# Athos: Efficient Authentication of Outsourced File Systems

Tobias Hardes - 6687549

July 11, 2014

Cloud computing has become increasingly important these days. This can be recognized by the number of requests handled by Amazon [11] and Microsoft [3] or by the huge amount of data that is stored in cloud services. If data is given to a cloud provider, this data is out of any administrative control and it could be manipulated by the storage unit.
The focus of the underlying paper [6] is a platform-independent and user-trans-parent architecture for an authenticated file-system that is outsourced to some untrusted cloud provider. The platform is called *Athos* where file-system operations are verified in logarithmic time related to the size of the file system. *Athos* additional requires constant storage on the client side.
This is achieved by using basic cryptography and data structures that allows a client to efficiently verify the consistency of a file system, including the history of update operations.

# 1 Introduction

When we're talking about cloud computing, we're talking about a set of services, like storage or computational power, that is outsourced to some third party. These are offered through the Internet. Within cloud computing the consumer is enabled to access all resources through the Internet. Within cloud computing the consumer is enabled to access all resources through the Internet from anywhere at any time and usually without any restrictions to technical and physical requirements. Also, the maintenance is outsourced to the service provider. The quick increase of offers in cloud computing has brought various security challenges for both customers and the service providers. [24]
If a user retrieves data from a remote storage, the user must be able to verify the integrity and authenticity of the requested data and with this it must be possible to verify the result of update, delete and insert operations, to ensure the reliability of the storage provider. Furthermore, the status of the file system has to be consistent with the file systems history.
As a general rule we assume, that a server can act maliciously. *Athos* (AuTHenticated Outsourced Storage) is an implementation of a platform-independent and user-transparent architecture for authenticated and outsourced storage. The aim of *Athos* and the underlying paper is to design authentication protocols that allow a client to verify the integrity and consistency of a dynamically evolving file system, including the history of operations requested by the client. Of course any malicious manipulation in the file system should be detected. Also, the overhead for the client should be constant and there shouldn't be any asymptotic extra costs at the server. If this would not be the case, there would be no need for the client to outsource its data. Furthermore, the time for the verification should only be logarithmic or sublinear in the size of the file-system. Otherwise the client could just download the file-system, which is signed and timestamped, after every operation.

The remainder of this document is organized in the following manner. Section 2 of this document gives a motivation and some reasons why standard cryptography is not enough to guarantee authenticity of an outsourced storage [6].
Section 3 discusses other approaches to set up an authenticated file-system that is outsourced to an untrusted server. Section 4 expands the concrete implementation of *Athos*, presenting the overall architecture of *Athos* in the first place. The next paragraph gives a short introduction into authenticated maps that are implemented using skip lists. Based on this I give concrete example of an authenticated file-system based on authenticated maps and skip lists. Section 4 is closed with an evaluation of the security and section 5 gives the analysis of the experimental implementation and discusses related issues. Finally, this paper concludes with section 6 which discusses possible improvements and extensions for this solutions.

## 2 Motivation

In the field of cryptography and IT security, we say that an adversary may not be able to break a scheme efficiently. However, this is not precise enough and so we have to introduce another notion of security. We allow an adversary to potentially succeed with some *small* probability, that is small enough so we are not concerned that it will ever really happen [10]. We use an asymptotic approach, which is rooted in complexity theory and considers the running time and the success probability of an adversary. Furthermore a cryptographic scheme needs a security parameter, which is an integer $n$[1]. According to Katz and Lindell [10] I use the following definition of security:

**Definition 2.1 (Secure scheme)** *A scheme is secure if every probabilistic polynomial time adversary succeeds in breaking the scheme with negligible probability. [10]*

This leads to the definition of a negligible function that is needed to prove a scheme secure.

**Definition 2.2 (Negligible function)** *A function $\mu : \mathbb{N} \to \mathbb{R}^+$ is called negligible, if $\forall c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0\ \mu(n) \leq \frac{1}{n^c}$ [10]*

In other words: For every polynomial $p(\cdot)$ and all large enough n, it holds that $f(n) < \frac{1}{p(n)}$.
If integrity is required, usually cryptographic hash functions are the state of the art:

**Definition 2.3 (Cryptographic hash function)** *A cryptographic hash function is a pair $\pi$ = (Gen, H) of probabilistic polynomial time algorithms, where*

- *Gen($1^n$) takes the security parameter $1^n$ as an input and outputs a key s that is made public. The key s indexes the family[2] of the hash function.*
- *H is deterministic and it takes the key s and an $x \in \{0,1\}^*$ as an input. Then there is a polynomial $l : \mathbb{N} \to \mathbb{N}$ such that $H(s,x) \in \{0,1\}^{l(n)}$.*

*If $H(s,x) = H(s,x')$ with $x \neq x'$ is a collision, $\pi$ is called collision-resistant, if for every probabilistic polynomial time adversary there is a negligible function $\mu : \mathbb{N} \to \mathbb{R}$ such that*

$$Pr[H(s,x) = H(s,x')|x \neq x'] \leq \mu(n)$$

*[10]*

With this definition, a hash function simply maps a string of arbitrary length to a fixed length string that looks random[3].
Applied to an outsourced file-system, the client could keep the hash locally, for each file for example by using Merkle's hash tree [29], and with this, the clients space complexity for authentication of the files can be reduced from linear to constant. This is because

[1] Of course this approach guarantees security only for a large enough n.
[2] For example *SHF1* is a family of hash functions and *SHA1* is an instance of this family. [10]
[3] On a simplistic level, if H(s,x) looks random, no adversary can learn anything about x.

the client just has to store the hash value of the root node. [20]
This approach only provides a partial solution. The integrity of a file-system does not only consists of the integrity of files, but also of the integrity of the directory hierarchy. In many cases an operation is defined within the directory path. For example the default private key for SSH connections is located in $HOME/.ssh. This means the integrity can be ensured by applying a hash over the complete directory tree, but this solution leads to high update costs. Another possible solution could be based on MAC's[4] or digital signatures.

