# Athos: Efficient Authentication of Outsourced File Systems*

Michael T. Goodrich[1], Charalampos Papamanthou[2], Roberto Tamassia[2], and Nikos Triandopoulos[3]

[1] Dept. of Computer Science, U. California, Irvine, USA
goodrich@ics.uci.edu
[2] Dept. of Computer Science, Brown University, USA
{cpap,rt}@cs.brown.edu
[3] Dept. of Computer Science, University of Aarhus, Denmark
nikos@daimi.au.dk

**Abstract.** We study the problem of authenticated storage, where we wish to construct protocols that allow to outsource any complex file system to an untrusted server and yet ensure the file-system's integrity. We introduce *Athos*, a new, platform-independent and user-transparent architecture for authenticated outsourced storage. Using light-weight cryptographic primitives and efficient data-structuring techniques, we design authentication schemes that allow a client to efficiently verify that the file system is fully consistent with the exact history of updates and queries requested by the client. In Athos, file-system operations are verified in time that is logarithmic in the size of the file system using optimal storage complexity—constant storage overhead at the client and asymptotically no extra overhead at the server. We provide a prototype implementation of Athos validating its performance and its authentication capabilities.

## 1 Introduction

Current trends in the design of data-storage systems are towards decentralized and networked architectures where data resides "in the cloud", outside any administrative control, and is being manipulated in storage units of minimal trust assumptions (e.g., NAS or SAN, storage providers, Internet-based computing). Operating on remotely managed data inherently entails security risks: when the storage provider is not trusted by the data source, verifying the integrity of the stored data and the correctness of the computations performed on this data is necessary to ensure the trustworthiness of the storage system; and verifying complex operations over general file systems efficiently is rather challenging.

* Research supported in part by the U.S. National Science Foundation under grants IIS–0713403, IIS-0713046, CNS-0312760 and OCI–0724806, the I3P Institute under a U.S. DHS award, the Center for Algorithmic Game Theory at the University of Aarhus under an award from the Carlsberg Foundation, the Center for Geometric Computing and the Kanellakis Fellowship at Brown University, and IAM Technology, Inc. The views in this paper do not necessarily reflect the views of the sponsors.

T.-C. Wu et al. (Eds.): ISC 2008, LNCS 5222, pp. 80–96, 2008.

In this paper, we study the problem of authenticating the integrity and operational correctness of a file system that is outsourced by a client to an untrusted server. We assume that the remote server's host machine and its storage units can behave maliciously. We wish to design authentication protocols that allow the client to efficiently verify the integrity of a dynamically evolving file system, namely to verify that its status is consistent with the exact history of file-system operations requested by the client, and to correctly detect any malicious data-update or data-retrieval patterns produced by the server. To conform to the outsourced data model, we require that the authentication protocols incur constant storage overhead at the client and asymptotically no extra storage costs at the server—otherwise, the client has no reason to outsource its data, at the first place—and also that verification is achieved efficiently, in time that is sublinear or logarithmic in the file system's size—or else, the client could trivially download the entire signed (and timestamped) file-system data on every operation.

**Goals and Assumptions.** Using cryptographic hashing is the state-of-the-art solution for verifying the integrity of simple put-get operations over a collection of files in the outsourced data model: the client locally keeps the hash of each file against which file retrievals or updates can be verified in constant time. The use of Merkle's tree [18] can reduce the client's space from linear to constant: the client only stores the root hash and both file retrievals and file updates (e.g., using existing techniques [3, 28]), can be verified in logarithmic time. Unfortunately, applying this approach in our setting provides only a partial solution: file-system integrity requires not only data integrity at the file or data-block level, but also integrity of the *directory hierarchy* of the file system. Indeed, all file-system operations are defined with respect to the directory path, and in many cases, the integrity of a file depends not only on its content, but also on its location in the file system. For example, the context of an .htaccess file depends on its location—its contents identify access policies, but its location is critical to identify the directories it protects. Our goal, therefore, is primarily to efficiently verify the directory hierarchy of the file system, and through this, any file-system or metadata operation that depends on this hierarchy. Of course, applying hashing over the directory tree (e.g., as in [7]), possibly augmented by the (balanced) hash trees that correspond to large files lying in the directory tree, can provide a space-optimal solution. However, this approach incurs linear verification and update costs, for the directory tree is unlikely to be balanced! Our goal is to design dynamic authentication schemes that overcome this problem.

Another solution is to have the client authenticate each file-system update it makes in the outsourced file system (e.g., using a signature or HMAC based on a private key), which has some major drawbacks, however. First, it allows for replay attacks, since determining the freshness of signed statements is difficult with such a scheme. Second, this solution requires the client to sign every possible path in the directory hierarchy in order to be able to authenticate locations. This can be especially inefficient, for example when the client performs a directory operation that moves a large directory to a new location. Another possibility is to assume that the outsourced file system is partially trustworthy (e.g., [21])

or a part of its architecture uses some tamper-resistant trusted hardware (e.g., using trusted computing platforms [27]). This assumption postulates that the networked file system is itself at least partially trusted, which is not that much different than simply trusting the hosting server in the first place. As we show in this paper, such trust is not necessary for the sake of efficiency or reliability.

In this work, we consider the outsourced data authentication problem in the single-client setting. However, in a multi-client setting, the problem of outsourced data authentication has drastically different characteristics. If communication between the clients is not allowed, a malicious server can easily perform an attack against data consistency: the server can effectively hide the most recent update on the data from a client requesting to read the data, by unrolling its state to the state the server had just before that update took place. Undetected without communication, this attack can be used to "fork" the view that this client has about the outsourced data, harming the consistency of the system. In this scenario, the best one can hope for is *fork-consistency* [16], effectively disallowing anything more than the forking attack, and various schemes securely achieve this property (e.g., [16, 20, 15, 4]). However, in the single-client setting the forking attack can be detected and prevented (e.g., by the hash-based solution), therefore, the fork-consistency property is no longer relevant in this setting. Although less general, the single-client model has its own merits. First, it naturally captures the security problems for a wide application area, where a single user outsources a personal file system to a storage provider. Second, in certain applications the multi-user setting is easily reduced to the single-client setting; for instance, in a networked file system, the client can simply abstract the OS kernel or a designated filer machine through which all file-system operation requests coming from many users are serialized to the untrusted remote storage devices. Third, in applications that can tolerate reasonable delays in the response time, and under reasonable assumptions about the availability of a *constant-size* shared trusted storage, the multi-client setting is also reduced to the single-client setting, achieving a stronger property than fork-consistency: conceptually the shared memory replaces the communication between parties.

