance overhead. External time-stamping services [25, 3, 2] could be leveraged for authenticating a few important documents, but are not scalable to large document repositories.

In this paper, we propose a cost-effective and scalable mechanism to establish the integrity and creation time of electronic documents whose retention is mandated by governmental or state regulations. In our model, a set of users (or employees in an organization) generate documents that are archived for retention in archival storage. A local server in the organization maintains a persistent data structure containing all the hashes of the archived documents. The server commits to its internal state periodically by registering a short commitment with an external trusted medium. Assuming that the registered commitments are publicly available and securely stored by the trusted medium, the organization is able to provide compact proofs to any third party about the *existence* or *non-existence* of a particular document at any moment in time. Our solution aims to detect any modifications to documents occurring after they have been archived.

To enable the efficient creation of both existence and non-existence proofs, we describe a data structure that minimizes the amount of local storage and the size of commitments. The data structure supports fast insertion of documents, fast document search and can be used to generate compact proofs of membership and non-membership. Our data structure implements an append-only, persistent, authenticated dictionary (PAD) [1] and is of independent interest. Previously proposed PADs rely either on sorted binary trees [30], red-black trees [1, 31] or skip lists [1], and use the node duplication method proposed by Driscoll et al. [18]. By combining ideas from Merkle and Patricia trees in our append-only PAD, we reduce the total amount of storage necessary to maintain all versions of the data structure in time, as well as the cost of non-membership proofs compared to previous approaches.

We present a rigorous security definition for time-stamping schemes that offers document authenticity against a powerful inside attacker. We provide a detailed proof that our constructions satisfy the security definition. Finally, we confirm the efficiency of our optimized construction through a Java implementation and an evaluation on the Enron email data set [16].

Organization. In Section 2 we review related literature on compliance storage, time-stamping services and authenticated data structures. We present our security model in Section 3. The details of our append-only persistent data structure, as well as some optimizations, are given in Section 4. We present the performance evaluation of our Java implementation using the Enron email data set [16] in Section 5, and conclude in Section 6. Appendix A contains the detailed security analysis of our solution.

2 Related Work

In response to the increasing number of regulations mandating secure retention of data, several products offering *compliance storage* have been released recently (e.g., EMC Centera [19]). As optical storage medium is expensive and technologically limited, most of the industrial offerings in this area enforce WORM (Write-Once-Read-Many) semantics through software, using hard disks as the underlying storage media. On the academic side, Huang et al. [26] have proposed *content-immutable storage*, also a software-based WORM solution. However, enforcing WORM storage through software is vulnerable to inside attackers with full access privileges and physical access to the disks. Sion [39] proposes a solution to secure WORM storage and prevent data modification and deletion by using active tamper-resistant hardware. In the compliance storage area we have also seen recent literature, orthogonal to our work, on designing secure indexing schemes [41, 34, 6], and secure deletion of index entries [35].

A method proposed in the early 90s to authenticate the content and creation time of documents leverages time-stamping services [25, 3]. Such services generate a document time-stamp in the form of a digital signature on the document digest and the time the document has been submitted to the service. To diminish the amount of trust in the time-stamping service, different techniques to link documents have been proposed, e.g., linear linking [25] and binary tree linking [3, 2]. Buldas et al. [9] make binary linking schemes accountable in the sense that clients could verify the relative ordering of documents generated in the same round. Subsequent accountable schemes are given in [7, 11, 5].

Besides linking and accountability, two other techniques have been designed to reduce the amount of trust in time-stamping services. In Buldas et al. [10], the time-stamping service is audited periodically by submitting the hashes of all the documents received in a round to an auditor, who checks that the time-stamp for the round has been computed correctly. A completely different proposal leverages multiple time-stamping services through a technique called "timeline entanglement" [31], at the cost of more expensive protocols. More recent research in this area is concerned with "provable secure" time-stamping services [14, 36, 12, 13].

Previous research on time-stamping as outlined above is useful to prevent back-dating and establish the relative ordering of documents, but it does not prevent forward-dating as users could obtain multiple time-stamps on the same document. Moreover, time-stamping services are not scalable to a large number of documents. In our constructions, we provide scalable methods to authenticate the content and creation time of documents archived for compliance requirements.

As our solution builds upon a new design of a persistent authenticated dictionary, our work is also related to research on authenticated data structures. *Authenticated dictionaries* (AD) support efficient insertion, search and deletion of elements, as well as proofs of membership and non-membership with short commitments. ADs based on hash trees were first proposed for certificate revocation [28, 37]. Buldas et al. [8] introduce the first undeniable authenticated search tree in the sense that proofs of both membership and non-membership

for an element can not be given. ADs based on either skip lists [22, 24, 21] or red-black trees [1] have been proposed subsequently. Among these, [21] has a broader goal of authenticating all operations performed on an outsourced file system. There exist other constructions of ADs with different efficiency tradeoffs that do not support non-membership proofs, e.g., based on dynamic accumulators [15, 23] or skip lists [4].

Persistent authenticated dictionaries (PAD) are ADs that maintain all versions in time and can answer membership and non-membership proofs for any time interval in the past. The PADs proposed first in the literature were based on red-black trees and skip lists [1], and use the node duplication method of Driscoll et al. [18]. Goodrich et al. [20] analyzed the performance of different implementations of PADs based on skip lists. PADs are used in the design of several systems related to our work. KASTS [30] is a system designed for archiving of signed documents, ensuring that signatures can be verified even after key revocation. Timeline entanglement [31] is a technique that leverages multiple time-stamping services for eliminating trust in a single service. CATS [40] is a system that enables clients of a remote file system to audit the remote server, i.e., get proofs about the correct execution of each read and write operation. While KASTS is built using node duplication and supports all operations of a PAD, neither timeline entanglement nor CATS support non-membership proofs.

The persistent authenticated data structure that we propose in our system differs from previous work by only permitting append operations, with no mechanism for deletion. This allows us to design a more space efficient data structure (without reverting to node duplication) and construct very efficient non-membership proofs.

A different and interesting model of persistent authenticated data structures based on Merkle trees, called *history trees*, has been developed recently by Crosby and Wallach [17] in the context of tamper-evident logging. The history tree authenticates a set of logged events by generating a commitment after every event is appended to the log. To audit an untrusted logger, the history tree enables proofs of consistency of recent commitments with past versions of the tree called *incremental proofs*, and membership proofs for given events. The history tree bears many similarities with our unoptimized data structure. In both constructions, events (or documents) have a fixed position in the tree, based on their index, or document handle, respectively. We organize our data structure based on document handles to enable non-membership proofs and efficient content searches. We could easily augment our unoptimized data structure with similar incremental proofs as those supported by history trees. However, generating incremental proofs for our optimized data structure is challenging, as document handles might change their position in the tree from one version to the next.

Finally, cryptographic techniques to commit to a set of values so that membership and non-membership proofs for an element do not reveal additional knowledge have been proposed [33, 29]. Micali et al. [33] introduce the notion of zero-knowledge sets, and implement it using a tree similarly organized to the binary trees we employ in our data structure. However, the goal of their system,

in contrast to ours, is to reveal no knowledge about the committed set through proofs of membership and non-membership.

3 System Model

We model an organization in which users (employees) generate electronic documents, some of which need to be retained for regulatory compliance. Archived documents might be stored inside the organization or at a remote storage provider. We assume that all documents retained in archival storage are received first by a local server $\mathcal{S}$. There exists a mechanism (which we abstract away from our model) through which documents are delivered first to the local server before being archived. $\mathcal{S}$ maintains locally some state which is updated as new documents are generated and reflects the full state of the document repository. Periodically, $\mathcal{S}$ computes a short digest from its local state and submits it to an external trusted party $\mathcal{T}$.

The trusted party $\mathcal{T}$ mainly acts as a reliable storage medium for commitments generated by $\mathcal{S}$. With access to the commitments provided by $\mathcal{T}$ and proofs generated by $\mathcal{S}$, any third party (e.g., an auditor $\mathcal{V}$) could verify the authenticity and exact creation time of documents. Thus, organizational compliance could be assessed by a third party auditor. In particular, the external party used to store the periodic commitments could itself be an auditor, but that is certainly not necessary.

Our system operates in time intervals or rounds, with the initial round numbered 1. $\mathcal{S}$ maintains locally a persistent, append-only data structure, denoted at the end of round t as $\mathsf{DataStr}_t$. $\mathcal{S}$ commits to the batch of documents created in round t by sending a commitment C_t to $\mathcal{T}$. Documents are addressed by a fixed-size name or handle, which in practice could be implemented by a secure hash of the document (e.g., if SHA-256 is used for creating handles, then their sizes is 32 bytes). For a document D, we denote its handle as h_D.

3.1 System Interface

Our system consists of several functions available to $\mathcal{S}$ and another set of functions exposed to an auditor $\mathcal{V}$. We start by describing the interface available to $\mathcal{S}$, consisting of the following functions.

$\mathsf{Init}(1^\kappa)$ This algorithm initializes several system parameters (in particular the round number $t = 1$, and $\mathsf{DataStr}$), given as input a security parameter.