**Definition 2.4 (Signature scheme)** *A signature scheme is a tuple of probabilistic polynomial time algorithms (Gen, Sign, Vrfy) such that:*

- *Gen*$(1^n)$ *takes the security parameter* $1^n$ *as an input and outputs a pair of keys* $(p_k, s_k)$*.* $p_k$ *is called the public key and* $s_k$ *is called the private key.* $p_k$ *is made public.*
- $Sign_{s_k}(m)$ *is the signing algorithm and it takes the private key* $s_k$ *and a message* $m \in \{0,1\}^*$ *as an input. The output is the signature* $\delta$*.*
- $Vrfy_{p_k}(m, \delta)$ *is the deterministic verification algorithm and it takes the public key* $p_k$*, the message* $m \in \{0,1\}^*$ *and the signature* $\delta$ *as an input. The output is a boolean value* $b$*, where* $b = 1$ *meaning valid and* $b = 0$ *meaning invalid.*

*It is required that for every* $n$*, every pair* $(p_k, s_k)$ *outputted by Gen*$(1^n)$ *and every* $m \in \{0,1\}^*$*, it holds that* $Vrfy_k(m, Sign_{s_k}(m)) = 1$*. [10]*

Based on definition 2.4, it is required to sign every possible path in the directory hierarchy to be able to authenticate the locations of files or directories. If the client performs a lot of directory operations that move directories with multiple files, this solution can become inefficient.
Furthermore we could require some tamper-resistant trusted hardware or some trusted storage on the server side like implemented in [4] or [18]. With this approach we assume that the network file-system is at least partly trusted. As shown in [6], this assumption doesn't differ much from simply trusting the complete cloud infrastructure.
So we assume there is no trusted component on the servers side.
*Athos* supports a complete outsourced file-system using the client server model. The implemented protocols support various file-system operations that are performed by the untrusted server and with this the integrity of data and integrity of the file-systems structure can be verified by the client. Thereby, *Athos* ensures that the clients view is always the same as if the file-system was never outsourced.
For this the client only keeps a hash value of the file-system, called the *digest* or the *state information*. Using this *state information* the client is able to validate each operation performed by the server by using small proofs.
To perform these proofs, an *authentication service module* is implemented. This module runs parallel to the file-system module and stores partial file-system meta-data and the hashes. All in all we get the architecture as shown in figure 1.

[4]Message Authentication Code [10]

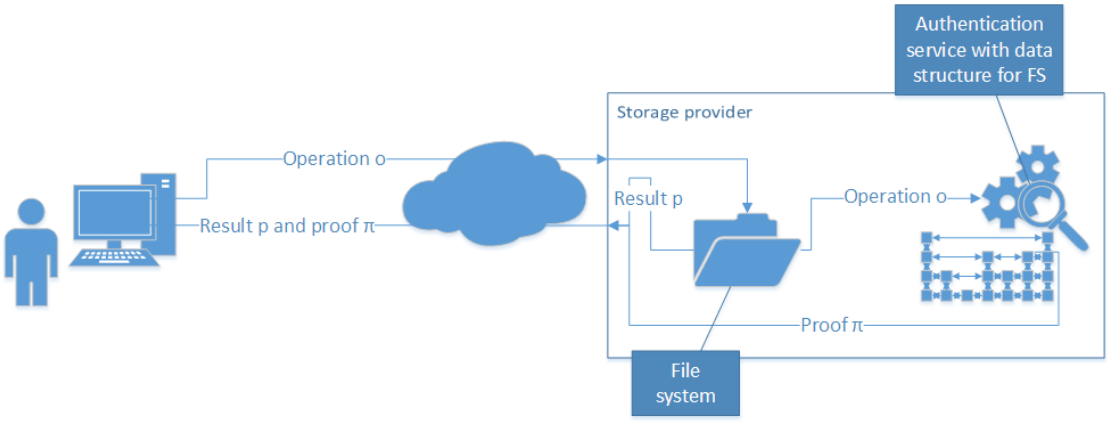



Figure 1: Architecture of *Athos* - [6]

*Athos* uses a single client-server model. This is a common scenario, if a single file-system is outsourced to some cloud provider. Also, a multi-user scenario can be reduced to a single-user scenario. This is done by serializing all requests to the remote machine by using a designated file machine. The multi-client scenario also brings different characteristics. If the clients are not allowed to talk to each other, the server could simply hide operations that were triggered by other clients. This leads to the problem of inconsistency between multiple clients [14]. The presented problem is also known as *fork-consistency* and it is achieved by other projects that are presented in chapter 3. However, using the single-client-server model the forking attack can be detected and prevented by using a simple hash based solution and therefore the fork-consistency property is no longer relevant.

The challenge is to authenticate the structure of the file-system, which is usually not balanced. To achieve this, two authentication structures were introduced. The first construction is based on relations that are used to represent the file-system hierarchy. This approach leads to a low-cost authentication and it uses all benefits of authenticated dictionaries. The data structure is called *skip list* and this approach achieves low-cost authentication and also uses the benefits of authenticated dictionaries that are widely researched [1, 8].
The second possible construction is based on *dynamic trees* [25] to meet the problem of unbalanced file-system structures. However this approach is only of theoretical interest and due to space limitations the focus will be the implementation using skip lists.

# 3 Related work

There are different approaches to ensure the integrity of a file-system. Miller et al. [16] presented a solution, that needs $\mathcal{O}(n)$ space at the client. The fundamental idea is to hash each file and store the hash value on the client side. This approach can be used to ensure the integrity of the file itself, but it doesn't consider the integrity of the file-systems hierarchy.
There are also schemes that are not based on hashing and signing. Oprea and Reiter [19] introduced a scheme that is based on entropy and it just works with encrypted files. David McGrew [15] introduced the technique using the Galois/Counter Mode(GCM), that is a set of operations used to implement an incremental MAC, called the Galois MAC (GMAC). The implementation of McGrew needs linear time for an update operation [6] and it needs an additional memory checker[5] [2] to perform the verification.
Another approach by Oprea and Reiter [18] requires trusted hardware on the servers side. This approach doesn't differ much from simply trusting the complete infrastructure and with *Athos* this additional component is not needed.
Other approaches authenticate the file-system hierarchy by hashing over the complete directory tree or by signing each individual path. However, this approach leads to linear update costs [9] or it is just possible to implement a limited set of operations to reduce the time and space complexity [5].
Athos uses a single client-server model, but there are also implementations that focus on the multi-client-server scenario, because there are different characteristics and different requirements to ensure consistency. Fork consistency (see chapter 2) is handled by the *Secure Untrusted Data Repository* called *SUNDR* in the multi-user-server scenario. Digital signatures and collision resistant hash functions are used to sign different versions of the file-system. The version structure is stored on every client and it is also downloaded from other clients to compare them. If the states of two clients are inconsistent, the server performed a forking attack, which is simply detected by comparing the stored version of different clients. [12, 14].
*Iris* is another implementation to support outsourced file-systems from large enterprises [26]. To reduce the costs for network transfer, *Iris* uses several caching techniques on the client side and for this MACs[6] are used to ensure integrity. The upper layer for the authentication is a balanced Merkle-tree-based structure, but some trusted component is needed to perform the caching and to maintain all file system operations between the enterprise clients and the cloud storage. So in comparison to *SUNDR* there is no need to talk to other clients, but a more extensive infrastructure is needed to organize all clients.