**Related Work.** Previous work makes use of cryptographic hashing or signatures for primarily protecting the integrity of individual files or the corresponding data blocks that reside at storage units. Most of the systems (e.g., [5, 2, 9, 19]) provide file integrity using authentication information at the client that is proportional to the size $n$ of the file system (i.e., the number of files or corresponding data blocks). More efficient constructions involve the use of Merkle trees [18] over the data blocks of individual files (e.g., [6, 24, 14, 21, 15]) or over the blocks or files of the entire file system (e.g., [8, 31]). Beyond hashing and signing, other space-efficient techniques have been proposed for file integrity, such as an entropy-based integrity method for encrypted (only) files [22] and a scheme based on the Galois counter mode [17], where however updates take linear time. Some constructions do authenticate the directory hierarchy or related meta-data of the file system, but, by hashing over the directory tree or signing each individual object, they result in linear update costs (e.g., [13, 7, 10]), or only support

verification of a limited set of operations (e.g., [15, 7, 10, 14]). Other schemes, additionally assume the existence of a trusted component at the untrusted server (e.g., [21, 25, 30], or some external trusted party (e.g., [31]) to authenticate file operations. Finally, SUNDR [15] and [16, 20, 4] use hashing and signatures to provide file-system integrity and fork-consistency in the multi-client setting; solving a harder problem, these schemes have increased performance costs.

**Our Contributions.** We present the design and a prototype implementation of an authentication architecture, which we call *Athos* (AuTHenticated Outsourced Storage), that supports an *authenticated outsourced file system* in the client-server model. We construct protocols for authenticating a rich set of file-system operations that are requested by the client and performed by the untrusted server. Our protocols support verification of the file system's full functionality by efficiently providing not only integrity of the stored data, but also integrity of the file-system directory structure. Security in our model corresponds to the natural notion of *consistency* in the single-client setting: at all times, the interaction with the server over any series of file-system operations should give the client the same view as the one obtained by a trusted server (as if the file system was never outsourced), and any deviation should be immediately detected. To achieve this, the client maintains only a hash digest of the file system, against which the validity of each operation performed by the server can be verified, using small proofs. These proofs are generated by an *authentication service* module that uses an authentication data structure stored in the server's untrusted memory, and runs in parallel with the file-system management module, and they consist of partial file system meta-data and hashes stored in the authentication data structure. This data structure defines the file-system digest, in a hash-tree fashion.

To achieve our efficiency goals, we use ideas from the domain of data authentication, employing efficient data structuring techniques for representing an entire file system. The challenge is to efficiently authenticate the directory hierarchy, which is typically highly unbalanced. We contribute two concrete authentication structures: Our first construction is based on a novel mapping of the directory hierarchy to a set of relations, and the authentication of put-get operations on this set using a skip list as the underlying authentication data structure. This approach achieves simplicity and low-cost authentication, and also leverages all the benefits of the widely researched authenticated dictionaries (e.g., [11]). Our second construction is of more theoretical interest, providing an optimal authentication scheme based on dynamic trees, a classical data structuring technique for operating on unbalanced trees in a balanced way. Overall, Athos achieves optimal storage usage (constant for the client and linear for the server) and efficient integrity verification (logarithmic or sublinear depending on the operation) and achieves generality by being agnostic of the specific implementation of the networked file system and by being platform-independent. Finally, a prototype implementation of Athos and an experimental evaluation of its verification capabilities for real-life file systems confirm our theoretical analysis.

Section 2 overviews our authentication model and Section 3 describes our authentication schemes. Section 4 presents the experimental evaluation of Athos

and discusses related issues. Section 5 presents our concluding remarks. Details on our construction that is based on dynamic trees and our experimental evaluation can be found in the Appendix. This extended abstract omits complete proofs and other details that will appear in the full version of the paper.

## 2 Model and Definitions

We study storage authentication in the following model (see also Figure 1). A client $\mathcal{C}$ owns and (incrementally) outsources a file system $FS$ to an untrusted server $\mathcal{S}$. In additional to the file system, $\mathcal{S}$ hosts and controls an *authentication service* module $\mathcal{A}$ that stores authentication information about $FS$. The file system is generated and queried through a series of *update* and *query* operations issued by the client $\mathcal{C}$. At any time, $\mathcal{C}$ keeps some *state information* $s$ that encodes information about the current state of $FS$. If $\mathcal{P}$ is the set of operations supported over the file system, then the communication protocol is as follows:

1. Client $\mathcal{C}$ keeps state information $s$ and issues a query or update operation $o \in \mathcal{P}$ to the server $\mathcal{S}$.
2. Server $\mathcal{S}$ runs a certification algorithm, which performs operation $o$ and accordingly answers the query or updates $FS$ to a new version $FS'$, and, by using $\mathcal{A}$, also generates a *verification* or respectively *consistency proof* $\pi$ which is returned to $\mathcal{C}$, along with the result $\rho$ of the operation; $\rho$ is the corresponding answer if $o$ is a query operation or else the empty string $\perp$. We write $\pi \leftarrow \mathsf{certify}(o, FS, FS', \rho)$.
3. Client $\mathcal{C}$ runs a verification algorithm, which takes as input the current state $s$, the operation $o$ along with its result $\rho$, and the corresponding (consistency or verification) proof $\pi$ and either accepts or rejects the input. If the input is accepted the state $s$ is appropriately updated to state $s'$, where $s' = s$ if $o$ is a query operation or else $s' \neq s$. We write $\{(yes, s'), (no, \perp)\} \leftarrow \mathsf{verify}(s, \rho, \pi)$.

