Authentic Time-Stamps for Archival Storage

Alina Oprea and Kevin D. Bowers

RSA Laboratories, Cambridge, MA, USA
{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 confirm through an experimental evaluation with the Enron email corpus its feasibility in practice.

Key words: time-stamping, regulatory compliance, archival storage, authenticated data structures

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., [15]) 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 compromise the integrity of data stored on the disk. Sion [32] 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 [21, 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 [26], red-black trees [1, 27] or skip lists [1], and use the node duplication method proposed by Driscoll et al. [14]. 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.

Another contribution of this paper is giving rigorous security definitions for time-stamping schemes that offer document authenticity against a powerful inside attacker. Our constructions are proven secure under the new security definitions. Finally, we confirm the efficiency of our optimized construction through a Java implementation and an evaluation on the Enron email data set [12].

Organization. We start in Section 2 by reviewing related literature. We present our security model in Section 3 and our constructions in Section 4. We give the performance evaluation of our implementation in Section 5, and conclude in Section 6. Full details of the persistent data structure and a complete security analysis of our constructions can be found in the full version of the paper [31].

2 Related Work

In response to the increasing number of regulations mandating secure retention of data, *compliance storage* (e.g., [15, 22]) has been proposed. 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, and, as such are vulnerable to inside attackers with full access privileges and physical access to the disks. Sion [32] proposes to secure WORM storage with active tamper-resistant hardware.

A method proposed in the early 90s to authenticate the content and creation time of documents leverages time-stamping services [21, 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 reduce the amount of trust in such services, techniques such as linking [21, 3, 2], accountability [8, 6, 10, 5], periodic auditing [9], and "timeline entanglement" [27] have been proposed. Time-stamping schemes are useful in preventing back-dating and establishing the relative ordering of documents, but they do 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. Our goal is to provide scalable methods to authenticate the content and creation time of documents archived for compliance requirements.

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. First ADs based on hash trees were proposed for certificate revocation [24, 30, 7]. ADs based on either skip lists [18, 20, 17] or red-black trees [1] have been proposed subsequently. There exist other constructions of ADs with different efficiency tradeoffs that do not support non-membership proofs, e.g., based on dynamic accumulators [11, 19] 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. First PADs were based on red-black trees and skip lists [1], and use the node duplication method of Driscoll et al. [14]. Goodrich et al. [16] 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 [26] is a system designed for archiving of signed documents, ensuring that signatures can be verified even after key revocation. Timeline entanglement [27] is a technique that leverages multiple time-stamping services for eliminating trust in a single service. CATS [33] 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 data structures based on Merkle trees, called *history trees*, has been developed recently by Crosby and

Wallach [13] 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 [29, 25]. Micali et al. [29] 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 documents 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, $\mathsf{GetTimestamp}$, $\mathsf{GetAllDocs}$, $\mathsf{GenCommit}$, $\mathsf{GenProofExistence}$, $\mathsf{GenProofNonExistence}$ available to $\mathcal{S}$, and algorithms $\mathsf{VerExistence}$ and $\mathsf{VerNonExistence}$ available to $\mathcal{V}$. Some of these algorithms implicitly call the trusted party $\mathcal{T}$ for storing and retrieving commitments for particular time intervals.

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.

$\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Ver-TS}}(T)$:	$\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Ver-NE}}(T)$:
$s \leftarrow \lambda$	$s \leftarrow \lambda$
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 $\mathsf{Exp}^{\mathsf{Ver-NE}}$).

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.

4.1 Merkle Trees

Merkle trees [28] 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 [23] 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 $u1$. This can be easily generalized to trees of higher degree, and we explore such tries constructed from arbitrary degree trees further in our implementation.

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 based on a binary tree 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 when implemented with a binary tree.

For more efficient insert and search operations, Patricia trees [23] 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.

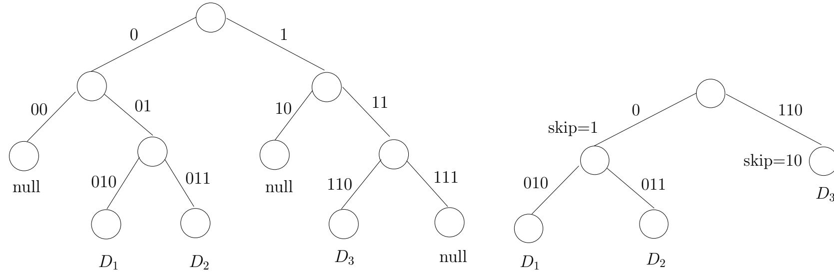

Fig. 2. Unoptimized trie for strings $D_1 = 010$, $D_2 = 011$ and $D_3 = 110$.