[5] Definition in [2] - Page 2
[6] Message Authentication Codes

# 4 Implementation

*Athos* uses the architecture that is shown in figure 1 on page 5. We have a client $\mathcal{C}$ and an incremental outsourced file-system $\mathcal{FS}$ that is hosted by an untrusted server $\mathcal{S}$. As mentioned in chapter 1, an authentication service module $\mathcal{A}$ is implemented and it is also hosted and controlled by $\mathcal{S}$ to store authentication information about $\mathcal{FS}$. We allow $\mathcal{C}$ to create the file-system $\mathcal{FS}$ using a series of *insert*, *update* and *query* operations that are performed by $\mathcal{S}$. $\mathcal{C}$ keeps the *state information*[7] $s$ which encodes information about the current state of the file-system $\mathcal{FS}$.
Furthermore, let $\mathcal{P}$ be the set of operations that are supported by the file-system $\mathcal{FS}$.
Using these requirements, the communication protocol is as follows [6, 17]:

1. The client $\mathcal{C}$ keeps the state information $s$ and submits a query or update operation $o \in \mathcal{P}$ on $\mathcal{FS}$ that should be performed by $\mathcal{S}$.

2. The server $\mathcal{S}$ performs the operation $o \in \mathcal{P}$ to query or update $\mathcal{FS}$. An update operation leads to a new version of $\mathcal{FS}$ that is called $\mathcal{FS}'$. In both cases we get a result p. Furthermore, let $operate(\cdot,\cdot)$ be the algorithm that performs an operation $o$ on $\mathcal{FS}$ and updates the file-system to $\mathcal{FS}'$. For this we write: $(\mathcal{FS}', p) \leftarrow operate(o, \mathcal{FS})$. By using the authentication service module $\mathcal{A}$, $\mathcal{S}$ also generates a *verification*[8] or *consistency proof*[9] $\pi$.
   The result $p$ of the operation and the proof $\pi$ are returned to $\mathcal{C}$, where $p$ is the answer if $o$ was a query operation or otherwise the empty string $\perp$. Furthermore we set $\mathcal{FS}' = \mathcal{FS}'$ if $o$ was a query operation.
   We write $\pi \leftarrow certify(o, \mathcal{FS}, \mathcal{FS}', p)$.

3. Using the proof $\pi$, the state information $s$ and the operation $o$ with the result $p$, the client either accepts or rejects the input.
   We write $\{(yes, s'), (no, \perp)\} \leftarrow verify(s, p, \pi)$.
   - **o was an update operation and** $verify(\cdot,\cdot,\cdot)$ **outputs 'yes':**
     $\mathcal{C}$ accepts the input and sets $s = s'$.
   - **o was an update operation and** $verify(\cdot,\cdot,\cdot)$ **outputs 'no':**
     $\mathcal{C}$ rejects the input and terminates the protocol.
   - **o was a query operation and** $verify(\cdot,\cdot,\cdot)$ **outputs 'yes':**
     $\mathcal{C}$ accepts the input .
   - **o was a query operation and** $verify(\cdot,\cdot,\cdot)$ **outputs 'no':**
     $\mathcal{C}$ rejects the input and terminates the protocol.

Figure 2 shows a sequence diagram of the protocol.

[7] Also known as the *digest*
[8] Proof returned to a query operation
[9] Proof returned to an update operation

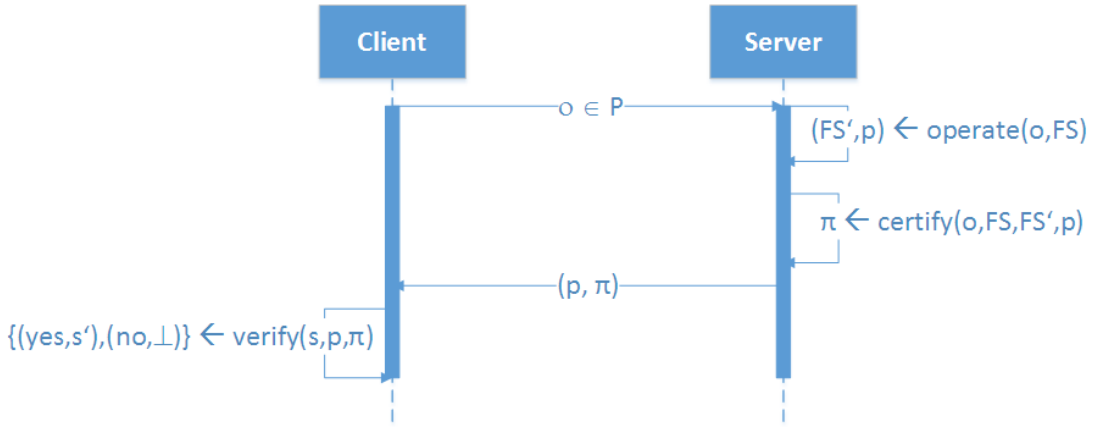



Figure 2: Sequence diagram for the protocol

Using this kind of protocol we can define the concept of an authenticated storage scheme:

**Definition 4.1 (Authenticated storage scheme)** *An authenticated storage scheme is a tuple of algorithms (certify, verify).*

For the verification of the output we need two basic conditions called correctness[10] and consistency.

**Definition 4.2 (Correctness)** *An operation $o \in \mathcal{P}$ is called **correct**, if $(FS'_\tau, p) \leftarrow operate(o, \mathcal{FS}_\tau)$ then $(yes, s') \leftarrow verify(s, p, certify(o, \mathcal{FS}_\tau, \mathcal{FS}'_\tau, p))$ holds.*

So if the operation $certify$ is performed correctly, it generates a proof $\pi$. This proof has to be accepted by the $verify$ operation and with this, $verify$ outputs a new consistent state $s'$ according to the new file-system $\mathcal{FS}'_\tau$.
Consistency is needed to avoid problems like the fork scenario that was mentioned in chapter 3. With this the untrusted server is not able to conceal the users action from each other and so users don't get divided into groups, not seeing operations of the others.