We call the pair of algorithms ($\mathsf{certify}$, $\mathsf{verify}$) an *authenticated storage scheme*.[1] The security property we wish such a scheme to satisfy expresses the intuitive requirement that the verification performed at $\mathcal{C}$ must be a reliable test for checking the file system's integrity. Let $\mathsf{operate}(\cdot, \cdot)$ be the algorithm that, given the current file system $FS$ and an operation $o \in \mathcal{P}$, performs $o$ and updates $FS$ to $FS'$. We write $(FS', \rho) \leftarrow \mathsf{operate}(o, FS)$ ($\rho = \perp$ for updates and $FS' = FS$ for queries). We say that state $s$ is *consistent* with $FS_\tau$ for a series $\tau$ of operations on $FS$, if $s$ and $FS_\tau$ have been computed by running algorithms $\mathsf{operate}$, $\mathsf{certify}$ and $\mathsf{verify}$ sequentially for all operations in series $\tau$ starting from $FS$.

**Definition 1 (Security of authenticated storage schemes.).** *We say that an authenticated storage scheme* ($\mathsf{certify}, \mathsf{verify}$) *(with security parameter $\kappa$) is* secure*, if for any series of operations $\tau$ and a state $s$ that is consistent with file system $FS_\tau$ for $\tau$ on an initially empty file system, the following conditions hold:*

[1] Both algorithms take as input also a public key that is known by both $\mathcal{C}$ and $\mathcal{S}$.

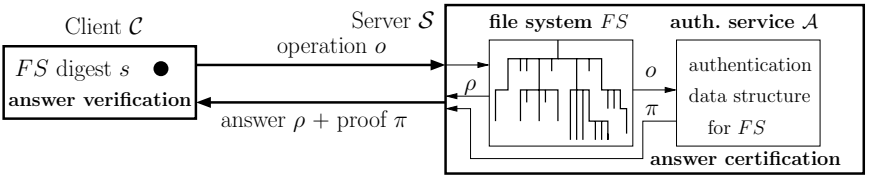



**Fig. 1.** The authenticated data storage model. Keeping only constant-size state $s$, client $\mathcal{C}$ remotely manages a file system $FS$ that resides at untrusted server $\mathcal{S}$. Every query or update operation $o$ requested by $\mathcal{C}$ on $FS$ is certified by $\mathcal{S}$, using an authentication service module $\mathcal{A}$ (that stores authentication information related to $FS$) to produce a verification or consistency proof $\pi$; this proof is used by $\mathcal{C}$, along with the result $\rho$ of the operation, to verify that the request was handled consistently, and finally update $s$.

**Correctness.** *For any $o \in \mathcal{P}$, when $(FS'_\tau, \rho) \leftarrow \mathsf{operate}(o, FS_\tau)$, it holds that $(yes, s') \leftarrow \mathsf{verify}(s, \rho, \mathsf{certify}(o, FS_\tau, FS'_\tau, \rho))$. I.e., for any correctly performed operation,* certify *generates a proof that is always accepted by* verify*, which also computes a new, consistent with the new file system $FS'_\tau$, state $s'$.*

**Consistency.** *For any series $\tau$ of operations and new operation $o$, such that state $s$ is consistent with file system $FS_\tau$ for $\tau$ on an initially empty file system and $(FS'_\tau, \rho) \leftarrow \mathsf{operate}(o, FS_\tau)$, then for any polynomial-time adversary, controlling $\mathcal{S}$ and having oracle-access to algorithm* verify*, that on input the file system $FS_\tau$, series $\tau$ and operation $o$, produces proof $\pi'$ and result $\rho'$, whenever $(yes, s') \leftarrow \mathsf{verify}(s, \rho', \pi')$, then the probability that either $\rho' \neq \rho$ or $s'$ is not consistent with $FS'_\tau$ for operation $o$ on $FS_\tau$ is negligible (in the security parameter $\kappa$). I.e., assuming a polynomially bounded adversary that observes polynomially many protocol invocations and then produces a pair of proof $\pi'$ and result $\rho'$, if $\rho'$ and $\pi'$ for the new operation $o$ are accepted by* verify*, then for all but negligible probability the operation has been performed correctly and the new state is consistent with the new file system.*

Starting from an initially empty set and using a secure authenticated storage scheme and the appropriate series of updates, client $\mathcal{C}$ is able to "export" *any* file system to server $\mathcal{S}$, such that $\mathcal{C}$ has a consistent state with the current file system. Therefore, the file system is consistent with the history of updates and all future operations will be verified. With respect to efficiency, we say that an authenticated storage scheme is time-*efficient* if the verification time at $\mathcal{C}$ is sublinear in the file-system size, and space-*optimal* if $\mathcal{C}$ stores state of constant size. We next exhibit *time-efficient*, *space-optimal* and *secure* authenticated storage schemes for a rich set of operations on an outsourced file system.

## 3 Efficient Authenticated Storage

To give some intuition of our general approach, let us consider the special case where we want to implement an *authenticated map* in the client-server outsourced

storage model; actually, this authentication functionality on the map data structure will also be our core authentication tool for verifying file system operations. Each entry of the map is a tuple $(k, v)$, where $v$ is a value corresponding to a key $k$ ($v$ can be a collection of objects). The entries of the map are sorted according to their keys (using some comparator). The authenticated map data structure resides at the server. Using a *hashing scheme* for skip lists, i.e., a hierarchical way to produce a hash digest by recursively applying a cryptographic hash function $h$ over some data (see, e.g., [11]), we can define a digest of the authenticated map, computed according to the skip-list tree structure (Figure 2(a)).

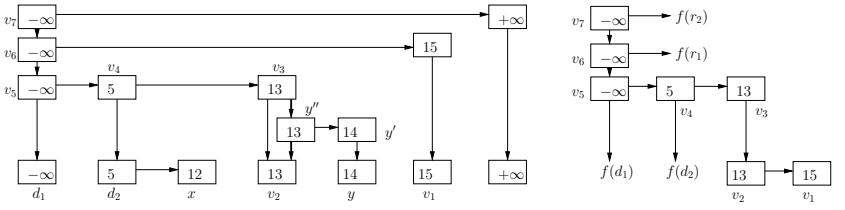



**Fig. 2.** (a) The skip list hashing scheme for verifying operations on a map data structure and insertion of key 14. (b) The consistency proof $P$ returned by $\mathcal{S}$, containing all the hashing and structural information needed to verify the consistency of $P$ subject to the current digest and to locally perform the update.