$\mathsf{Append}(t, h_D)$ Appends a new document handle h_D (or a set of document handles) to $\mathsf{DataStr}$ at the current time t.

$\mathsf{GetTimestamp}(h_D)$ Returns document h_D's timestamp.

$\mathsf{GetAllDocs}(t)$ Returns all documenthandles generated at time t.

$\mathsf{GenCommit}(t)$ Generates a commitment C_t to the set of documents that are currently stored in $\mathsf{DataStr}$ and sends it to $\mathcal{T}$. The call to this function also signals the end of the current round t, and the advance to round $t + 1$.

$\mathsf{GenProofExistence}(h_D, t)$ Generates a proof π that document with handle h_D existed at time t.

$\mathsf{GenProofNonExistence}(h_D, t)$ Generates a proof π that document with handle h_D was not created before time t.

The functions exposed by our system to the auditor are the following.

$\mathsf{VerExistence}(h_D, t, C_t, \pi)$ Takes as input document handle h_D, time t, commitment C_t provided by $\mathcal{T}$, and a proof π provided by $\mathcal{S}$. It returns true if π attests that document h_D existed at time t, and false otherwise.

$\mathsf{VerNonExistence}(h_D, t, C_t, \pi)$ Takes as input document handle h_D, time t, commitment C_t provided by $\mathcal{T}$, and a proof π provided by $\mathcal{S}$. It returns true if π demonstrates that document h_D was not created before time t, and false otherwise.

A *time-stamping scheme for archival storage* consists of algorithms Init, Append, GetTimestamp, GetAllDocs, GenCommit, GenProofExistence, GenProofNonExistence available to $\mathcal{S}$, and algorithms VerExistence and VerNonExistence available to $\mathcal{V}$. Some of these algorithms implicitly call the trusted party $\mathcal{T}$ for storing and retrieving commitments for particular time intervals.

Remark. Given the basic interface defined above, we could introduce more functionality in the system. For instance, we could define an algorithm that provides proofs on the exact creation time of a document. We discuss further in Section 4.4 how we could implement such an algorithm.

3.2 Security Definition

To define security for our system, we consider an *inside attacker*, Alice, modeled after a company employee. Alice has full access privileges similar to a system administrator and physical access to the storage system (in particular to the local server $\mathcal{S}$). In addition, Alice intercepts and might tamper with other employees documents, and regularly submits her own documents to $\mathcal{S}$ for timestamping and archival. However, Alice as a rational adversary who is consciously trying to escape internal detection of fraud, behaves correctly most of the time. If she tampered with a large number of documents periodically, the risk of detection would be highly increased.

The value of documents generated by an organization is usually established after they are archived. One such example is a scenario in which a company is required to submit all emails originating from Alice in a given timeframe as part of litigation. When the company is subpoenaed, Alice might want to change the date or content of some of the emails she sent. It is very unlikely, however, that Alice predicts in advance all emails that will incriminate her later in court and the exact timeframe of a subpoena. As a second example, consider the scenario of a pharmaceutical company working on development of a new cancer drug. If the company finds out suddenly that one of its competitors already developed

a similar drug, it has incentives to back-date some of the technical papers and patent applications describing the invention.

In both cases we look to prevent the modification of the documents themselves, or their creation date, after a commitment has been generated and sent to the trusted medium. Alice is granted full access to $\mathcal{S}$ and may modify the underlying DataStr, but should not be able to make false claims about documents which have been committed to $\mathcal{T}$. Alice's goal, then, is to change a document or falsify its creation time, after a commitment has been generated and received by $\mathcal{T}$. We assume that commitments sent by the local server to the trusted party are securely stored and cannot be modified by the adversary.

$\mathsf{Exp}^{\mathsf{Ver-TS}}_{\mathcal{A}}(T)$:	$\mathsf{Exp}^{\mathsf{Ver-NE}}_{\mathcal{A}}(T)$:
$s \leftarrow \lambda$	$s \leftarrow \lambda$
for $t = 1$ to T	for $t = 1$ to T
$(\mathcal{H}_t, s) \leftarrow \mathcal{A}_1(s, t)$	$(\mathcal{H}_t, s) \leftarrow \mathcal{A}_1(s, t)$
$\mathcal{S}.\mathsf{Append}(t, \mathcal{H}_t)$	$\mathcal{S}.\mathsf{Append}(t, \mathcal{H}_t)$
$C_t \leftarrow \mathcal{S}.\mathsf{GenCommit}(t)$	$C_t \leftarrow \mathcal{S}.\mathsf{GenCommit}(t)$
$(D^*, t^*, \pi) \leftarrow \mathcal{A}_2(s)$	$(D^*, t^*, \pi) \leftarrow \mathcal{A}_2(s)$
$h_{D^*} \leftarrow h(D^*)$	$h_{D^*} \leftarrow h(D^*)$
if $\exists t^* \leq t \leq T$ such that $(h_{D^*} \notin \cup_{j=1}^{t} \mathcal{H}_j) \wedge$	if $\exists t \leq t^*$ such that $(h_{D^*} \in \mathcal{H}_t) \wedge$
$(\mathcal{V}.\mathsf{VerExistence}(h_{D^*}, t^*, C_{t^*}, \pi) = \mathsf{true})$	$(\mathcal{V}.\mathsf{VerNonExistence}(h_{D^*}, t^*, C_{t^*}, \pi) = \mathsf{true})$
return 1	return 1
else return 0	else return 0

Fig. 1. Experiments that define security of time-stamping schemes.

To formalize our security definition, our adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ is participating in one of the two experiments described in Figure 1. $\mathcal{A}$ maintains state s, and sends to the local server a set of document handles $\mathcal{H}_t$ in each round (generated by both legitimate employees and by the adversary herself). After T rounds in which documents are inserted in DataStr, and commitments are generated, $\mathcal{A}$ is required to output a document, a time interval and a proof. The adversary is successful if either: (1) she is able to claim existence of a document at a time at which it was not yet created (i.e., outputs 1 in experiment $\mathsf{Exp}^{\mathsf{Ver-TS}}$); or (2) she is able to claim non-existence of a document that was in fact committed in a previous time round by the server (i.e., outputs 1 in experiment $\mathsf{Exp}^{\mathsf{Ver-NE}}$).

Remark. It is necessary to use a trusted medium for storing the commitments to guarantee their uniqueness per time interval, as well as their authenticity and availability. A simple solution that time-stamps the commitments with a time-stamping service and stores them locally would not prevent a compromised server from time-stamping multiple commitments for the same round.

4 Time-Stamping Construction

In this section we present the design of a time-stamping scheme for archival storage. We start with a quick background on Merkle trees, tries and Patricia

trees. We then describe in detail our append-only persistent authenticated dictionary, and how it can be used in designing time-stamping schemes. We include a detailed security analysis of the construction in Appendix A.

4.1 Merkle Trees

Merkle trees [32] have been designed to generate a constant-size commitment to a set of values. A Merkle tree is a binary tree with a leaf for each value, and a hash value stored at each node. The hash for the leaf corresponding to value v is $h(v)$. The hash for an internal node with children v and w is computed as $h(v||w)$. The commitment for the entire set is the hash value stored in the root of the tree. Given the commitment to the set, a proof that a value is in the set includes all the siblings of the nodes on the path from the root to the leaf that stores that value. Merkle trees can be generalized to trees of arbitrary degree.

4.2 Tries and Patricia Trees

Trie data structures [27] are organized as a tree, with branching performed on key values. Let us consider a binary trie in which each node is labeled by a string as follows. The root is labeled by the empty string λ, a left child of node u is label by $u0$ and a right child of node u is labeled by $u1$.

When a new string is inserted in the trie, its position is uniquely determined by its value. The trie is traversed starting from the root and following the left path if the first bit of the string is 0, and the right path, otherwise. The process is repeated until all bits of the string are exhausted. When traversing the trie, new nodes are created if they do not already exist. Siblings of all these nodes with a special value null are also created, if they do not exist. Figure 2 shows an example of a trie containing strings 010, 011 and 110.

For our application, we insert into the data structure fixed-size document handles, computed as hashes of document contents. In the basic trie structure depicted in Figure 2, document handles are inserted only in the leaves at the lowest level of the tree. In consequence, the cost of all operations on the data structure is proportional to the tree height, equal to the size of the hash.

For more efficient insert and search operations, Patricia trees [27] are a variant of tries that implement an optimized tree using a technique called *path compression*. The main idea of path compression is to store a skip value skip at each node that includes a 0 (or 1) for each left (or right, respectively) edge that is skipped in the optimized tree. The optimized tree then does not contain any null values.

For instance, the null leaves with labels 00, 10 and 111 in Figure 2 could be eliminated in an optimized Patricia tree, as shown in Figure 3. In the optimized tree, we have to keep track of node labels, as they do not follow directly from the position of the node in the tree. A node label can be obtained from node's position in the tree and skip values for nodes on the path from the root to that particular node.