**Definition 4.3 (Consistent)** *Given a series of operations $\tau$, its file-system $\mathcal{FS}_\tau$, a state $s$ that is consistent with the file-system $\mathcal{FS}_\tau$ and a new operation $o \in \mathcal{P}$ such that $(\mathcal{FS}'_\tau, p) \leftarrow operate(o, \mathcal{FS}_\tau)$, then for any polynomial time adversary $A$ having oracle access to $verify(\cdot,\cdot,\cdot)$ and controlling $\mathcal{S}$, given the file-system $\mathcal{FS}_\tau$, the series $\tau$ and the operation $o \in \mathcal{P}$, $A$ produces a proof $\pi'$ and a result $p'$, the probability that either $p \neq p'$ or $s'$ is not **consistent** with $\mathcal{FS}'_\tau$ for the operation $o \in \mathcal{P}$ on $\mathcal{FS}_\tau$ is negligible.*
***The experiment:*** *consistency-forgery$_A(FS_\tau, \tau, o)$*

- *Adversary $\mathcal{A}$ is given $\mathcal{FS}_\tau, \tau, o \in \mathcal{P}$ and oracle access to $verify(\cdot,\cdot,\cdot)$.*
- *$\mathcal{A}$ outputs $p$ and a proof $\pi'$: $(\mathcal{FS}'_\tau, p) \leftarrow operate(o, \mathcal{FS}_\tau)$, $\pi \leftarrow certify(o, \mathcal{FS}, \mathcal{FS}', p)$*

[10] Also called soundness

- *The output is 1 if and only if* $(yes, s') \leftarrow verify(s, p', \pi')$ ***or*** *if* $p \neq p'$*. Otherwise the output is 0.*

$$Pr[consistency\text{-}forgery_A(FS_\tau, \tau, o) = 1] \leq \mu(n)$$

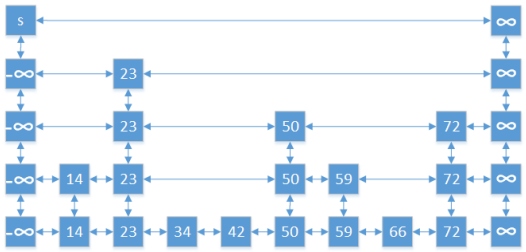
So an adversary is allowed to query polynomial many protocol invocations and then outputs a proof $\pi'$ and a result $p'$. If the operation $verify$ accepts the input $\pi'$ and $p'$, the operation was performed correct, except with negligible probability and the state $s$ is consistent with the new file-system $\mathcal{FS}'_\pi$. So in other words: A state $s$ is consistent with a file-system $\mathcal{FS}_\tau$ for a series $\tau$ of operations on $\mathcal{FS}$, if $s$ and $\mathcal{FS}_\tau$ have been computed by running the algorithms $operate(\cdot,\cdot)$, $certify(\cdot,\cdot,\cdot)$ and $verify(\cdot,\cdot,\cdot)$ sequentially for all operations in series $\tau$ starting with $\mathcal{FS}$ [6].
Using the definitions of consistency and correctness, we can give a definition of a secure authenticated storage scheme:

**Definition 4.4 (Secure authenticated storage scheme)** *An authenticated storage scheme (certify, verify) is secure, if for any series* $\tau$*, a consistent state* $s$ *and a file-system* $\mathcal{FS}_\tau$*, that was created using the series of operations* $\tau$*, the result is correct and consistent according to definition 4.2 and 4.3.*

## 4.1 Athos with skip lists

One way to implement *Athos* is based on an authenticated map that is stored within a skip list. An entry in the map consists of a tuple $(k, v)$, where k is a unique key and v is the value corresponding to the key. The items are sorted by their keys and the authenticated map is stored on the untrusted server $\mathcal{S}$. By using a hash scheme, the state information $s$ for the map is computed according to the skip-list structure in a hierarchical way. This is the way to generate the proofs and the basic approach of *Athos* [8].

The term *skip list* was introduced by William Pugh in 1990 and it was published as a probabilistic alternative to balanced trees [23]. The starting point of a skip list is a simple linked list. In a sorted linked list, we have a worst search time of $\mathcal{O}(n)$, so even though the list is sorted, we don't have a random access to the elements in that list, assumed there is only a pointer at the head. To improve a linked list, simply more links are used by creating different levels of the list. Figure 3 a) shows an example of a skip list with five levels [27]. As shown in figure 3 a), the bottom list contains every element in the data structure. I call this bottom list $L_0$. Then $L_1$, $L_2$ and $L_3$ just copy some elements. In fact $L_1$ holds half of the elements in $L_0$, so $|L_1| = \frac{|L_0|}{2}$ w.h.p. and $L_2$ holds half of the elements in $L_1$ w.h.p. and so on. I also use a special key $-\infty$ as the head node and $\infty$ as the end node for each list.
Using this kind of data structure, we're allowed to execute search queries in time $\mathcal{O}(log(n))$ on average [22, 28].
A search query starts at top left item $-\infty$ and we run the following algorithm: [28]




(a) Example for a skip list with 5 levels, including the state information s.

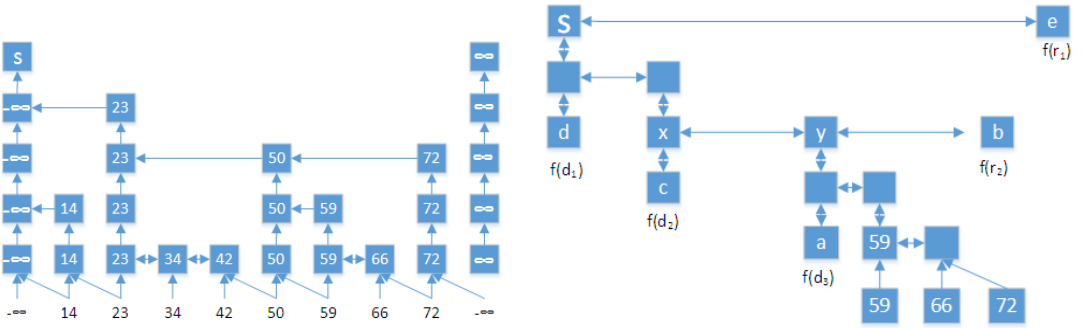
(b) Search for element 66 in the skip list. The visited nodes are highlighted.

Figure 3: Example for a skip list - Based on [7]

**Data**: Key k to search
**Result**: Position p of the key k
p = topleft **while** *pred(p)≠ NULL* **do**
  p = pred(p)
  **while** $k \geq$ *key(succ(p))* **do**
    p=next(p)
  **end**
**end**
return p

**Algorithm 1:** Search in Skip List

Starting from $-\infty$ the algorithm moves on a horizontal link until a position with a key is reached that is greater or equal x. The last link to the next list is used to move downwards and the algorithm starts moving to the right on the new list. Figure 3 b) shows an example of search(66).