Let $d_0$ be the initial digest stored at the client $\mathcal{C}$ which is consistent with the current state of the skip list. Suppose now that $\mathcal{C}$ wants to insert a new key $x$. The server $\mathcal{S}$ returns to $\mathcal{C}$ a consistency proof that consists of the search path $P$ in the unsuccessful search for $x$ before the update (Figure 2(b)). Path $P$ is related with the key insertion, satisfying the following properties, which in turn imply the security of the scheme. First, $P$ contains the two keys, say $\mathsf{succ}(x)$ and $\mathsf{pred}(x)$, that are the successor and predecessor of $x$ in the ordering of the keys, and also contains all the necessary *hashing* information (hash values) that allows $\mathcal{C}$ to recompute the current digest $d_0$ starting from $\mathsf{succ}(x)$ and $\mathsf{pred}(x)$ and hashing according to the hashing scheme that is used. Due to the collision-resistance property of the hash function, $\mathcal{C}$ can check if the received path is the correct one, and if $P$ is verified, $\mathcal{C}$ verifies that key $x$ is not in the directory. Also, $\mathcal{C}$ knows the position at which this file should be added. Second, $P$ contains all the necessary *structural* information that allows $\mathcal{C}$ to locally perform the update in the hashing scheme that corresponds after the file insertion, by placing $x$ between $\mathsf{succ}(x)$ and $\mathsf{pred}(x)$ and computing the new hash values for only those nodes of the skip list that need a new hash. Knowing the new hash values, $\mathcal{C}$ can compute the new digest $d_0'$, which is consistent with the update. Thus, the key insertion (performed by $\mathcal{S}$) can be verified in two steps: first, path $P$ is verified and then it is used to locally perform the update and compute the new digest. Using results on authenticated skip lists (e.g., [23, 11]) we have the following:

**Lemma 1.** *There exists an authenticated storage scheme for operations on n key-value pairs in a map that is based on an authenticated skip list, with the*

*following expected complexity bounds: (i) The expected update (insertion and removal), query and verification time is $O(\log n)$ w.h.p.; (ii) The expected size of the consistency and verification proof (communication cost) is $O(\log n)$ w.h.p.*

Here, *update time* is the time required by $\mathcal{S}$ to do the actual update, *query time* is the time $\mathcal{S}$ needs to compute the (consistency or verification) proof, *verification time* is the time that $\mathcal{C}$ needs in order to process the proof and validate or reject the query or the update. Note that for *set-membership queries and updates* (through which we are going to implement all file system operations) the size of a proof is always asymptotically equal to the verification time; therefore, the verification time bounds will indirectly imply the size of the proof.

We next use the authenticated-map functionality to verify more complex operations on general file systems. Let $T$ be the tree that corresponds to the hierarchy induced by the structure of the directories and files of a file system, where the left-right ordering of sibling nodes coincides with the chronological order of the node creations. The idea is to carefully map $T$'s structural information to a set of special entries and store this set in an authenticated map, in a way that allows to authenticate the integrity of the entire file system. Node $v$ in $T$ (a directory or file) defines an authenticated-map entry that stores, under key $\mathsf{key}(v)$ that is the corresponding i-node in the file-system, the following fields:

- $\mathsf{name}$: the actual name of the node of the file system;
- $\mathsf{file}$: a hash (e.g., SHA-1) of the file represented by $v$ ($\mathsf{null}$ for a directory);
- $\mathsf{key}(\mathsf{parent})$: the key of the entry corresponding to the parent node of $v$;
- $\mathsf{key}(\mathsf{sibling})$: the key of the entry corresponding to the successor sibling of $v$ in $T$ ($\mathsf{null}$ if $v$ is the last created node of the children list);
- $\mathsf{key}(\mathsf{backsibling})$: the key of the entry corresponding to the predecessor sibling of $v$ in $T$ ($\mathsf{null}$ if $v$ is the first created node of the children list);
- $\mathsf{key}(\mathsf{child})$: the key of the entry corresponding to the first created child of $v$.

We next map each file-system query or update to a small set of (regular) query or update operations in the authenticated map, effectively reducing file-system operations to set-membership operations. We have the following:

**Theorem 1.** *Assuming the existence of collision-resistant hash functions, there exists a secure and space-optimal authenticated storage scheme that is implemented with skip lists, achieving the following performance, where $n$ is the size of the file-system tree $T$, $T_v$ is the subtree rooted on node $v$, $\ell_v$ is the number of children of node $v$ and $\Pi = \pi_1\pi_2\ldots\pi_k$ is a path in $T$: (1) The authentication of any path $\Pi$ takes $t(\Pi) = O(k \log n)$ query and verification time; (2) Query operations $\mathsf{cd}(\Pi)$, $\mathsf{read}(\Pi)$ and update operations, $\mathsf{write}(\Pi)$, $\mathsf{rm}(\Pi)$, $\mathsf{mkdir}(\Pi)$, $\mathsf{touch}(\Pi)$ take $t(\Pi)$ query, verification and update, query, verification time respectively; (3) Query operation $\mathsf{ls}(\Pi)$ takes $t(\Pi) + O(\ell_{\pi_k} \log n)$ query and verification time; (4) Update operation $\mathsf{rmdir}(\Pi)$ takes $t(\Pi) + O(|T_{\pi_k}| \log n)$ update, query and verification time; (5) Update operation $\mathsf{mv}(\Pi, \Pi')$ takes $t(\Pi) + t(\Pi')$ update, query and verification time.*

We now discuss another possible method of representing the file system using a skip list. Instead of setting the i-node number as node's $v$ key, we can set as $\mathsf{key}(v)$

**Table 1.** Efficiency comparison of our authenticated storage schemes w.r.t. the query, update and verification times, using skip lists (local and global approaches) and dynamic trees. Here, $n$ is the size of the file system, $\Pi = \pi_1\pi_2\ldots\pi_k$ is the directory argument, $\ell$ is the size of the children list and $T$ is the subtree rooted on $\pi_k$.