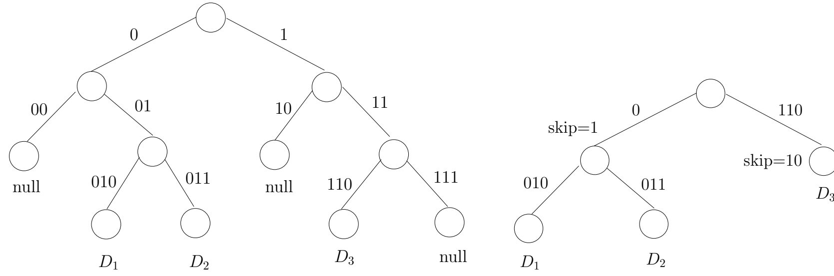

Fig. 2. Unoptimized trie for strings $D_1 = 010$, $D_2 = 011$ and $D_3 = 110$.

Fig. 3. Optimized Patricia tree for strings $D_1 = 010$, $D_2 = 011$ and $D_3 = 110$.

Knuth [27] proves that, if keys are distributed uniformly in the key space, then the time to search a key in a Patricia tree with N strings is $O(\log N)$.

4.3 Overview of the Data Structure

To construct a time-stamping scheme for archival storage, the local server needs to maintain a persistent data structure DataStr that supports insertions of new documents, enables generation of proofs of membership and non-membership of documents for any time interval, and has short commitments per interval. In the terminology used in the literature, such a data structure is called a persistent authenticated dictionary [1]. Other desirable features for our PAD is to enable efficient search by document handle, and also to enumerate all documents that have been generated in a particular time interval.

A Merkle-tree per time interval. A first, simple idea to build our PAD is to construct a Merkle tree data structure for each time interval that contains the handles of all documents generated in that interval. Such a simple data structure enables efficient appends, and efficient proofs of membership and non-membership. However, searching for a document handle is linear in the number of time intervals.

A trie or Patricia tree indexed by document handles. To enable efficient search by document handles, we could build a trie (or more optimized Patricia tree), indexed by document handles. We could layer a Merkle tree over the trie by computing hashes for internal nodes using the hash values of children. The commitment for each round is the value stored in the root of the tree. At each time interval, the hashes of internal nodes might change as new nodes are inserted into the tree. In order to generate membership and non-membership proofs at any time interval, we need a mechanism to maintain all versions of node hashes. In addition, we need an efficient mechanism to enumerate all documents generated at time t.

Our persistent authenticated dictionary. In constructing our PAD, we show how the above data structure can be augmented to support all features of a time-stamping scheme. Our data structure is a Merkle tree layered over a trie (or Patricia tree, in the optimized version). Each node in the tree stores a list of hashes (computed similarly to Merkle trees) for all time intervals the hash of the node has been modified. The list of hashes is ordered by time intervals.

To prove a document's existence at time t, the server provides evidence that the document handle was included in the tree at its correct position at time t. Similarly to Merkle trees, the server provides the siblings of the nodes on the path from the leaf to the root and the auditor computes the root hash value and checks it is equal to the commitment at time t.

A document's non-existence at time t needs to demonstrate (for the trie version) that one of the nodes on the path from the root of the tree to that document's position in the tree has value null. For the optimized Patricia tree version, non-existence proofs demonstrate that the search path for the document starting from the root either stops at a leaf node with a different handle, or encounters an internal node with both children's labels non-prefixes of the document handle.

To speed the creation of existence and non-existence proofs in the past, we propose to store some additional values in each node. Specifically, each node u maintains a list of records $\mathcal{L}_u$, ordered by time intervals. $\mathcal{L}_u$ contains one record v_u^t for each time interval t in which the hash value for that node changed. $v_u^t.\mathsf{hash}$ is the hash value for the node at time t, $v_u^t.\mathsf{lpos}$ is the index of the record at time t for its left child in $\mathcal{L}_{u0}$, and $v_u^t.\mathsf{rpos}$ is the index of the record at time t for its right child in $\mathcal{L}_{u1}$. If one of the children of node u does not contain a record at time t, then $v_u^t.\mathsf{lpos}$ or $v_u^t.\mathsf{rpos}$ store the index of the largest time interval smaller than t for which a record is stored in that child.

By storing these additional values, the subtree of the current tree for any previous time interval t can be easily extracted traversing the tree from the root and following at each node v the lpos and rpos pointers from record v_u^t. The cost of generating existence and non-existence proofs at any time in the past is then proportional to the tree height, and does not depend on the number of time intervals. In addition, all documents generated at a time interval t can be determined by traversing the tree in pre-order and pruning all branches that do not have records created at time t.

Let us consider an example. Figure 4 shows a data structure with four documents. A record v_u^t for node u at time t has three fields: $(v_u^t.\mathsf{hash}, v_u^t.\mathsf{lpos}, v_u^t.\mathsf{rpos})$. After the first round, documents D_1 and D_2 with handles 011 and 101 are inserted. Document D_3 with handle 000 is inserted at interval 2, and document D_4 with handle 010 is inserted at time 3.

A proof of existence of D_4 at time 3 consists of records $v_{010}^3, v_{011}^1, v_{00}^2, v_1^1$ and the commitment $C_3 = h(v_\lambda^3||3)$. This path includes all the siblings of nodes from root to leaf D_4 for the subtree at time 3.

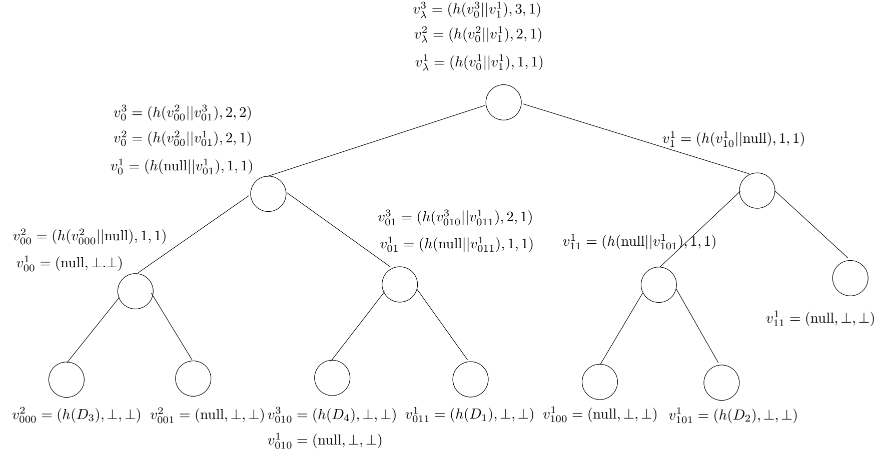

Fig. 4. Tree at interval 3.

A non-existence proof for D_4 at time 2 consists of records $v_{010}^1, v_{011}^1, v_{00}^2, v_1^1$ and the commitment $C_2 = h(v_\lambda^2||2)$. This is also a Merkle-like proof, but one that shows a null value in the leaf corresponding to D_4 for the subtree at time 2.

4.4 Algorithm Details for Tries

In this section, we give the details of the operations for an unoptimized version of our data structure based on tries, for simplicity of exposition. In the following section, we describe several optimizations including the use of Patricia trees.

We start by introducing several pieces of notation. The data structure $\mathsf{DataStr}_t$ at time t is a tree with its root denoted as $\mathsf{DataStr}_t.\mathsf{root}$. Nodes in the tree are given labels as described above. Each node with label u in $\mathsf{DataStr}$ contains its creation time denoted $u.\mathsf{crtime}$, a list of records $\mathcal{L}_u$ for different time intervals, and left and right pointers to its children, denoted $u.\mathsf{left}$ and $u.\mathsf{right}$, respectively. The list $\mathcal{L}_u$ contains a sequence of records v_u^t ordered by time t, as described above. $\mathcal{L}_u$ supports three operations: $\mathsf{append}(v)$ appends a new record v at the end of the list; $\mathsf{retrieve}(i)$ returns record in position i from the beginning of the list; and $\mathsf{pos_last}$ outputs the position of the last element appended to the list. For a node u, $u.\mathsf{parent}$ is the node's parent in the tree.

Initially, the data structure $\mathsf{DataStr}_1$ at time 1 contains only the root node $\mathsf{DataStr}_1.\mathsf{root}$. $\mathcal{L}_\lambda$ has only value $v_\lambda^1.\mathsf{hash} = \mathsf{null}$, and the left and right children are initialized to $\perp$.