(a) Flow of computation for the hash values. An arrow denotes the flow information and not a link in the skip list

(b) The proof for a *search*(66) operation according to figure 3 b)

Figure 4: Examples for the flow computation and a proof - Based on [7] and [8]

To specify the proof in detail, we need a definition of the **search path:**

**Definition 4.5 (Search path)** *Given the skip list and and element $x$ for the search. Then the search path $\Pi(x) = v_1 \leftarrow v_2 \leftarrow \cdots \leftarrow v_{j-1} \leftarrow v_j \leftarrow v_{j+1} \leftarrow \cdots \leftarrow v_m$ is a sequence of nodes where $v_m$ is the node in the top-leftmost and $v_m$ stores the state information $s$. [6]*

To store the structural information and to calculate the proof, I use the strategy that was used by Goodrich et al. during a conference [7] in 2008, in the paper about *Athos* [6] and was also published in a similar way by Papamanthou et. al. in [21].
Assume there is a cryptographic hashing scheme $\mathcal{H}$. $\mathcal{H}$ is extended to $\mathcal{H}'$, that additionally encodes the structural information to use it in the authentication structure[11]. The function value of $\mathcal{H}'$ is represented as $f(v)$ and it is computed as follows:

**Definition 4.6 (Calculation of $f(v)$)** *For every node $v$ of the skip list, let $key(v)$ be the key for the element the node belongs to, $l(v)$ is the respective level of this node and $right(v)$ or $down(v)$ encode the structural information. Then then the definition of $f(v)$ depends on whether $down(v)$ exists: [7, 8]*

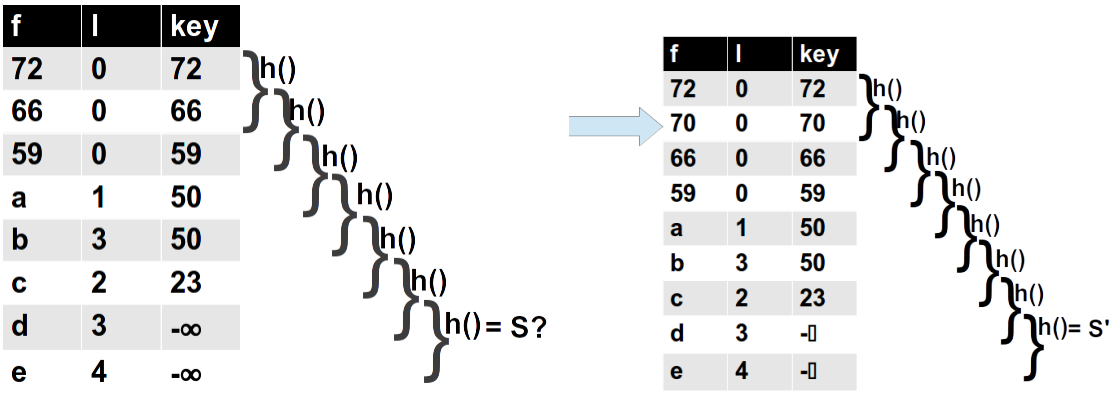
- $down(v) = null \Rightarrow l(v) = 0$
  - *If $right(v)$ is a tower node:* $\Rightarrow f(v) = h(key(v), key(right(v))$
  - *If $right(v)$ is a plateau node:* $\Rightarrow f(v) = h(key(v), f(right(v))$
- $down(v) \neq null \Rightarrow l(v) > 0$
  - *If $right(v)$ is a tower node or $right(v) = null$:* $\Rightarrow f(v) = f(down(v))$
  - *If $right(v)$ is a plateau node:*
    $\Rightarrow f(v) = h([f(right(v)), l(v), key(v)], [f(down(v)), l(v), key(v)])$

*An element that exists in $l(v_i) - 1$, but not in $l(v_i)$, is said to be a plateau element of $l(v_i) - 1$. An element that is in both $l(v_i) - 1$ and $l(v_i)$, is said to be a tower element in $l(v_i) - 1$. [8]*

Figure 4 a) shows the flow computation using definition 4.6.
By calculating the proof, the hashing proceeds from the last to the first element, so from $v_1$ to $v_m$ according to definition 4.5 of the search path.

**Definition 4.7 (Proof path)** *Given an authenticated skip list and an element $x$ with the search path $\Pi(x) = v_1 \leftarrow v_2 \leftarrow \cdots \leftarrow v_m$, let the proof path be $\Upsilon(x) = f(v_1) \circ f(v_2) \circ \cdots \circ f(v_m)$ where $f(v_x)$ is calculated according to definition 4.6.*

[11] skip list




(a) Hashing operations for the *search*(66) operation. The client recomputes the the by accumulating the elements starting from $v_1$ and verifies that the final value is equal to $s$.

(b) Verification for the *insert*(70) operation. $\mathcal{C}$ places x between the successor and predecessor of x and computes the new hash values.

Figure 5: Hashing for search(66) and insert(70) - Based on [7]

Using the scenario of figure 4 b), the first values are computed as follows: $f(72) = key(72)$, $f(66) = key(66)$. We can combine those two values and compute $f(v_1) = h(66, 72)$ according to definition 4.6, because $l(66) = l(72) = 0$ [7]. If we continue in the proof $\Upsilon$, the value for node x is computed as $f(x) = h([f(y), 4, key(23)], [f(c), 4, key(23)])$ and so on.

Assuming that client $\mathcal{C}$ wants to insert an item $x$ that is not stored in the file-system yet. The server returns a consistency proof $\pi$ that consists of the proof path $\Upsilon$ and the search path P for the given element x before the update. P contains the successor $succ(x)$ and predecessor $pred(x)$ of x in the ordering of the keys in level 0. It also contains hashing information to allow $\mathcal{C}$ to recompute the current state information $s$, starting from $v_1$. Because $\mathcal{H}'$ is a cryptographic hash function (definition 2.3) it is collision resistant and $\mathcal{C}$ can check whether the path P is the correct one except with negligible probability (definition 2.2). If P is verified, the client also verifies that x is not in P and with this $\mathcal{C}$ knows the position of x. By using the scenario of figure 4, figure 5 a) shows the hashing to verify the result of *search*(66). The result of the last hash-value is compared to the state information s.
Furthermore, P contains information about the structure that allows $\mathcal{C}$ to perform the update locally. For this $\mathcal{C}$ places x between $succ(x)$ and $pred(x)$ and computes the new hash values for those nodes that need a new hash value. With the new hash values, $\mathcal{C}$ is able to calculate the new state information $s'$ that will be consistent with the update, provided the server has not cheated. By using the scenario of figure 4, figure 5 b) shows the hashing to verify the result of *insert*(70).