| operation | skip list (local) | skip list (global) | dynamic tree |
|---|---|---|---|
| $\mathsf{cd}(\Pi)$, $\mathsf{touch}(\Pi)$, $\mathsf{read}(\Pi)$ $\mathsf{write}(\Pi)$, $\mathsf{rm}(\Pi)$, $\mathsf{mkdir}(\Pi)$ | $O(k \log n)$ | $O(\log n + k)$ | $O(\log n + k)$ |
| $\mathsf{ls}(\Pi)$ | $O((k+\ell)\log n)$ | $O(\ell(\log n + k))$ | $O(k + \ell + \log n)$ |
| $\mathsf{rmdir}(\Pi)$ | $O((k+\lvert T\rvert)\log n)$ | $O(\lvert T\rvert \log n + k)$ | $O(k + \log n)$ |
| $\mathsf{mv}(\Pi, \Pi')$ | $O((k+k')\log n)$ | $O(\lvert T\rvert \log n + k + k')$ | $O(k + k' + \log n)$ |

the name of the path from the file-system root to node $v$ (e.g., the key for file pub.txt lying in path /users/user/ is now the string "/users/user/pub.txt"). Thus, in the previous representation we stored "local" information, whereas now we rather store "global" information. This solution yields better complexity bounds for the path authentication (which is now $t(\Pi) = O(\log n + |\Pi|)$). However, update operation $\mathsf{mv}(\Pi, \Pi')$ takes $O\left(t(\Pi) + t(\Pi') + |T| \log n\right)$ update, query, and verification time, where $T$ is the subtree rooted at $\pi_{|\Pi|}$. This representation is suitable for cases where the majority of the operations are file system navigations and move operations are less frequent. Finally, by using authenticated path operations [12] implemented with dynamic trees [26], we get the following:

**Theorem 2.** *Assuming the existence of collision-resistant hash functions, there exists a secure, time-efficient and space-optimal authenticated storage scheme that is implemented with dynamic trees, achieving the following performance, where $n$ is the size of the file system tree and $\ell_v$ is the number of children of $v$: (1) The authentication of any path $\Pi$ takes $t(\Pi) = O(k + \log n)$ query and verification time; (2) Query operations $\mathsf{cd}(\Pi)$, $\mathsf{read}(\Pi)$ and update operations $\mathsf{write}(\Pi)$, $\mathsf{rm}(\Pi)$, $\mathsf{mkdir}(\Pi)$, $\mathsf{touch}(\Pi)$ take $t(\Pi)$ query, verification and update, query, verification time respectively; (3) Query operation $\mathsf{ls}(\Pi)$ takes $t(\Pi) + O(\ell_{\pi_k} + \log n)$ query and verification time; (4) Update operation $\mathsf{rmdir}(\Pi)$ takes $t(\Pi)$ update, query and verification time; (5) Update operation $\mathsf{mv}(\Pi, \Pi')$ takes $t(\Pi) + t(\Pi') + O(\log n)$ update, query and verification time.*

A more detailed description of this scheme appears in the Appendix. Table 1 presents a comparison between the efficiency levels achieved by our schemes.

**Security.** Our authenticated storage schemes are based on the following general approach. Given a secure hashing scheme $\mathcal{H}$ for a specific query type $\mathcal{Q}$, that is, a directed acyclic graph that defines how a hash digest is computed from a data set and a corresponding authentication structure,[2] we augment $\mathcal{H}$ to a new hashing scheme $\mathcal{H}'$ that additionally encodes (in its produced digest) the entire structural and balancing information that is defined in the underlying authentication

[2] Against this (authentic) digest answers to queries in $\mathcal{Q}$ can be efficiently verified; this is the general verification technique used by authenticated data structures.

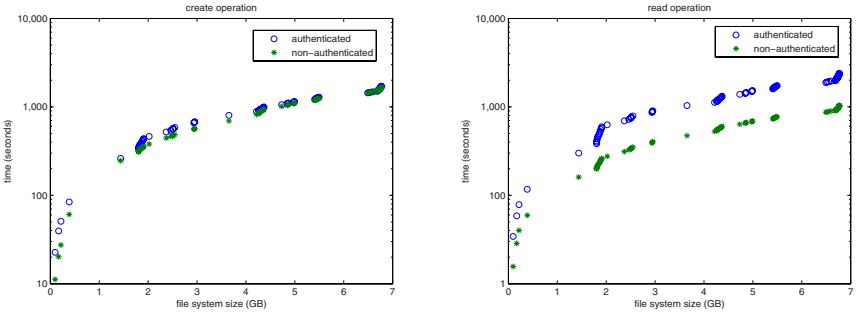
structure. In particular, if the hash value $h_v$ of node $v$ in the data structure is computed as $h_v = h(h_{u_1}, \ldots, h_{u_k})$ in $\mathcal{H}$, we define $h_v = h(h_{u_1}, \ldots, h_{u_k}, h(b_v, s_v))$ in $\mathcal{H}'$, where $b_v$, $s_v$ describe the balancing and respectively structural information about node $v$. In our constructions, we make use of the hashing schemes corresponding to the skip list and the dynamic-tree data structure for efficiently verifying set-membership [11] and respectively path property [12] queries.

Given the augmented hashing trees, security is proved as follows. Starting from the state corresponding to the empty file system, we inductively show that after any update on the file system the client $\mathcal{C}$ updates its state $s$ consistently for the new update on the currently existing file system $FS$. For both data structures used in our schemes, the consistency proof by the definition of the corresponding augmented hashing scheme $\mathcal{H}'$ contains all the balancing and structural information that completely characterizes the changes in $FS$ due to the update. Assuming that the state is consistent, the consistency proof coming from an honest server $\mathcal{S}$ will be verified, thus also the balancing and structural information related to the update. Thus, $\mathcal{C}$ is able to locally perform the correct update as if $\mathcal{C}$ had direct access to the entire file system $FS$, thus is able to correctly and consistently update his state $s$ to $s'$, which is simply the new digest according to $\mathcal{H}'$. Given this invariant, any query is securely verified since the underlying hashing scheme is secure: assuming that finding hash collisions is computationally hard, any malicious behavior by $\mathcal{S}$ will be rejected by the verification algorithm, since any undetected inconsistency corresponds to a collision.