Append In the Append operation given in Figure 5, a new document with handle h_D is added to $\mathsf{DataStr}_t$ in a leaf determined uniquely by its handle. The tree is traversed following the bits of h_D using the function TraverseEdge. The nodes on the path from the root to that leaf, as well as their siblings, are created in lines 13-19 (using the functions CreateNode and CreateSib), if they do not already

10.	Append(t, h_D) :	
11.	$p \leftarrow$ DataStr$_t$.root	p is initialized to the root of the tree
12.	for $i = 0$ to $\lvert h_D \rvert - 1$	
13.	par $\leftarrow p$	par will be the parent of node p after line 15
14.	$(p, s) \leftarrow$ TraverseEdge$(p, h_D[i])$	Advance one edge in the tree by handle
15.	if $p = \perp$	
16.	CreateNode$(p$, par, $h_D[i])$	Grow the tree down by adding a new node p
17.	if $s = \perp$	s is the sibling of p
18.	CreateSib$(s$, par, $h_D[i])$	Create sibling of p with hash value null if it does not exist
19.	p.left $\leftarrow \perp$; p.right $\leftarrow \perp$	Update left and right pointers for leaf h_D
20.	$v^t_{h_D}$.hash $\leftarrow h_D$;	Initialize the record for leaf h_D at time t
21.	$v^t_{h_D}$.left $\leftarrow \perp$; $v^t_{h_D}$.right $\leftarrow \perp$	
22.	$\mathcal{L}_{h_D}$.append$(v^t_{h_D})$	Append record to list of records for the leaf
23.	do	Traverse the tree from leaf h_D to root to update hashes
24.	par $\leftarrow p$.parent	par is the parent of p
25.	UpdateIndex$(p$, par$)$	Include in v^t_{par} the position of record t in $\mathcal{L}_p$ and $\mathcal{L}_s$
26.	v^t_{par}.hash $\leftarrow h(v^t_{\mathsf{par.lpos}}.\mathsf{hash} \Vert v^t_{\mathsf{par.rpos}}.\mathsf{hash})$	Compute hash value at time t for node par
27.	$\mathcal{L}_{\mathsf{par}}$.append$(v^t_{\mathsf{par}})$	Append record for time t in $\mathcal{L}_{\mathsf{par}}$
28.	$p \leftarrow$ par	Move up the tree
29.	while $p \neq$ DataStr$_t$.root	Continue until reach the root

Fig. 5. The Append algorithm.

exist in the tree. The record for leaf h_D is updated in lines 21-23. New records for the nodes on the path from the root to the leaf are also computed in lines 24-30. The hash value v^t_u.hash for a node u at time t is computed as in a Merkle tree, whereas the indices v^t_u.lpos and v^t_u.rpos are computed with function UpdateIndex. The functions TraverseEdge, CreateNode, CreateSib, and UpdateIndex are given in Figure 6.

GetTimestamp To get the timestamp of a document, its handle h_D is searched in the tree. If the search returns a node u, then GetTimestamp returns u.crtime, otherwise it returns $\perp$.

GetAllDocs To enumerate all documents generated at time t, the tree is traversed in pre-order, starting with the record v^t_{root}. The left child of a node u is visited if $v^t_u.\mathsf{lpos} = t$, and the right child is visited if $v^t_u.\mathsf{rpos} = t$. The algorithm returns all leafs visited by this pre-order search.

GenCommit To ensure that commitments for different rounds are different, the time interval is included in the computation of the commitment. The commitment generated at time t is the hash value $h(v^t_\lambda.\mathsf{hash} \Vert t)$, where v^t_λ.hash is the value stored in the root of the tree at time t.

GenProofExistence *and* GenProofNonExistence To prove that document with handle h_D existed at time t, the tree is traversed from the root following the bits of h_D. At each step, the proof is updated using function UpdateProof to include the hash of the sibling of the current node. If one of the nodes on the path from the root to the leaf did not exist at time t, then a proof of existence could not be generated and the algorithm returns $\perp$. Algorithms GenProofExistence and UpdateProof are given in Figure 7.

10. $\mathsf{DataStr}_t$.TraverseEdge(p, b):	$\mathsf{DataStr}_t$.CreateSib(s, par, b):
11. if $b = 0$	s.crtime $\leftarrow t$
12. $p \leftarrow p$.left; $s \leftarrow p$.right	v_s^t.hash $\leftarrow$ null; v_s^t.lpos $= \perp$; v_s^t.rpos $= \perp$
13. else	$\mathcal{L}_s$.append(v_s^t)
14. $p \leftarrow p$.right; $s \leftarrow p$.left	s.left $\leftarrow \perp$; s.right $\leftarrow \perp$
15. return (p, s)	if $b = 0$
16.	par.right $\leftarrow s$
17.	else
18.	par.left $\leftarrow s$
19. $\mathsf{DataStr}_t$.CreateNode(p, par, b):	DataStr.UpdateIndex(p, par)
20. p.crtime $\leftarrow t$	if par.left $= p$
21. $\mathcal{L}_p \leftarrow \phi$	$s \leftarrow$ par.right
22. if $b = 0$	v_{par}^t.lpos $\leftarrow \mathcal{L}_p$.pos_last; v_{par}^t.rpos $\leftarrow \mathcal{L}_s$.pos_last
23. par.left $\leftarrow p$	else
24. else	$s \leftarrow$ par.left
25. par.right $\leftarrow p$	v_{par}^t.rpos $\leftarrow \mathcal{L}_p$.pos_last; v_{par}^t.lpos $\leftarrow \mathcal{L}_s$.pos_last

Fig. 6. The functions TraverseEdge, CreateNode, CreateSib, and UpdateIndex of DataStr.

To prove that document with handle h_D did not exist at time t, the tree is traversed from the root following the bits of h_D. Once the first node with value null is found, the algorithm returns all the siblings on the path from the root to that null node. This represents an existence proof for a node with value null on the path from the root to leaf h_D and is sufficient to attest that node h_D was not created at time t. Algorithm GenProofNonExistence is also given in Figure 7.

10. GenProofExistence(h_D, t):	
11. $p \leftarrow \mathsf{DataStr}_t$.root; $\pi \leftarrow \phi$	p traverses the tree from root to leaf h_D; π is the returned proof
12. for $i = 0$ to $\lvert h_D \rvert - 1$	
13. $(p, v_p^t, \pi) \leftarrow$ UpdateProof$(p, h_D[i], \pi)$	Advance in the tree and update π with hash of sibling of p
14. if $v_p^t = \perp$	If node p did not exist at time t,
15. return $\perp$	then a proof of existence for h_D can not be constructed
16. return π	Return proof of document's existence at time t
17. GenProofNonExistence(h_D, t):	
18. $p \leftarrow \mathsf{DataStr}_t$.root; $\pi \leftarrow \phi$	p traverses the tree from root to leaf h_D; π is the returned proof
19. for $i = 0$ to $\lvert h_D \rvert - 1$	
20. $(p, v_p^t, \pi) \leftarrow$ UpdateProof$(p, h_D[i], \pi)$	Advance in the tree and update π with hash of sibling of p
21. if v_p^t.hash $=$ null	If node p at time t has value null, then a proof of its existence
22. return π	demonstrates that h_D was not created before time t
23. return $\perp$	If no node on the path from root to leaf had value null at time t,
24.	then document existed at time t
25. UpdateProof$(p, h_D[i], \pi)$:	
26. par $\leftarrow p$	par will be the parent of node p after line 27
27. $(p, s) \leftarrow$ TraverseEdge$(p, h_D[i])$	Advance one edge in the tree by handle; s is the sibling of p
28. if $h_D[i] = 0$	
29. $v_p^t \leftarrow \mathcal{L}_p$.retrieve$(v_{\mathsf{par}}^t$.lpos$)$	v_p^t is the record for node p at time t
30. $v_s^t \leftarrow \mathcal{L}_s$.retrieve$(v_{\mathsf{par}}^t$.rpos$)$	v_s^t is the record for sibling s at time t
31. else	
32. $v_p^t \leftarrow \mathcal{L}_p$.retrieve$(v_{\mathsf{par}}^t$.rpos$)$	
33. $v_s^t \leftarrow \mathcal{L}_s$.retrieve$(v_{\mathsf{par}}^t$.lpos$)$	
34. π.append$(v_s^t$.hash$)$	Append a left/right bit and hash of sibling s at time t to proof
35. return (p, v_p^t, π)	

Fig. 7. The GenProofExistence and GenProofNonExistence operations.

VerExistence *and* VerNonExistence. The VerExistence(h_D, t, C_t, π) algorithm checks the existence π of document handle h_D at time t. It re-computes the hashes of the nodes whose siblings are included in π (i.e., the nodes on the path from the root to leaf h_D), and checks the root hash against the commitment C_t. Similarly, algorithm VerNonExistence(h_D, t, C_t, π) traverses the path from the root to leaf h_D up to the first null node. We omit here the pseudo-code for these algorithms.

4.5 Optimizations

Path compression. We give here more details on the optimized data structure based on Patricia trees. We store at each node an additional string, called skip, which includes a 0 (or 1) for each left (or right, respectively) edge that was skipped in the optimized tree, as described in Section 4.2. To commit to node positions in the tree, the hash for an internal node is computed over the hashes of its children, as well as the skip value of the node. The hash for a leaf node is computed over the hash of the document stored at the leaf and its skip value. For each node in the tree, except for the root, we define its *position bit* to be 0 if it is a left child, and 1 if it is a right child of its parent.

A proof of existence sent to the auditor is similar to an existence proof for the unoptimized case. It consists, for each node on the path from the root to the leaf containing the document, of the hash of its sibling node and the skip value of the node. The proof is constructed in a top-down fashion, starting with the root node and descending to the leaf storing the document.

A non-existence proof is different than in the unoptimized case, since in the optimized version of the tree, the proof does not end in a null node. A proof of non-existence for h_D is in fact an existence proof for a node u with label l_u a strict substring of h_D such that either: (1) u is a leaf node; or (2) u is an internal node, but both its two children have labels incomparable to h_D in lexicographic ordering (i.e., neither is a substring of the other).