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

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.

Knuth [23] 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 stored in an array ordered by time intervals. In the optimized version, the hashes at each node are computed over the skip value of the node, in addition to the children's hashes (for an internal node), and the document handle (for a leaf node).

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 version t hashes of the sibling 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. In addition, in the optimized version, the proof includes skip values of all nodes on the path from the leaf to the root, and the auditor needs to check that the position of the document handle in the tree is correct, using the skip values sent in the proof.

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. Again, in the optimized version, skip values on the search path are included in the proof so that the auditor could determine if the tree is correctly constructed.

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.

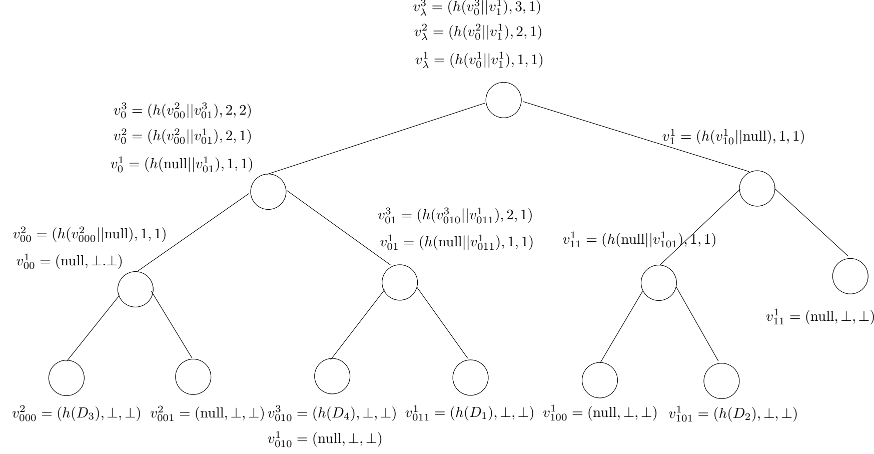

Fig. 4. Tree at interval 3.

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.

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.

For lack of space, we omit full details of our data structure, and refer the reader to the full version of our paper for complete algorithm description and security analysis [31].

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 for its implementation is to include a proof of document's existence at a time t and its non-existence at all previous time intervals $1, \ldots, t-1$. To reduce the complexity, probabilistic proofs can be used in which the server provides non-existence proofs only for a set of intervals chosen pseudorandomly by the auditor.

4.4 Efficiency

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 [33]. 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.

	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, 33]	$\log n_t$ 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.

All previously proposed persistent authenticated dictionaries we are aware of use the node duplication method of Driscoll et al. [14] 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.

	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, 33]	$\log n_t$	$O(n_t \log n_t)$	$(\log n_t)\|h\|$	$2(\log n_t)\|h\|$

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.

5 Experimental Evaluation

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

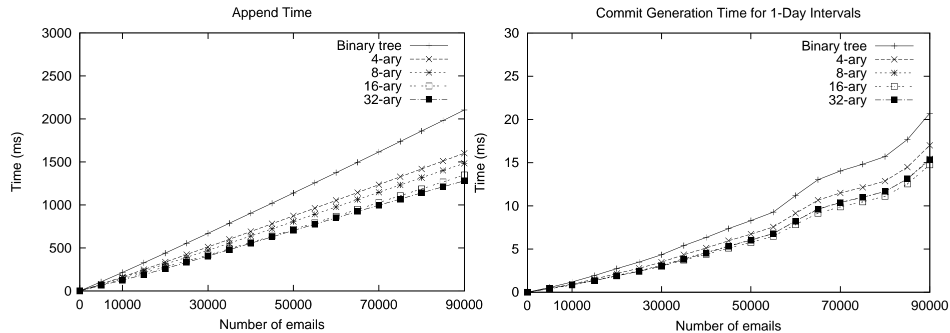

Fig. 5. Performance of Append and GenCommit operations.