## 4 Analysis, Experiments and Discussion

We have developed a prototype implementation of Athos using skip lists. Our implementation uses a flat representation of the file system tree since this representation outperforms dynamic trees when the depth of the tree is less than 200 [29], which typically occurs in file systems. We have implemented the authentication service (both the server and the client) in Java. The experiments were conducted on a 64-bit, 2.8GHz Intel based, dual-core, dual processor machine with 2GB main memory and 2MB cache, running Debian Linux 3.1 with Linux kernel 2.6.15 and using the Sun Java JDK 1.5. The time consumed by the garbage collector is excluded from the presented times. We have implemented authenticated versions of all the major commands of a file system (all the commands described in Theorem 1) and the basic functionality of a skip list. We executed the experiments on a remote file system (that lies however in close proximity to the terminal machine), the tree of which consists of roughly 77,779 nodes, of which 61,241 are files and the rest are directories. The average size of the files is 1.22 MB. The total size of the file system is 6.92 GB. However, the distribution of the files is not uniform (certain subtrees are very "heavy").

In Figures 3(a)/3(b) we plot the time taken to write/read our test file system as a function of the size of the portion of the file system processed. Note that our authentication service does not add much overhead to the non-authenticated write/read. Also note that the overhead of the authentication service is more




(a) Writing the file system (b) Reading the file system

**Fig. 3.** Times to write and to read our test file system, using standard and authenticated operations. The cumulative time elapsed is plotted as a function of the size of the portion of file system processed. Each point corresponds to a new batch of 1,000 files processed. Since files have different sizes, the points on the plot are not uniformly spaced in the horizontal direction.

noticeable in the read experiment, with an average overhead per node (directory or file) of 17.61 ms. This is due to the fact that, when we read a file $x$, that lies in a path $\Pi$, we have to authenticate *both* the contents of the file and the existence of the path (by Theorem 1, this task takes $O(|\Pi| \log n)$ time). Also, for a directory $d$, we have to issue $|\mathsf{children}(d)|$ queries to the skip list in order to authenticate completeness. Due to space limitations, more experimental results can be found in the Appendix.

**Discussion.** Our protocols are designed in the client-server model. However, certain applications that require file-system integrity may involve a large number of users, and therefore, to achieve full consistency user interaction is necessary,[3] which results in impractical protocols (since, without other assumptions, $n$ users need to exchange $\Omega(n)$ messages after any update). Unless one resorts to fork-consistency [16, 4], some communication assumptions must be made. For instance, when different users access a remote file system through the same network infrastructure, our protocols are applicable if we assume a single *designated* trusted client that serializes all users' operations and verifies them locally.[4]

An additional issue is related to failure recovery and persistent in a real-life usage of our authentication protocols. In the case of an unsuccessful verification

[3] To see why, assume any secure protocol for verifying the integrity of outsourced storage, and consider an update on the data performed and verified by user $A$. Consider the next operation on the data issued by user $B$. Without interaction, even if users locally keep unbounded state, replay attacks are impossible to defeat, since a malicious server can ignore $A$'s updates on the data without being noticed by $B$.

[4] That is, Athos' verification client can serve as an add-on module of the hosting operating-system kernel that runs in parallel with the system's filer.

of a file system operation, Athos can provide to the higher (or hosting) application complete information about the problematic operation and the current state of the file system in terms of its integrity. In particular, Athos functionality can characterize the exact location in the file system where integrity was not verified and thus pinpoint which file or directory was maliciously (or accidentally) modified by the untrusted server or by the remote storage devices. By keeping appropriate additional information, the higher application is thus able to infer useful information for failure recovery and a complete view of the problem. For instance, one can find which concrete user and with which concrete operation most recently, correctly accessed the (currently problematic) file or directory. Additionally, by using our skip list based authentication approach in combination with existing techniques [1] for authenticating membership queries in the past (i.e., queries that span through previous states of a data set), Athos can offer persistent authentication capabilities, where file-system operations or queries about past views of the file system can be issued and authenticated. In this way, we can support secure audit of the entire outsourced file system.

Finally, Athos can also support authentication of files at the block level. To do that, we introduce one more level of authentication using a skip list on top of a file. The digest of this skip list is now what is stored in the original skip list. The client can update individual blocks of the file and also query for certain blocks of the file. The length of the proof depends on the granularity we use to partition the file into blocks. Obviously there is a trade-off between the size of the verification proof and the data someone needs to download for authentication.

## 5 Conclusions

In this paper we present efficient protocols for verifying the integrity of a file system that is outsourced to an untrusted storage facility. We use cryptographic hashing and efficient data structures to produce and incrementally update, after file system operations, a short and secure digest of the entire file system. This digest is used by a client to efficiently verify that the file system is fully consistent with the history of query and update operations requested by the client to the host server. Our protocols authenticate both the contents of the files and the directory hierarchy of the file system, thus verifying a rich set of file system operations. The authentication of operations uses a short verification or consistency proof that is computed by the server and involves communication and computation overheads that are sublinear in file system size. This makes our authentication schemes applicable in settings where low-computing power and/or low-storage devices need to access a remote file system in a secure way. We authenticate common and important file system operations such as `cd`, `ls` in *logarithmic* time and, through a prototype implementation, we experimentally confirm the efficiency and practicality of our authentication methods.