Checking an existence proof is similar to the unoptimized case, in that $\mathcal{V}$ needs to ensure the consistency of the root hash computed from the values transmitted in the proof with the commitment for that round. In addition, in the optimized version, the auditor needs to check the position of the document handle in the tree. For that, $\mathcal{V}$ constructs the label s of the leaf node storing the document handle from the proof. For each node in the proof, its position bit and skip value are appended to s. The $\mathcal{V}$ checks that s is equal to h_D.

To check a non-existence proof, $\mathcal{V}$ checks the consistency of the hashes on the path from the root to leaf node included in the proof. It also computes the labels of all nodes on the path from the root to the leaf and it needs to ensure that either: (1) the last node whose label is a substring of h_D is a leaf node; or (2) the children of the last node whose label is a prefix of h_D have labels incomparable to h_D.

Trees with larger degree. We could increase the degree of the tree from binary to any arbitrary value. This would affect the tree height, as well as the cost of

Append, GetTimestamp, GetAllDocs, GenProofExistence and GenProofNonExistence operations, and the sizes of the proofs of existence and non-existence.

For our evaluation in Section 5, we have implemented a tree of arbitrary degree with path compression. We have determined that the optimal performance for our various metrics is obtained for trees of degree 8.

Probabilistic proofs of creation time. Starting from the basic functionality we have provided in a time-stamping scheme, we could implement an algorithm that attests to the creation time of documents. One simple method to implement such an algorithm is to include a proof of document's existence at a time t and its non-existence at all previous time intervals $1, \ldots, t-1$.

With this simple approach, the proofs of creation time might become prohibitively expensive to generate and check as the number of rounds increases. We propose a method to reduce their complexity. When asking for a proof of creation time for a document, the auditor could send a set of intervals less than t and ask the server to provide non-existence proofs only for those intervals. It is important that the intervals for which the server is required to submit non-existence proofs are unpredictable to the server. In particular, this requirement is fulfilled if the intervals are chosen pseudorandomly by the auditor.

4.6 Efficiency

	Append at time t	GenProofExistence(h_D, t)	GenProofNonExistence(h_D, t)
Compressed tree	$O(1)$ node creation $\log n_t$ hash comp.	$\log n_t$ tree ops.	$\log n_t$ tree ops.
Previous schemes [1, 40]	$\log n_t$ node creation $\log n_t$ hash comp.	$\log n_t$ tree ops.	$2 \log n_t$ tree ops.

Table 1. Worst-case cost of Append, GenProofExistence and GenProofNonExistence algorithms at time t for compressed trees and previous schemes.

	Tree growth at Append	Total number of nodes in tree	Size of existence proofs	Size of non-existence proofs
Compressed tree	$O(1)$	$O(n_t)$	$(\log n_t)\|h\|$	$(\log n_t)\|h\|$
Previous schemes [1, 40]	$\log n_t$	$O(n_t \log n_t)$	$(\log n_t)\|h\|$	$2(\log n_t)\|h\|$

Table 2. Tree growth rate of Append, total number of nodes in the tree, and the size of existence and non-existence proofs at time t for compressed trees and previous schemes.

In this section, we provide a detailed comparison of the cost of the relevant metrics for our optimized compressed tree construction based on Patricia trees, and previous persistent authenticated dictionaries, based either on

red-black trees and skip lists [1], or authenticated search trees [40]. Table 1 gives the comparison for the worst-case cost of Append, GenProofExistence and GenProofNonExistence algorithms at time t (assuming that document handles are uniformly distributed). Table 2 compares the tree growth rate of Append, the total number of nodes in the tree, and the sizes of existence and non-existence proofs at time t for our data structure and previous schemes. In these tables, n_t represents the number of nodes in the data structure at time t.

All previously proposed persistent authenticated dictionaries we are aware of use the node duplication method of Driscoll et al. [18] in order to insert or delete nodes in the data structure. This adds a $O(\log n_t)$ space overhead to the data structure for every append or delete operation. The main improvements that our data structure achieves over previous schemes is the reduction in the total number of nodes in the tree, and the reduction in the size, construction and verification time of non-existence proofs. We are able to reduce the tree growth to only a constant value because in our archival storage model we only support append operations, and we disallow deletions from the data structure.

5 Experimental Evaluation

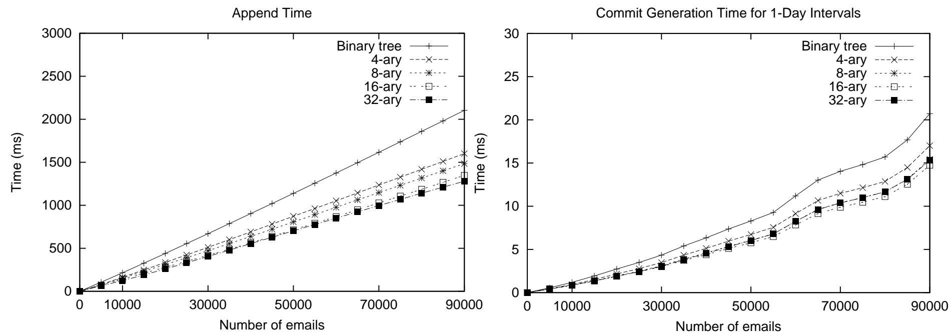

Fig. 8. Performance of Append and GenCommit operations.

To assess the practicality of our constructions, we have implemented the time-stamping scheme using the optimized data structure in Java 1.6 and performed some experiments using the Enron email data set [16]. From this email corpus, we only chose the emails sent by Enron's employees, which amount to a total of about 90,000 emails, with average size 1.9KB. The emails were created between October 30th, 1998 and July 12th, 2002. We inserted the emails into our data structure in increasing order of creation time. For our tree data structure implementation, we vary the degree of the tree by powers of two between 2 to 32. We use SHA1 for our hash function implementation.

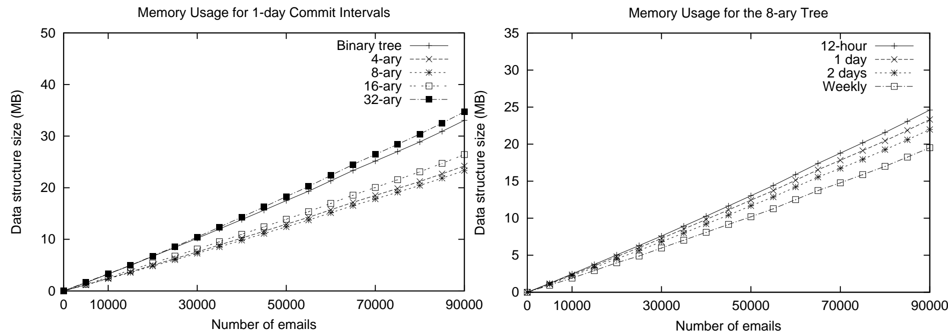

Fig. 9. Data structure storage requirements.

We report our performance numbers from an Intel Core 2 processor running at 2.32 GHz. The Java virtual machine has 1 GB of memory available for processing. The results we give are averages over five runs of simulation.

Performance of Append *and* GenCommit. We present in Figure 8 the performance of Append and GenCommit operations for different tree degrees, as a function of the number of emails in the data structure. The Append graph only includes the time to append a new hash to the data structure, and not the time to hash the email. Our experiments show that the time to hash the email is about 2.28 larger than the time to append a hash to the data structure. We get an append throughput of 42,857 emails per second for a binary tree, and 60,120 emails per second for an eight-ary tree. If we include the hash computation time, then the total append throughput is 18,699 emails per second for a binary tree, and 20,491 emails per second for an eight-ary tree. The Append operation becomes more efficient with the increase of the tree degree, as its cost is proportional to the tree height.

In our implementation, we defer the computation of hashes for tree nodes until the end of each round. Then, we traverse the tree top-down and compute new version of hashes for the nodes that change (i.e., at least one of their children is modified). We compute a new commitment for that round, even if no new nodes are added in the tree at that interval. We call the time of both these operations *the commit time*. The right graph in Figure 8 shows the commit time for intervals of one day. As some time intervals contain few emails, we choose to plot this graph as a function of the number of emails in the data structure. For $x > 0$ number of documents on the horizontal axis, the commit time includes the time to compute commitments for the time intervals spanned by the previous 1000 documents. The results show that the commit operation is efficient, e.g., for the eight-ary if there are 89,000 emails in the data structure, then the total commit time for 1,000 new emails is 15ms.