**Lemma 4.8** *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:*

1. *The expected update (insertion and removal by $\mathcal{S}$), query (time $\mathcal{S}$ needs to compute the proof) and verification time (time $\mathcal{C}$ needs to in order to process the proof)is $\mathcal{O}(log(n))$ w.h.p.;*

2. *The expected size of the consistency and verification proof (communication cost) is $\mathcal{O}(log(n))$ w.h.p.*

This authenticated hash map can be used to solve more complex operations in a file-system. The hierarchy of a file-system can be visualized using a tree structure.

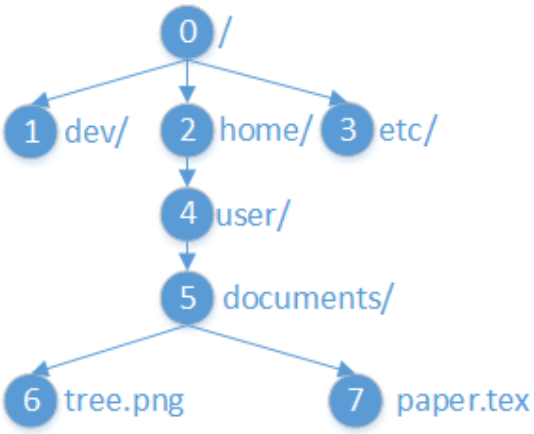



Figure 6: File-system as a tree

Let T be a tree like shown in figure 6 including all files and directories in a file-system. The intention of *Athos* is to map the structural information of the tree T to entries in an authenticated map. One node in T, which can be a file or a directory, belongs to an entry $(k, v)$ in the hash map. To implement an outsourced file-system the following information, according to an i-node in a file-system are stored, using the hash map:

1. **name**: The name in the file-system

2. **hash(file)**: A hash of the file (NULL of v is a directory)

3. **key(parent)**: The key of the parent node of v (NULL if v is the root node)

4. **key(sibling)**: The key of the successor node of v (NULL, if v is the last node in the list)

5. **key(backsibling)**: The key of the predecessor node of v (NULL, if v is the first node in the list)

6. **key(child)**: The key of the first child node of v (NULL, if v has no child)

For this we use the i-node in the file-system as the value for $key(v)$. As an example, the following entries are stored for the nodes 2 and 7 (see figure 6):

$$(key, [name, hash(item), parent, sibling, backsibling, child])$$
$$(2, [\text{“}home'', null, 0, 3, 1, 4])$$
$$(7, [\text{“}paper.tex'', hash(paper.tex), 5, null, 6, null])$$

So assume the user $\mathcal{C}$ performs a $search(/home/user/documents/paper.tex)$ operation. In a file-system we have to verify the path and not only the file itself to ensure integrity. The starting point is the file itself, so the server returns the entry for node 7 and the proof. $\mathcal{C}$ checks whether the proof fits to the state s and then the name of the returned entry. If both checks are successful, $\mathcal{C}$ continues with the parent node of v and does the same checks, until the parent node is set to *null*, which indicates the root node of T. All in all we get the following algorithm: [7]

```
Data: Nodeid
Result: Parent_Nodeid
((7,["paper.tex",h(paper.tex),5,null,6,null]) + π)← server.get(Nodeid);
if s == h(π) then
    if name == "paper.tex" then
        if parent == null then
            exit();
        else
            return 5;
        end
    else
        throw CurruptedDataException();
    end
else
    throw CurruptedDataException();
end
```

**Algorithm 2:** Verification for the element key(v) = 7 - Reflecting the work in [7]

With this we map each file-system operation to a small set of update and query operations in the authenticated map - so file-system operations are reduced to operations like *“Is v in $\mathcal{FS}$?”*[12] with a yes/no answer together with the authentication information which yields the proof [8].
Using this approach we get the following theorem: [6]

**Theorem 4.9** *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, $l_v$ is the number of children of node v and $\Pi = v_1 v_2 \cdots v_k$ is a path in T:*

[12] Also known as 'set-membership operations' [8]

1. *The authentication of any path* $\Pi$ *takes* $t(\Pi) = \mathcal{O}(klog(n))$ *query and verification time.*

2. *Query operations* $cd(\Pi)$, $read(\Pi)$ *and update operations,* $write(\Pi)$, $rm(\Pi)$, $mkdir(\Pi)$, $touch(\Pi)$ *take* $t(\Pi)$ *query, verification and update, query, verification time respectively.*

3. *Query operation* $ls(\Pi)$ *takes* $t(\Pi) + \mathcal{O}(l_{\pi_k} log(n))$ *query and verification time.*

4. *Update operation* $rmdir(\Pi)$ *takes* $t(\Pi) + \mathcal{O}(|T_{\pi_k}|log(n))$ *update, query and verification time.*

5. *Update operation* $mv(\Pi, \Pi')$ *takes* $t(\Pi) + t(\Pi')$ *update, query and verification time.*

Using this approach, just local information are stored (the i-Node number was used as the key). To reduce the complexity for path verification, simply the name of the path from the file-system root can be used for $key(v)$, so according to figure 6, we use */home/user/documents/paper.tex* as the key. This gives us $t(\Pi) = \mathcal{O}(log(n)+|\Pi|)$ for the path verification. However, update operation $mv(\Pi, \Pi')$ takes $\mathcal{O}(t(\Pi)+t(\Pi')+|T|log(n))$ update, query and verification time, where T is the subtree rooted at $\pi_{|\Pi|}$ [6].
This implementation is useful, if the major operations are file system navigations and move operations are less frequent. [6]