## References

[1] Anagnostopoulos, A., Goodrich, M.T., Tamassia, R.: Persistent authenticated dictionaries and their applications. In: Proc. Information Security Conference, pp. 379–393 (2001)
[2] Blaze, M.: A cryptographic file system for Unix. In: Proc. Conference on Computer and Communications Security, pp. 9–16 (1993)
[3] Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: Proc. Foundations of Comp. Science, pp. 90–99 (1991)
[4] Cachin, C., Shelat, A., Shraer, A.: Efficient fork-linearizable access to untrusted shared memory. In: Proc. Principles of Distr. Computing, pp. 129–138 (2007)
[5] Cattaneo, G., Catuogno, L., Sorbo, A.D., Persiano, P.: The design and implementation of a transparent cryptographic file system for Unix. In: Proc. USENIX Annual Technical Conference, pp. 199–212 (2001)
[6] Fu, K.: Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology (May 1999)
[7] Fu, K., Kaashoek, M.F., Mazières, D.: Fast and secure distributed read-only file system. ACM Trans. Comput. Syst. 20(1), 1–24 (2002)
[8] Fujita, T., Ogawara, M.: Arbre: A file system for untrusted remote block-level storage. IPSJ Digital Courier 1, 381–393 (2005)
[9] Gobioff, H., Nagle, D., Gibson, G.A.: Integrity and performance in network attached storage. In: Proc. International Symposium on High Performance Computing, pp. 244–256 (1999)
[10] Goh, E.-J., Shacham, H., Modadugu, N., Boneh, D.: SiRiUS: Securing Remote Untrusted Storage. In: Proc. Network and Distr. Sys. Security, pp. 131–145 (2003)
[11] Goodrich, M.T., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: Proc. DARPA Information Survivability Conference and Exposition, pp. 68–82 (2001)
[12] Goodrich, M.T., Tamassia, R., Triandopoulos, N., Cohen, R.: Authenticated data structures for graph and geometric searching. In: Proc. RSA Conference—Cryptographers' Track, pp. 295–313 (2003)
[13] Jammalamadaka, R.C., Gamboni, R., Mehrotra, S., Seamons, K.E., Venkatasubramanian, N.: gVault: A gmail based cryptographic network file system. In: Proc. Conf. on Data and Applications Security, pp. 161–176 (2007)
[14] Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K.: Plutus: Scalable secure file sharing on untrusted storage. In: Proc. USENIX Conference on File and Storage Technologies, pp. 29–42 (2003)
[15] Li, J., Krohn, M.N., Mazières, D., Shasha, D.: Secure untrusted data repository (SUNDR. In: Proc. Operating Systems Design and Impl., pp. 121–136 (2004)
[16] Mazières, D., Shasha, D.: Building secure file systems out of byantine storage. In: Proc. Principles of Distributed Computing, pp. 108–117 (2002)
[17] McGrew, D.: Efficient authentication of large, dynamic data sets using galois/counter mode. In: Proc. Security in Storage Workshop, pp. 89–94 (2005)
[18] Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
[19] Miller, E.L., Long, D.D.E., Freeman, W.E., Reed, B.: Strong security for network-attached storage. In: Proc. File and Storage Tech., pp. 1–13 (2002)
[20] Oprea, A., Reiter, M.K.: On consistency of encrypted files. In: Dolev, S. (ed.) Proc. International Symposium on Distributed Computing, pp. 254–268 (2006)

[21] Oprea, A., Reiter, M.K.: Integrity checking in cryprographic file systems with constant trusted storage. In: Proc. USENIX Security, pp. 183–198 (2007)
[22] Oprea, A., Reiter, M.K., Yang, K.: Space-efficient block storage integrity. In: Proc. Network and Distributed System Security Symposium, pp. 17–28 (2005)
[23] Papamanthou, C., Tamassia, R.: Time and space efficient algorithms for two-party authenticated data structures. In: Proc. Information and Communications Security, pp. 1–15 (2007)
[24] Pletka, R., Cachin, C.: Cryptographic security for a high-performance distributed file system. In: Proc. Mass Storage Systems Tech., pp. 227–232 (2007)
[25] Sarmenta, L.F.G., van Dijk, M., O'Donnell, C.W., Rhodes, J., Devadas, S.: Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In: Proc. Workshop on Scalable Trusted Computing, pp. 27–41 (2006)
[26] Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. 26(3), 362–381 (1983)
[27] Smith, S.W.: Trusted Computing Platforms: Design and Applications. Springer, Heidelberg (2005)
[28] Tamassia, R., Triandopoulos, N.: Efficient content authentication in P2P networks. In: Proc. Applied Cryptography and Network Security, pp. 354–372 (2007)
[29] Tarjan, R., Werneck, R.: Dynamic trees in practice. In: Proc. Workshop on Experimental Algorithms, pp. 80–93 (2007)
[30] van Dijk, M., Rhodes, J., Sarmenta, L.F.G., Devadas, S.: Offline untrusted storage with immediate detection of forking and replay attacks. In: Proc. Workshop on Scalable Trusted Computing, pp. 41–48 (2007)
[31] Yumerefendi, A.Y., Chase, J.S.: Strong accountability for network storage. In: Proc. Conference on File and Storage Tech., pp. 77–92 (2007)

## Appendix

**Dynamic Trees Implementation.** Let $T$ be the tree that represents our file system. The leaves of $T$ are either files or empty directories. We transform $T$ to a new data structure which is essentially a tree $\mathcal{T}$ of paths (Figure 4(b)). Our data structure is based on dynamic trees [26]. On the tree $\mathcal{T}$, we make use of the hashing scheme for authentication of *path properties* in trees from [12]. This hashing scheme is defined over trees of the form of our final tree $\mathcal{T}$ and allows authentication of path properties that satisfy the concatenation criterion: Let $p = p' \| p''$ be a path in $T$ that is the concatenation of paths $p'$ and $p''$. A

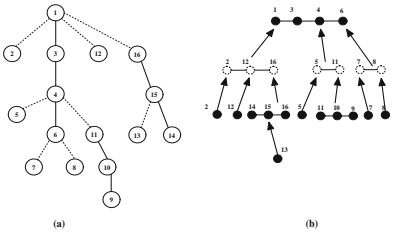


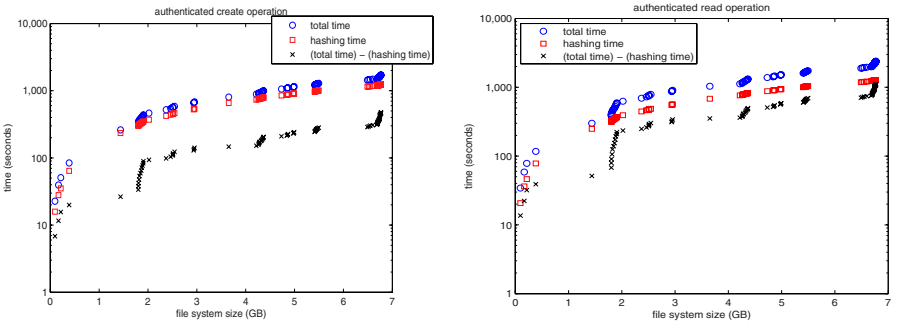
**Fig. 4.** (a) File system tree. (b) The tree of paths $\mathcal{T}$.

path property $\mathcal{P}$ satisfies the *concatenation criterion* if $\mathcal{P}(p) = \mathcal{F}(\mathcal{P}(p'), \mathcal{P}(p''))$, where $\mathcal{F}$ is a function that can be computed in $O(1)$ time; e.g., a path property that satisfies this property is the *length of the path*, where $\mathcal{F}$ is "addition".