Storage requirements. Second, we evaluate the storage requirements of our data structure. The left graph in Figure 9 shows the total size of the data structure

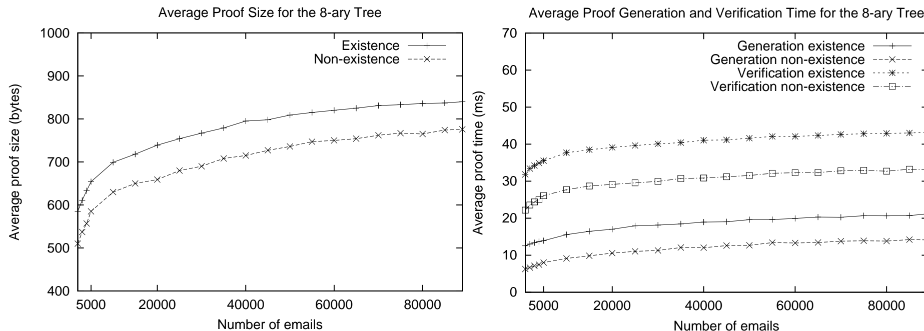

Fig. 10. Proof size and proof generation and verification time for an eight-ary tree.

for different tree degrees. It turns out that the data structure size is optimal for trees of degree 8, and increases for trees of larger degree. In fact, the memory usage of the data structure with trees of degree 32 surpasses that of the binary tree data structure. The reason for this is an artifact of our implementation: to optimize the search in the tree, we store all the children of a node in a fixed-size array. For a large degree tree, a lot of nodes are empty and unused memory is allocated. We could alternatively store children of a node in a linked-list, but this choice impacts the search efficiency.

We show how the memory usage of the data structure varies for different commit intervals in the right graph in Figure 9. The data structure is space-efficient, as it requires less than 25MB for a 12-hour commit interval, and about 20MB for a weekly commit interval, in order to store the hashes of all sent emails.

Proof cost. Finally, we evaluate the cost of proof generation and verification, as well as proof sizes, for both existence and non-existence proofs. We add emails to an eight-ary tree in batches of 1000. After a batch of 1000 emails is added, we generate existence proofs for all these 1000 emails. We also generate non-existence proofs for the 1000 emails that will be inserted in the next round. In the left graph of Figure 10, we show the average proof size over the last (or next) 1000 emails inserted in the tree, as a function of the total number of emails in the data structure. In the right graph of Figure 10, we show the average proof generation and verification time. We have performed experiments with different tree degrees, but we choose to include only the results for an eight-ary tree, which turned out to be optimal.

The experiments demonstrate that our proofs are compact in size, reaching 800 bytes for a data structure of 90,000 emails, and efficient in generation and verification time. Non-existence proofs are in general shorter and faster to generate and verify than existence proofs, since the path included in a proof does not usually reach a leaf node. Our work improves upon previous persistent authenticated dictionaries that have the cost of non-existence proofs about twice as large as that of existence proofs. As we have explained previously, we are

able to reduce the cost of non-existence proofs and the size of the data structure because we implement an append-only data structure.

6 Conclusions

We have proposed new techniques to authenticate the content integrity and creation time of documents generated by an organization and retained in archival storage for compliance requirements. Our constructions enable organizations to prove document existence and non-existence at any time interval in the past. There are several technical challenges in the area of regulatory compliance that our work does not address. Regulations mandate not only that documents are stored securely, but that they are properly disposed of when the expiration period is reached. An interesting question, for instance, is how to prove that documents have been properly deleted.

Also of interest is the ability to offload the storage of $\mathcal{S}$ to a remote server without compromising integrity of the data structure. The remote server could periodically be audited to show that it has correctly added and committed to the documents that were sent to it. As our security only depends on the correct computation of the commitment, this proof is sufficient to guarantee correct behavior, even though the data structure is kept entirely out of our control by a potentially malicious server. For our unoptimized data structure, auditing could be performed with a mechanism similar to the incremental proofs from [17]. Designing an efficient auditing procedure for our optimized data structure is more challenging and deserves further investigation.

Acknowledgement

The authors would like to gratefully thank Dan Bailey, John Brainard, Ling Cheung, Ari Juels, Burt Kaliski, and Ron Rivest for many useful discussions and suggestions on this project. The authors also thank the anonymous reviewers from ESORICS 2009 for their comments and guidance on preparing the final version of the paper.

References

1. A. Anagnostopoulos, M. Goodrich, and R. Tamassia, “Persistent authenticated dictionaries and their applications,” in *Proc. Information Security Conference (ISC)*, vol. 2200 of *LNCS*, pp. 379–393, Springer-Verlag, 2001.
2. D. Bayer, S. Haber, and W. Stornetta, “Improving the efficiency and reliability of digital time-stamping,” *Sequences II: Methods in Communication, Security, and Computer Science*, pp. 329–334, 1993.
3. J. Benaloh and M. deMare, “Efficient broadcast time-stamping,” Technical report TR-MCS-91-1, Clarkson University, Departments of Mathematics and Computer Science, 1991.

4. K. Blibech and A. Gabillon, "CHRONOS: An authenticated dictionary based on skip lists for time-stamping systems," in *Proc. Workshop on Secure Web Services*, pp. 84–90, ACM, 2005.
5. K. Blibech and A. Gabillon, "A new time-stamping scheme based on skip lists," in *Workshop on Applied Cryptography and Information Security*, vol. 3982 of *LNCS*, pp. 395–405, Springer-Verlag, 2006.
6. N. Borisov and S. Mitra, "Restricted queries over an encrypted index with applications to regulatory compliance," in *ACNS*, Springer-Verlag, 2008.
7. A. Buldas and P. Laud, "New linking schemes for digital time-stamping," in *Proc. 1st International Conference on Information Security and Cryptology (ICISC)*, pp. 3–13, Korea Institute of Information Security and Cryptology (KIISC), 1998.
8. A. Buldas, P. Laud, and H. Lipmaa, "Accountable certificate management using undeniable attestations," in *Proc. 7th ACM Conference on Computer and Communication Security (CCS)*, pp. 9–17, 2000.
9. A. Buldas, P. Laud, H. Lipmaa, and J. Villemson, "Time-stamping with binary linking schemes," in *Proc. Crypto 1998*, vol. 1462 of *LNCS*, pp. 486–501, Springer-Verlag, 1998.
10. A. Buldas, P. Laud, M. Saarepera, and J. Villemson, "Universally composable time-stamping schemes with audit," in *Proc. Information Security Conference (ISC)*, vol. 3650 of *LNCS*, pp. 359–373, Springer-Verlag, 2005.
11. A. Buldas, P. Laud, and B. Schoenmakers, "Optimally efficient accountable time-stamping," in *Proc. Public Key Cryptography (PKC)*, vol. 1751 of *LNCS*, pp. 293–305, Springer-Verlag, 2000.
12. A. Buldas and S. Laur, "Do broken hash functions affect the security of time-stamping schemes?," in *ACNS*, vol. 3989 of *LNCS*, pp. 50–65, Springer-Verlag, 2006.
13. A. Buldas and S. Laur, "Knowledge-binding commitments with application to time-stamping," in *Proc. Public Key Cryptography (PKC)*, vol. 4450 of *LNCS*, pp. 150–165, Springer-Verlag, 2007.
14. A. Buldas and M. Saarepera, "On provably secure time-stamping schemes," in *Proc. Asiacrypt 2004*, vol. 3329 of *LNCS*, pp. 500–514, Springer-Verlag, 2004.
15. J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Proc. Crypto 2002*, vol. 2442 of *LNCS*, pp. 61–76, Springer-Verlag, 2002.
16. W. Cohen, "Enron email dataset." `http://www.cs.cmu.edu/~enron`.
17. S. Crosby and D. Wallach, "Efficient data structures for tamper evident logging," in *Proc. 18th USENIX Security Symposium*, 2009.
18. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, 1989.
19. EMC, "Centera compliance edition plus." `http://www.emc.com/products/detail/hardware/centera-compliance-edition-plus.htm`.
20. M. Goodrich, C. Papamanthou, and R. Tamassia, "On the cost of persistence and authentication in skip lists," in *Proc. Workshop on Experimental Algorithms*, vol. 4525 of *LNCS*, pp. 94–107, Springer-Verlag, 2007.
21. M. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Athos: Efficient authentication of outsourced file systems," in *Proc. Information Security Conference (ISC)*, pp. 80–96, 2008.
22. M. Goodrich and R. Tamassia, "Efficient authenticated dictionaries with skip lists and commutative hashing," technical report, Johns Hopkins Information Security