## 4.2 Security

The security of *Athos* is based on cryptographic hash functions (definition 2.3). The starting point was a hash function $\mathcal{H}$ that has been extended to $\mathcal{H}'$, which encodes the structural information additionally (see chapter 4.1). The security is based on the used hash function, which must be collision resistant. SHA-2 is an example for a collision resistant hash function [13].
*Athos* makes use of the hashing schemes corresponding to the skip list data structure to efficiently verify *search* operations. Given this scheme the security is proven as follows: Starting with an empty file-system and the consistent state $s$, we can inductively show that client $\mathcal{C}$ updates the state $s$ to $s'$ consistently after any update on the file-system $\mathcal{FS}$. By the definition of the augmented hashing scheme, $\mathcal{H}'$ contains the balancing and structural information and with this the changes in the file-system $\mathcal{FS}$ are completely characterized.
Assume $\mathcal{C}$ performs an *update* operation using the empty file-system $\mathcal{FS}$ and the server $\mathcal{S}$ returns a consistency proof $\pi$. Because $\pi$ contains all the structural information $\mathcal{C}$ is able to locally perform the update and to calculate $s'$, which is the result of $\mathcal{H}'$ and which is consistent with the new file-system $\mathcal{FS}'$. So $\mathcal{C}$ is able to calculate the update as if $\mathcal{FS}$ has never been outsourced. Using this invariant we can conclude that any query is verified in a secure way since the underlying hashing scheme is collision resistant (see definition 2.3).
So we assume that finding a collision is computationally hard and any server $\mathcal{S}$ controlled by an adversary won't be able to forge the proof, except with negligible probability.

# 5 Evaluation

Goodrich et al. implemented a prototype of *Athos* using skip lists [6]. An experiment was performed on a file-system with 77,779 nodes, of which 61,241 were files and 16,538 were directories. The average size of a file was was 1.22 MB and the total size was 6.92 GB.

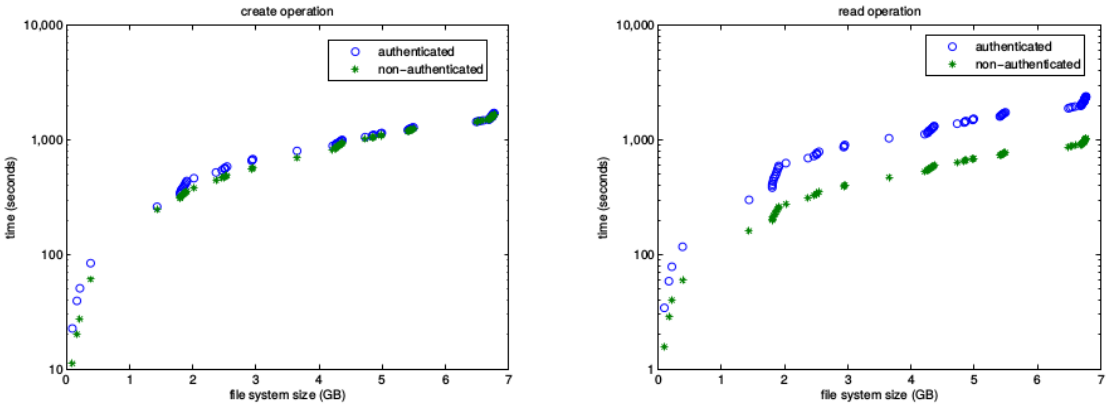



Figure 7: Writing the file system - Reading the file system [6]
Each point illustrates a batch of 1,000 files processed. The points are not uniformly spaced in the horizontal direction, because each file has a different size.

Figure 7 shows the taken time for read and write operations as a function of the size of the portion of the file system processed.
The plot shows that the overhead of the authentication module has an influence on read operations with an average overhead of 17.62 ms per node in T. This is because the file itself and the path have to be authenticated when x is read. Furthermore, if a directory $y$ is given, all children of $y$ in T have to be authenticated in order to guarantee consistency. Using write operations, the authentication services doesn't add that much overhead to the operation.

Figure 8 shows the time to read and write the file-system in more detail. It is conspicuous that the hashing time dominates the computation. The difference between the *total time* and *hashing time* is the time for the interaction with the authentication service module and this time is increasing in the area of 2 GB. This is because simply more files are processed in less amount of time and due to that the communication with the skip list increases. If a file is read, the hashes have to be computed in order to compare the overall result with the digest. It has been observed that 73% of the writing time and 53% of the read time is used for hashing. The remaining time is the time needed to send data to the skip list.

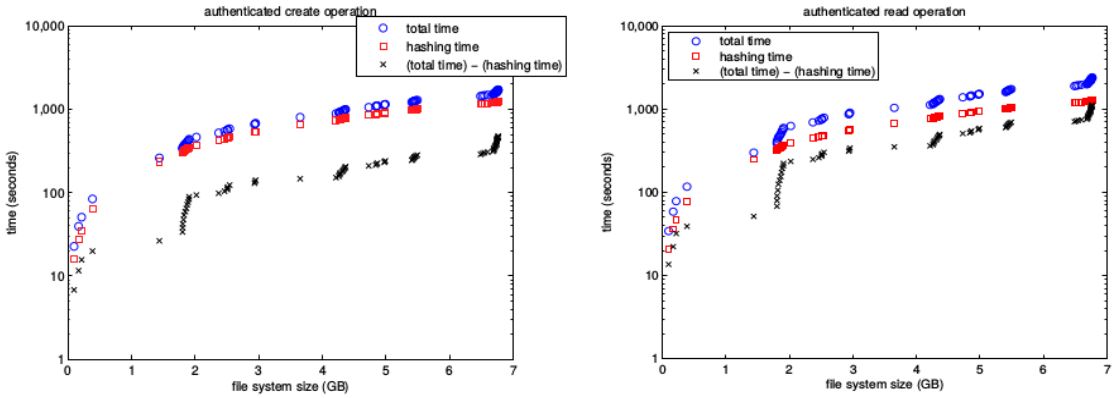



Figure 8: Authenticated create operation - Authenticated read operation [6]

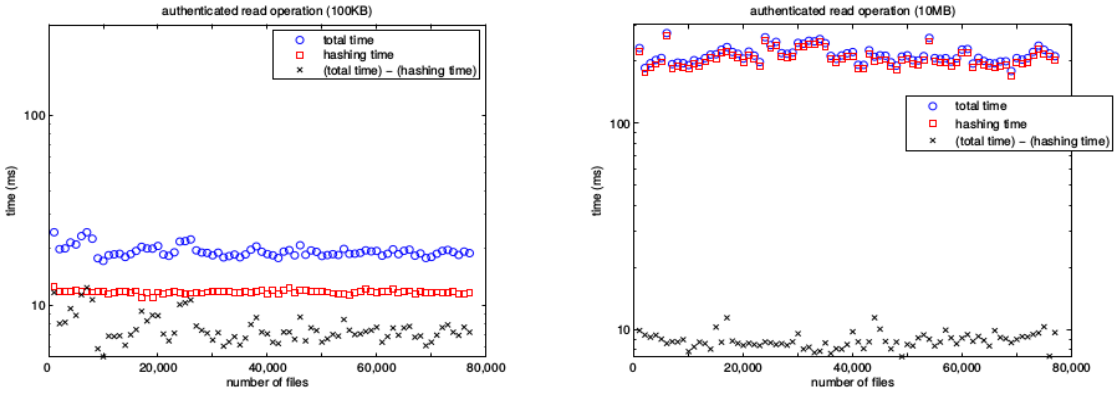



Figure 9: Authenticated create of 100 KB - Authenticated read of 10 MB [6]

Figure 9 shows the time needed for an authenticated read operation. Here each point is the average of 100 authenticated reads and we can see that for a 100 KB file, the hashing time is about half of the total time. By comparison the hashing time for a 10 MB file is almost the same to the total time.