We extend this hashing scheme to authenticate path properties not only for paths in the original tree $T$ but also for dashed paths in the intermediate tree $\mathcal{T}$ (i.e., properties of paths related to siblings). This extension is performed by including in the hashing scheme information associated with the files and subdirectories of any directory. Also, we include in the dashed path $d(v)$ related to $v$, the node in $T$ (file or subdirectory) that corresponds to the solid child of $v$ in $T$ (so that no file is missed). Finally, we augment the hashing scheme to include structural and balancing information related to $T$: now the hash value of any node in $\mathcal{T}$ includes its sibling rank and weight.

We now relate operations of the file system with certain path properties in $T$ or dashed-path properties. In order to do this, we define the appropriate path properties of interest: every node $v$ of the tree is related with a constant-size set of node attributes $\{N_1(v), \ldots, N_k(v)\}$. These for example can be the weight of $v$ or other variables that we want to relate with this node. We call the set of these node attributes the *node property* $\mathcal{N}(v)$ of this node. For the case of the file system, we define the node property $\mathcal{N}(v)$ of a node $v$ to contain two attributes: $S(v)$ and $C(v)$. $S(v)$ is the name of the file or directory and $C(v)$ is the hash of the certain file or directory. If node $v$ represents a directory, we define $C(v) = \{\emptyset\}$, otherwise $C(v)$ is a hash of the corresponding to $v$ file. Similarly, every path $p$ is related with a set of path attributes $\{P_1(p), \ldots, P_k(p)\}$. These can be the *length* of a path or other variables that we want to relate with this path. We call the set of these path attributes the *path property* $\mathcal{P}(p)$ of this path. The path attributes can be defined as a function of the corresponding node attributes. In our case, we define the first path attribute $S(p)$ of a path $p = u_1, \ldots, u_\ell$ as $S(p) = \bigotimes_{i=1}^{\ell} S(u_i)$. This is actually the name of the path ($\otimes$ denotes "string concatenation"). The second path attribute is similarly defined to be the content of the path $C(p)$. Hence, $C(p) = \bigoplus_{i=1}^{\ell} C(u_i)$, where $\oplus$ is simply the union operator. Note that the content of a path that consists only of directories is empty. Also the path property $\mathcal{P}(p) = (S(p), C(p))$ for any path $p = p'|p''$ of the file system satisfies the concatenation criterion since $S(p) = S(p') \otimes S(p'')$ and $C(p) = C(p') \oplus C(p'')$. Hence, in the file system context, we can authenticate the major file system operations (see Theorem 2), by reducing them to an appropriately path property query, where we also use complexity analysis in [12]. We finally note that the consistency proof used by the client to do the updates has logarithmic size: since all the update operations described above take logarithmic time, they cannot visit more than $O(\log n + |\Pi|)$ nodes of the tree. Hence, the server can send structural and hashing information of size $O(\log n + |\Pi|)$ that allows the client to update the digest.

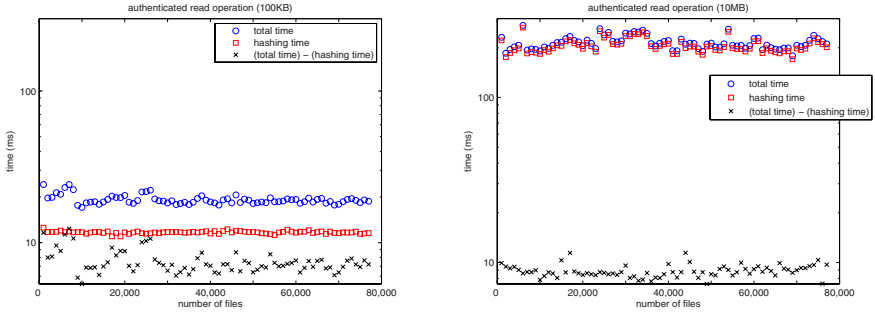
**Additional Experimental Results.** In Figure 5, we further analyze the time to read and write the test file system with authentication by accounting separately for the time taken to perform hashing (these experiments are a more fine-grained analysis of the experiments we presented before). We can see that for both the read and write experiments, the hashing time dominates the




(a) Authenticated create operation (b) Authenticated read operation

**Fig. 5.** Cumulative times for separate parts of the authenticated operations create/read. The most expensive part of either an authenticated create or an authenticated read is the hashing time, as indicated in the above figures. The remaining time is the time needed to send over data to the skip list.

computation. When writing the file system, we need to hash each file and then store the hash in the skip list, whereas to read the file system, we need to hash what we are reading in order to compare it with the authenticated hash that is returned by the skip list and was stored there during creation. We also note that the interaction with the authentication service ((total time) − (hashing time)) increases when our program parses a "light" region of the file system (e.g., the region around 2GB in Figure 5(b)). This is due to the fact that more files are being processed in less amount of time (the "light" region does not contain large files) and therefore the communication with the authenticated skip list increases. Finally, we observe that on average, hashing accounts for 73% of the write time and for 53% of the read time. This overhead is necessary in any authentication method based on cryptographic hashing.




(a) Authenticated read (100KB file) (b) Authenticated read (10MB file)

**Fig. 6.** Average time of an authenticated read operation. Every point is the average time (over 100 executions) for reading a file of certain size in a file system of varying size. The larger the file, the smaller is the difference (total time) − (hashing time).

In Figures 6(a) and 6(b), we plot the average time for reading a file with authentication, as a function of the number of nodes in the file system. Each point $p(x)$ is the average of 100 authenticated reads on a file system that contains $x$ files. Note that these plots are not cumulative. For a 100KB file, the hashing time is about half the total time, whereas for a 10MB file, the hashing time is almost equal to the total time.