Institute, 1991. Available from `www.cs.jhu.edu/~goodrich/cgc/pubs/hashskip.pdf`.
23. M. Goodrich, R. Tamassia, and J. Hasic, "An efficient dynamic and distributed cryptographic accumulator," in *Proc. Information Security Conference (ISC)*, vol. 2243 of *LNCS*, pp. 372–388, Springer-Verlag, 2002.
24. M. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an authenticated dictionary with skip lists and commutative hashing," in *DARPA Information Survivability Conference and Exposition II (DISCEX II)*, pp. 68–82, IEEE Press, 1991.
25. S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, 1991.
26. L. Huang, W. W. Hsu, and F. Zheng, "Cis: Content immutable storage for trustworthy record keeping," in *Proc. of the Conference on Mass Storage Systems and Technologies (MSST)*, 2006.
27. D. E. Knuth, *The art of computer programming*, vol. 3, Sorting and Searching. Addison-Wesley, 1973. Second Edition.
28. P. Kocher, "On certificate revocation and validation," in *Financial Cryptography*, vol. 1465 of *LNCS*, pp. 951–980, Springer-Verlag, 1998.
29. R. M. Lukose and M. Lillibridge, "Databank: An economics based privacy preserving system for distributing relevant advertising and content," Technical report HPL-2006-95, HP Laboratories, 2006.
30. P. Maniatis and M. Baker, "Enabling the archival storage of signed documents," in *Proc. First USENIX Conference on File and Storage Technologies (FAST)*, pp. 31–45, 2002.
31. P. Maniatis and M. Baker, "Secure history preservation through timeline entanglement," in *Proc. 11th USENIX Security Symposium*, pp. 297–312, 2002.
32. R. Merkle, "A cerified digital signature," in *Proc. Crypto 1989*, vol. 435 of *LNCS*, pp. 218–238, Springer-Verlag, 1989.
33. S. Micali, M. Rabin, and J. Kilian, "Zero-knowledge sets," in *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, IEEE Computer Society, 2003.
34. S. Mitra, W. W. Hsu, and M. Winslett, "Trustworthy keyword search for regulatory-compliant record retention," in *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, pp. 1001–1012, 2006.
35. S. Mitra, M. Winslett, and N. Borisov, "Deleting index entries from compliance storage," in *Proc. of the Conference on Extending Databases Technology*, 2008.
36. T. Moran, R. Shaltiel, and A. Ta-Shma, "Non-interactive timestamping in the bounded storage model," in *Proc. Crypto 2004*, vol. 3152 of *LNCS*, pp. 460–476, Springer-Verlag, 2004.
37. M. Naor and K. Nissim, "Certificate revocation and certificate update," in *Proc. 7th USENIX Security Symposium*, 1998.
38. P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *Fast Software Encryption*, vol. 3017 of *LNCS*, pp. 371–388, Springer-Verlag, 2004.
39. R. Sion, "Strong WORM," in *Proc. of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS)*, IEEE Computer Society, 2008.
40. A. Yumerefendi and J. Chase, "Strong accountability for network storage," in *Proc. 6th USENIX Conference on File and Storage Technologies (FAST)*, 2007.
41. Q. Zhu and W. W. Hsu, "Fossilized index: The linchpin of trustworthy non-alterable electronic records," in *Proc. ACM SIGMOD International Conference on Managemnt of Data*, pp. 395–406, 2005.

A Security Analysis of Construction

We start by introducing some notation, and defining a type of collision resistance denoted *everywhere second-preimage resistance* for a family of hash functions. We then proceed to the security analysis of our construction, in which we assume that the hash function used to compute document handles satisfies this notion of collision resistance.

Everywhere Second-Preimage Resistance for Hash Functions. Let $h : \mathcal{K}_h \times \mathcal{M} \rightarrow \{0,1\}^w$ be a family of hash functions. Intuitively, everywhere second preimage resistance requires that for any message $m \in \mathcal{M}$, it is hard to find a collision for a function chosen at random from the family, i.e., $m' \neq m$ such that $h_k(m') = h_k(m)$, with $k \stackrel{R}{\leftarrow} \mathcal{K}_h$. This definition has been formalized by Rogaway and Shrimpton ([38]) and it is stronger than the standard definition of second preimage resistance, but weaker than collision resistance.

Definition 1. *For an adversary algorithm $\mathcal{A}$, we define the advantage $\mathsf{Adv}^{\mathsf{spr}}_{h,\mathcal{A}} = \max_{m \in \mathcal{M}}\{\Pr[k \stackrel{R}{\leftarrow} \mathcal{K}_h, m' \leftarrow \mathcal{A}(k,m) : (m \neq m') \wedge h_k(m') = h_k(m)]\}$. The family h is everywhere second preimage resistance if $\mathsf{Adv}^{\mathsf{spr}}_{h,\mathcal{A}}$ is negligible in the key size.*

We prove below that if an adversary is successful in breaking our scheme, then there exists a document for which existence and non-existence proofs in the same interval can be provided.

Proposition 1. *Assume the hash function used to compute document handles is drawn uniformly at random from a family of everywhere second-preimage resistant hash functions. If there exists a polynomial-time adversary algorithm $\mathcal{A}$ successful in attacking a time-stamping scheme as defined in Section 3.2, then there exists a document D with handle h_D, a time interval $t \leq T$ and two proofs π and π' such that:* $\mathsf{VerExistence}(h_D, t, C_t, \pi) = \mathsf{true}$ *and* $\mathsf{VerNonExistence}(h_D, t, C_t, \pi') = \mathsf{true}$*.*

Proof: An adversary successful in attacking our time-stamping scheme outputs 1 in one of experiments $\mathsf{Exp}^{\mathsf{Ver\text{-}TS}}$ or $\mathsf{Exp}^{\mathsf{Ver\text{-}NE}}$.

1. If $\mathsf{Exp}^{\mathsf{Ver\text{-}TS}}_{\mathcal{A}}(T) = 1$, then $\exists t \leq T$ such that $h_{D^*} \notin \cup_{j=1}^{t}\mathcal{H}_j$ and $\mathsf{VerExistence}(h_{D^*}, t^*, C_{t^*}, \pi) = \mathsf{true}$. From $h_{D^*} \notin \cup_{j=1}^{t}\mathcal{H}_j$, it follows that handle h_{D^*} did not exist in $\mathsf{DataStr}$ at time t, and then there exists a proof π' such that $\mathsf{VerNonExistence}(h_{D^*}, t, C_t, \pi') = \mathsf{true}$.

2. If $\mathsf{Exp}^{\mathsf{Ver\text{-}NE}}_{\mathcal{A}}(T) = 1$, then $\exists t < t^*$ such that: $h_{D^*} \in \mathcal{H}_t$ and $\mathcal{V}.\mathsf{VerNonExistence}(h_{D^*}, t^*, C_{t^*}, \pi) = \mathsf{true}$. From $h_{D^*} \in \mathcal{H}_t$ and $t < t^*$, it follows that h_{D^*} existed at time t^*, and there exists a proof π' such that $\mathsf{VerExistence}(h_{D^*}, t^*, C_t, \pi') = \mathsf{true}$.

In both cases, the conclusion follows immediately.

A.1 Analysis of Unoptimized Version

Theorem 1. *If the hash function used to compute document handles is drawn uniformly at random from a family of everywhere second-preimage resistant hash functions, then the unoptimzed version of our construction is a secure time-stamping scheme.*

Proof: Let $\mathcal{A}$ be a successful adversary that outputs 1 in one of experiments $\mathsf{Exp}^{\mathsf{Ver\text{-}TS}}$ or $\mathsf{Exp}^{\mathsf{Ver\text{-}NE}}$. From Proposition 1, it follows that there exists a document D with handle h_D, a time interval $t \leq T$ and two proofs π and π' such that: $\mathsf{VerExistence}(h_D, t, C_t, \pi) = \mathsf{true}$ and $\mathsf{VerNonExistence}(h_D, t, C_t, \pi') = \mathsf{true}$.

Let us denote the nodes on the path from the root of the tree (at time t) to the leaf where h_D is stored as: $u_0 = \mathsf{DataStr}_t.\mathsf{root}, u_1, \ldots, u_w$ (w is the length of document handles).

π is an existence proof for h_D, thus it contains $w+1$ hash values. We denote $h^{\pi}_{u_0}, \ldots, h^{\pi}_{u_w}$ the hash values of the nodes on the path from the root of the tree at time t to leaf h_D, as computed from π. The last hash value $h^{\pi}_{u_w}$ is equal to h_D.

π' is a non-existence proof for h_D and it contains all the hash values of the siblings of the nodes on the path from the root to the leaf h_D, until the first null node is encountered. Denote u_{j+1} the first null node on that path. Then π' contains $j < w$ hashes, from which we could compute the hashes of the nodes $u_0, \ldots, u_j$, denoted: $h^{\pi'}_{u_0}, \ldots, h^{\pi'}_{u_j}$.

From the fact that π and π' are both correct proofs for round t, it turns out that $C_t = h(h^{\pi}_{u_0}||t) = h(h^{\pi'}_{u_0}||t)$. If $h^{\pi}_{u_0} \neq h^{\pi'}_{u_0}$, then we can output a collision for the hash function. Let us assume that $h^{\pi}_{u_0} = h^{\pi'}_{u_0}$. There are two cases that we consider:

1. $h^{\pi}_{u_j} = h^{\pi'}_{u_j}$. We know that $h^{\pi}_{u_j} = h(h^{\pi}_{u_j.\mathsf{left}}||h^{\pi}_{u_j.\mathsf{right}})$, and $h^{\pi'}_{u_j} = h(h^{\pi'}_{u_j.\mathsf{left}}||\mathsf{null})$ (or $h^{\pi'}_{u_j} = h(\mathsf{null}||h^{\pi'}_{u_j.\mathsf{right}})$). In both cases, we can construct an efficient algorithm that runs $\mathcal{A}$ and outputs pair $(h^{\pi}_{u_j.\mathsf{left}}||h^{\pi}_{u_j.\mathsf{right}}, h^{\pi'}_{u_j.\mathsf{left}}||\mathsf{null})$ (or pair $(h^{\pi}_{u_j.\mathsf{left}}||h^{\pi}_{u_j.\mathsf{right}}, \mathsf{null}||h^{\pi'}_{u_j.\mathsf{right}})$, respectively) as a collision.