# 6 Remarks

*Athos* is designed to meet the requirements of single-client-server model. However, when using *Athos* it is not possible to achieve consistency in a multi-client-server model. Even if we allow clients to talk to each other we have to exchange $\Omega(n)$[13] messages after any update of the file-system to avoid a forking attack.
By using an additional single trusted server that takes all operations of the clients, this server can serializes all operations and verifies the result from the untrusted server. With this adjustment it is possible to avoid the fork scenario but we have to trust one further

[13] n is the number of involved clients

server. This is also the fundamental idea of *Iris* that was explained in chapter 3.
As mentioned in chapter 4, the protocol terminates, if the verification of the servers result failed. At any time *Athos* knows the complete file system and of course the operation that causes an error within the verification. *Athos* could provide information about the problematic operation and about problems with the integrity to the upper layer, like the application or hosting layer. This upper layer could use further information to deduce confidential information. For instance, the upper layer could collect information to find a concrete user and a concrete operation that most recently, correctly accessed the file or directory which now causes problems.
It is also possible to extend *Athos* to authenticate query operations for the past like $search(x, t)$, which checks whether an element x was in the file-system $\mathcal{S}$ at time $t$. To implement this feature, the construction of Anagnostopoulos et al. [1] for authenticated search operations can be used. It is based on *collision resistant commutative hash functions*[14] and skip lists or red-black trees.
The implementation of Crosby and Wallach [4] improves the implementation of search-operations within *Athos*. The proof is simplified by removing nodes from the path. If the lookup proof already contains the right sibling of every node, then the successor node is already included in the proof. With this there is no need for any nodes to store the keys of their successors. [4] By removing those links, this construction simplifies the design and implementation of mutation operations.
Finally it is possible to adjust *Athos* to support authentication of block level. In addition to the overall skip list, we simply use an additional skip list for each file. With this a client is allowed to query and to update specific blocks of the files. It is then possible to perform proofs for each file on the block level. Of course there is more data to download, if a proof is necessary for each block instead of one proof for the file-system or file.

# 7 References

# References


[1] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *Proceedings of the 4th International Conference on Information Security*, ISC '01, pages 379–393, London, UK, UK, 2001. Springer-Verlag.

[2] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Algorithmica*, pages 90–99, 1995.

[3] Brad Calder. Windows azure storage – 4 trillion objects and counting, July 2012.

[4] Scott A. Crosby and Dan S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Trans. Inf. Syst. Secur.*, 14(2):17:1–17:30, September 2011.


[14] Probability to compute a pair (c,d) such that h(a,b) = h(c,d) while (a,b)≠(c,d) and (a,b)≠(d,c) is negligible

[5] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, page 24, 2002.

[6] Michael T. Goodrich, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Athos: Efficient authentication of outsourced file systems. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security*, volume 5222 of *Lecture Notes in Computer Science*, pages 80–96. Springer Berlin Heidelberg, 2008.

[7] Michael T. Goodrich, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Athos: Efficient authentication of outsourced file systems - presentation. In *Information Security*, 2008.

[8] M.T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference amp; Exposition II, 2001. DISCEX '01. Proceedings*, volume 2, pages 68–82 vol.2, 2001.

[9] Ravi Chandra Jammalamadaka, Roberto Gamboni, Sharad Mehrotra, Kent E. Seamons, and Nalini Venkatasubramanian. gvault: A gmail based cryptographic network file system.

[10] Jonathan Katz and Yehuda Lindell. Introduction to modern cryptography, 2008.

[11] Frederic Lardinois. Amazon's s3 now stores 2 trillion objects, up from 1 trillion last june, regularly peaks at over 1.1m requests per second, April 2013.

[12] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[13] Keith M. Martin. Everyday cryptography : Fundamental principles and applications, 2012.

[14] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 108–117, New York, NY, USA, 2002. ACM.

[15] D. McGrew. Efficient authentication of large, dynamic data sets using galois/counter mode (gcm). In *Security in Storage Workshop, 2005. SISW '05. Third IEEE International*, pages 6 pp.–94, Dec 2005.

[16] Ethan L. Miller, William E. Freeman, Darrell D. E. Long, and Benjamin C. Reed. Strong security for network-attached storage. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, 2002.

[17] Alina Oprea and Michael K. Reiter. On consistency of encrypted files. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 254–268. Springer Berlin Heidelberg, 2006.

[18] Alina Oprea and Michael K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 13:1–13:16, Berkeley, CA, USA, 2007. USENIX Association.

[19] Alina Oprea, Michael K. Reiter, and Ke Yang. Space-efficient block storage integrity. In *In Proc. of NDSS '05*, 2005.

[20] Hwee Hwa Pang, Jilian Zhang, and Kyriakos Mouratidis. Scalable verification for outsourced dynamic databases. *Proc. VLDB Endow.*, 2(1):802–813, August 2009.

[21] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *Information and Communications Security*, volume 4861 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2007.

[22] Prof. Charles Leiserson Prof. Erik Demaine. Lecture 12: Skip lists. In *Introduction to Algorithms (SMA 5503) - Video Lectures*, 2003.

[23] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[24] F.B. Shaikh and S. Haider. Security threats in cloud computing. In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pages 214–219, Dec 2011.

[25] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, June 1983.

[26] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 229–238, New York, NY, USA, 2012. ACM.

[27] Jieping Wang and Xiaoyong Du. Skip list based authenticated data structure in das paradigm. In *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*, pages 69–75, Aug 2009.

[28] Prof. Jennifer Welch. Lecture notes on skip lists, 2004.

[29] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. Efficient query integrity for outsourced dynamic databases. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, CCSW '12, pages 71–82, New York, NY, USA, 2012. ACM.