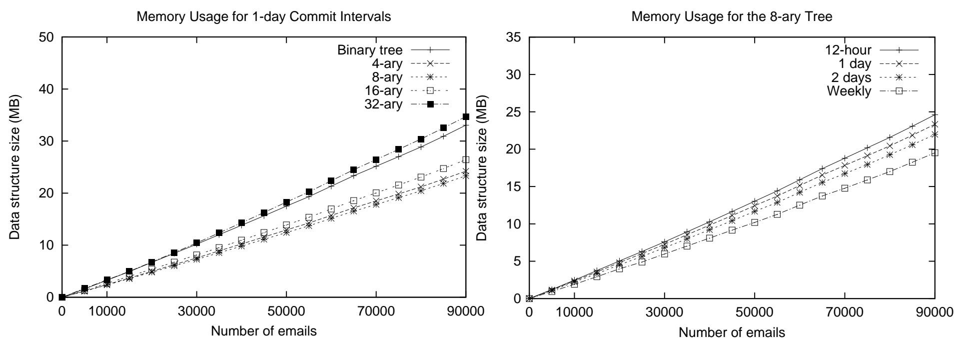

Fig. 6. Data structure storage requirements.

some experiments using the Enron email data set [12]. 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.

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 5 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 5 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 6 shows the total size of the data structure 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 6. 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 7, 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 7, 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

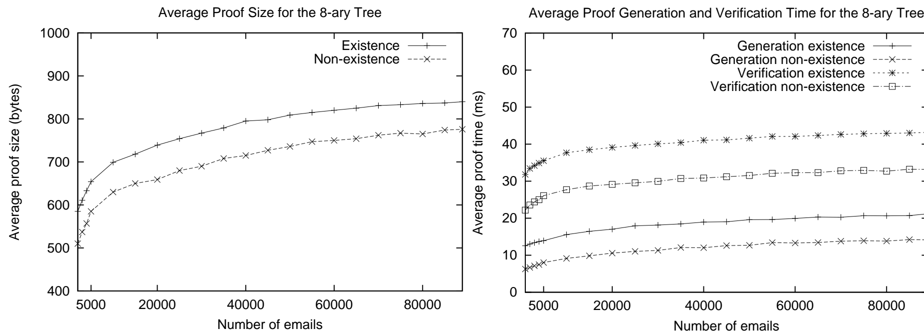

Fig. 7. Proof size and proof generation and verification time for an eight-ary tree.

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 correctly commits to all received documents. For our unoptimized data structure, auditing could be performed with a mechanism similar to the incremental proofs from [13]. 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 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. 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.
7. 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, ACM, 2000.
8. 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.
9. 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.
10. 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.
11. 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.
12. W. Cohen, "Enron email dataset." `http://www.cs.cmu.edu/~enron`.
13. S. Crosby and D. Wallach, "Efficient data structures for tamper evident logging," in *Proc. 18th USENIX Security Symposium*, USENIX, 2009.
14. 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.
15. EMC, "Centera Compliance Edition Plus." `http://www.emc.com/products/detail/hardware/centera-compliance-edition-plus.htm`.
16. 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.

17. 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.
18. 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`.
19. 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.
20. 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.
21. S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, 1991.
22. 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)*, IEEE Computer Society, 2006.
23. D. E. Knuth, *The art of computer programming*, vol. 3, Sorting and Searching. Addison-Wesley, 1973. Second Edition.
24. P. Kocher, "On certificate revocation and validation," in *Financial Cryptography*, vol. 1465 of *LNCS*, pp. 951–980, Springer-Verlag, 1998.
25. 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.
26. 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, USENIX, 2002.
27. P. Maniatis and M. Baker, "Secure history preservation through timeline entanglement," in *Proc. 11th USENIX Security Symposium*, pp. 297–312, USENIX, 2002.
28. R. Merkle, "A cerified digital signature," in *Proc. Crypto 1989*, vol. 435 of *LNCS*, pp. 218–238, Springer-Verlag, 1989.
29. 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.
30. M. Naor and K. Nissim, "Certificate revocation and certificate update," in *Proc. 7th USENIX Security Symposium*, USENIX, 1998.
31. A. Oprea and K. Bowers, "Authentic time-stamps for archival storage," 2009. Available from the Cryptology ePrint Archive.
32. R. Sion, "Strong WORM," in *Proc. of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS)*, IEEE Computer Society, 2008.
33. A. Yumerefendi and J. Chase, "Strong accountability for network storage," in *Proc. 6th USENIX Conference on File and Storage Technologies (FAST)*, USENIX, 2007.