2. $h^{\pi}_{u_j} \neq h^{\pi'}_{u_j}$. If we consider the two sequences $h^{\pi}_{u_0}, \ldots, h^{\pi}_{u_j}$ and $h^{\pi'}_{u_0}, \ldots, h^{\pi'}_{u_j}$, we know that $h^{\pi}_{u_0} = h^{\pi'}_{u_0}$ and $h^{\pi}_{u_j} \neq h^{\pi'}_{u_j}$. Then, there exists an index $1 \leq i < j$ such that $h^{\pi}_{u_i} = h^{\pi'}_{u_i}$ and $h^{\pi}_{u_{i+1}} \neq h^{\pi'}_{u_{i+1}}$.

Assume, wlog, that $h_D[i] = 0$. Then $h^{\pi}_{u_i} = h(h^{\pi}_{u_{i+1}}||h^{\pi}_{u_i.\mathsf{right}})$, and $h^{\pi'}_{u_i} = h(h^{\pi'}_{u_{i+1}}||h^{\pi'}_{u_i.\mathsf{right}})$. Since $h^{\pi}_{u_{i+1}} \neq h^{\pi'}_{u_{i+1}}$, we can construct an efficient algorithm that outputs pair $(h^{\pi}_{u_{i+1}}||h^{\pi}_{u_i.\mathsf{right}}, h^{\pi'}_{u_{i+1}}||h^{\pi'}_{u_i.\mathsf{right}})$ as a collision. Similarly, we can output a collision efficiently if $h_D[i] = 1$.

A.2 Analysis of Optimized Version

Theorem 2. *If the hash function used to compute document handles is drawn uniformly at random from a family of everywhere second-preimage resistant*

hash functions, then the optimzed version of our construction is a secure time-stamping scheme.

Proof: Let $\mathcal{A}$ be a successful adversary that outputs 1 in one of experiments $\mathsf{Exp}^{\mathsf{Ver\text{-}TS}}$ or $\mathsf{Exp}^{\mathsf{Ver\text{-}NE}}$. From Proposition 1, it follows that there exists a document D with handle h_D, a time interval $t \leq T$ and two proofs π and π' such that: $\mathsf{VerExistence}(h_D, t, C_t, \pi) = \mathsf{true}$ and $\mathsf{VerNonExistence}(h_D, t, C_t, \pi') = \mathsf{true}$.

π is an existence proof for h_D that contains $k+1$ hash values and k skip values ($k < w$). We denote $h^{\pi}_{u_0}, \ldots, h^{\pi}_{u_k}$ the hash values of the nodes on the path from the root of the tree at time t to leaf h_D, as computed from π. We denote $\mathsf{skip}^{\pi}_{u_1}, \ldots, \mathsf{skip}^{\pi}_{u_k}$ the skip values of these nodes (the root node u_0 does not have a skip by design). The last hash value $h^{\pi}_{u_k}$ is equal to $h(h_D||\mathsf{skip}^{\pi}_{u_k})$, and the label of node u_k is h_D.

π' is a non-existence proof for h_D that contains $m+1$ hash values and m skip values ($m < w$). We denote $h^{\pi'}_{u_0}, \ldots, h^{\pi'}_{u_m}$ the hash values of all the nodes on the path included in π', and $\mathsf{skip}^{\pi'}_{u_1}, \ldots, \mathsf{skip}^{\pi'}_{u_m}$ their skip values. Let $u_z, z \leq m$ be the last node on the path included in π' whose label is a substring of h_D.

Since both proofs are correctly formed, the value of the commitment at time t can be computed as either $h(h^{\pi}_{u_0}||t)$, or $h(h^{\pi'}_{u_0}||t)$, and thus these two hashes are equal. If $h^{\pi}_{u_0} \neq h^{\pi'}_{u_0}$, we can successfully output a collision for h. Otherwise, we assume that $h^{\pi}_{u_0} = h^{\pi'}_{u_0}$. We consider two cases:

1. u_z is a leaf node with value null. Then u_z is the last node in the proof π' and $z = m$. We can prove by induction that either, the sequences $h^{\pi}_{u_0}, \ldots, h^{\pi}_{u_m}$ and $h^{\pi'}_{u_0}, \ldots, h^{\pi'}_{u_m}$ are equal and the nodes $u_0, \ldots, u_m$ in the two proofs have the same label, or we can successfully output a collision for h.

Since u_1's label is a prefix of h_D, its position bit and label are the same in both proofs. The root of the tree does not store a skip value, thus: $h^{\pi}_{u_0} = h(h^{\pi}_{u_0.\mathsf{left}}||h^{\pi}_{u_0.\mathsf{right}})$, and $h^{\pi'}_{u_0} = h(h^{\pi'}_{u_0.\mathsf{left}}||h^{\pi'}_{u_0.\mathsf{right}})$. It follows that $h^{\pi}_{u_1} = h^{\pi'}_{u_1}$, or otherwise a collision can be output.

Similarly, assume that $h^{\pi}_{u_i} = h^{\pi'}_{u_i}$, for an i with $1 \leq i \leq m-1$, and nodes $u_0, \ldots, u_i$ have the same labels in both proofs. As the label of u_{i+1} is a prefix of h_D, it follows that its position bit is the same in both proofs. Let us assume it is 0. Then, $h^{\pi}_{u_i} = h(h^{\pi}_{u_{i+1}}||h^{\pi}_{u_i.\mathsf{right}}||\mathsf{skip}^{\pi}_{u_i})$ and $h^{\pi'}_{u_i} = h(h^{\pi'}_{u_{i+1}}||h^{\pi'}_{u_i.\mathsf{right}}||\mathsf{skip}^{\pi'}_{u_i})$. If $h^{\pi}_{u_{i+1}} \neq h^{\pi'}_{u_{i+1}}$, or $\mathsf{skip}^{\pi}_{u_i} \neq \mathsf{skip}^{\pi'}_{u_i}$, we can output a collision for h. Otherwise, it follows that $h^{\pi}_{u_{i+1}} = h^{\pi'}_{u_{i+1}}$, $\mathsf{skip}^{\pi}_{u_i} = \mathsf{skip}^{\pi'}_{u_i}$, and the labels of node u_i is the same in both proofs.

Thus, by the inductive argument $h^{\pi}_{u_m} = h^{\pi'}_{u_m}$, and the label of u_m is a substring of h_D, which implies $m < k$. It follows that $h^{\pi}_{u_m} = h(h^{\pi}_{u_m.\mathsf{left}}||h^{\pi}_{u_m.\mathsf{right}}||\mathsf{skip}^{\pi}_{u_m})$, and $h^{\pi'}_{u_m} = h(\mathsf{null}||\mathsf{skip}^{\pi'}_{u_m})$.

Since the length of null is less than that of a handle, it turns out that messages $h^{\pi}_{u_m.\mathsf{left}}||h^{\pi}_{u_m.\mathsf{right}}||$
$\mathsf{skip}^{\pi}_{u_m}$ and $\mathsf{null}||\mathsf{skip}^{\pi'}_{u_m}$ have different length, and thus they are a collision for h.

2. u_z is an internal node in the tree with label a substring of h_D, and both its children have labels incomparable to h_D in π'.

With a similar argument as in the first case we can prove inductively that the sequences $h^{\pi}_{u_0}, \ldots, h^{\pi}_{u_z}$ and $h^{\pi'}_{u_0}, \ldots, h^{\pi'}_{u_z}$ are equal and nodes $u_0, \ldots, u_z$ in the two proofs have the same label. Then $z < k$ and $z < m$.

We can compute $h^{\pi}_{u_z} = h(h^{\pi}_{u_z.\mathsf{left}} || h^{\pi}_{u_z.\mathsf{right}} || \mathsf{skip}^{\pi}_{u_z})$, and $h^{\pi'}_{u_z} = h(h^{\pi'}_{u_z.\mathsf{left}} || h^{\pi'}_{u_z.\mathsf{right}} || \mathsf{skip}^{\pi'}_{u_z})$. If $h^{\pi}_{u_z.\mathsf{left}} \neq h^{\pi'}_{u_z.\mathsf{left}}$, or $h^{\pi}_{u_z.\mathsf{right}} \neq h^{\pi'}_{u_z.\mathsf{right}}$, or $\mathsf{skip}^{\pi}_{u_z} \neq \mathsf{skip}^{\pi'}_{u_z}$, we can output a collision for h.

Let us consider the case $h^{\pi}_{u_z.\mathsf{left}} = h^{\pi'}_{u_z.\mathsf{left}}$, $h^{\pi}_{u_z.\mathsf{right}} = h^{\pi'}_{u_z.\mathsf{right}}$, and $\mathsf{skip}^{\pi}_{u_z} = \mathsf{skip}^{\pi'}_{u_z}$. From the first two hash equalities, it turns out that the skip values of $u_z.\mathsf{left}$ and $u_z.\mathsf{right}$ need to be the same to avoid a collision in h. This implies that the labels of $u_z.\mathsf{left}$ and $u_z.\mathsf{right}$ are equal in both proofs. But this contradicts the fact that both children of u_z have labels incomparable to h_D in π'.