Efficient Integrity Checking of Untrusted Network Storage

Alexander Heitzmann* aheitzma@cs.brown.edu
Bernardo Palazzi*†‡ palazzi@dia.uniroma3.it
Charalampos Papamanthou* cpap@cs.brown.edu
Roberto Tamassia* rt@cs.brown.edu

ABSTRACT

Outsourced storage has become more and more practical in recent years. Users can now store large amounts of data in multiple servers at a relatively low price. An important issue for outsourced storage systems is to design an efficient scheme to assure users that their data stored at remote servers has not been tampered with. This paper presents a general method and a practical prototype application for verifying the integrity of files in an untrusted network storage service. The verification process is managed by an application running in a trusted environment (typically on the client) that stores just one cryptographic hash value of constant size, corresponding to the "digest" of an authenticated data structure. The proposed integrity verification service can work with any storage service since it is transparent to the storage technology used. Experimental results show that our integrity verification method is efficient and practical for network storage systems.

Categories and Subject Descriptors

E.2 [**Data Storage Representations**]: Data

General Terms

Algorithms, Experimentation, Security, Verification

1. INTRODUCTION

Integrity checking of data and data structures has grown in importance recently due to the expansion of online services, which have become reliable and scalable, and often have a pay-per-use cost model with affordable rates. Corporations and consumers increasingly trust their data to outsourced resources and want to be assured that no one alters or deletes it. Commercial network storage applications are rapidly growing, with services that range from general file storage to web operating systems. Outsourced storage systems sometimes also offer services to assure confidentiality (through encryption) and integrity of data transmission (typically through checksum hashes). However, they do not provide a solution to the storage integrity problem. Thus, the client would have to develop its own authentication solution, such as a cache of the hashes of the data items, in order to verify that data returned by the storage server has not been tampered with. In the remainder of this paper, we use the term "authentication" to refer to the verification of the authenticity, or integrity, of *data*, as opposed to user identity authentication, which is a separate security issue.

It is sometimes assumed that symmetric encryption may be a solution for multiple security problems, but in fact, integrity checking and encryption are orthogonal services (see, e.g., [17]). For example, if we only encrypt files, an attacker can remove some files without our knowledge since decryption will still work perfectly on the remaining files. Only an integrity checking service can detect such an attack.

To deal with these problems, we propose a simple architecture that consists of three main parties:

- The *storage server* stores some outsourced data. The storage server is untrusted and can be any storage service available online.
- The *authentication server* stores and processes authentication information of the outsourced data. The authentication server is also untrusted and can be an outsourced computational resource.
- The *client* queries and updates both the storage server and the authentication server and verifies the results returned by them. We assume that no one can interfere with the state, computation and storage at the client. Of course, it is possible in the real world for a client to be compromised, but we are only interested in protecting the client against errors and malicious behavior by the storage server and authentication server.

In this paper, we propose an efficient and secure technique that allows the client to verify the integrity and completeness of network storage without having to trust the network storage system.

1.1 Previous and Related Work

There has been a considerable amount of work done on untrusted outsourced storage. Yumerefendi and Chase [30] propose a solution for authenticated network storage, using

*Dept. of Computer Science, Brown University, RI, USA.
†DIA, Roma TRE University, Rome, Italy.
‡ISCOM, Italian Ministry for Economic Development—Communications, Rome, Italy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
StorageSS'08, October 31, 2008, Fairfax, Virginia, USA.
Copyright 2008 ACM 978-1-60558-299-3/08/10 ...$5.00.

a Merkle tree [19] as the underlying data structure. PKI is used, however, and the basis (a trusted hash value associated with an authenticated data structure — see Section 2.2) is outsourced to an external medium, raising communication and security issues. Oprea and Reiter [23] present a solution for authenticated storage of files that takes advantage of the entropy of individual blocks. The client keeps hash values only for high-entropy blocks that pass a randomness test. A solution for authenticating an outsourced file system (hierarchically organized) is presented by Jammalamadaka et al. [11]. However their processing of updates is computationally expensive. Fu et al. [6] describe and implement a method for efficiently and securely accessing a read-only file system that has been distributed to many providers. The Athos architecture, developed by Goodrich et al. [9], is a solution for efficiently authenticating operations on an outsourced file system that is related to our approach. Our system and Athos both leverage algorithms described by Papamanthou and Tamassia [24] for querying and updating two-party authenticated data structures.

Untrusted storage where one digital signature for each object is kept is presented by Goh et al. [7]. The SUNDR system, introduced by Mazières et al. [15], protects data integrity in a fully distributed setting by digitally signing every operation and maintaining hash trees. The system requires off-line user collaboration for protection against replay attacks. Goodrich et al. [8] explore data integrity for multi-authored dictionaries, where clients can efficiently validate a sequence of updates. A method for the authentication of outsourced databases using a signature scheme appears in papers by Mykletun et al. [21] and Narasimha and Tsudik [22]. In this approach, the client's computation is computationally expensive. Also, the client has to engage in a multi-round protocol in order to perform an update. A number of works focus on proving retrievability of outsourced data. Schwarz and Miller [26] propose a scheme that makes use of algebraic signatures to verify that data in a distributed system, safeguarded using erasure coding, is stored correctly. Shacham and Waters [27] give provably secure schemes for verifying retrievability that use homomorphic authenticators based on signatures. The model of *provable data possession* (PDP) is proposed by Ateniese et al. [3]. The authors specifically target systems storing very large amounts of data. The client keeps a constant-size digest of the data and the server can demonstrate the possession of a file or a block by returning a compact proof of possession. SafeStore, a system devised by Kotla et al. [13], combines redundancy and hierarchical erasure coding with auditing protocols for checking retrievability.

Di Battista and Palazzi [5] present a method for outsourcing a dictionary, where a skip list is stored by the server into a table of a relational database management system (DBMS) and the client issues SQL queries to the DBMS to retrieve authentication information. Note that this method is fully applicable to our framework since the update of the basis is done at the client's side, whenever an update occurs. A related solution is presented by Miklau and Suciu [20]. Maheshwari et al. [16] take a different approach to the authentication of a database, detailing a new trusted database (TDB) system with built-in support for integrity checking and encryption, and a performance advantage over architectures that add a layer of cryptography on top of a typical unsecured database. A survey for secure distributed storage is presented by Kher and Kim [12]. The archival storage of signed documents is studied by Maniatis and Baker [18].

Our work is related to authenticated data structures in the three-party model, where the data owner outsources the data to a server, which answers queries issued by clients on behalf of the data owner. See [28] for a survey. A solution for the authentication of outsourced databases in the three-party model, using an authenticated B-tree for the indices, is presented by Li et al. [14]. Lower bounds on the client storage in the three-party model are given by Tamassia and Triandopoulos [29]. A method for the authentication of XML documents is provided by Devanbu et al. [4].

1.2 Our Contributions

The main contributions of this paper are the following:

1. We propose an architecture for verifying the integrity of untrusted outsourced storage. For our method to work, no trust is needed at either the storage server or the authentication server (see the definitions above). Our integrity verification service is independent from the storage service and works with any existing storage technology. Note that our solution addresses only the problem of integrity checking. Other security services, e.g., user authentication and data encryption, are orthogonal to and compatible with our service and are not addressed in this paper.

2. We provide efficient algorithms and protocols (of logarithmic complexity) for checking the integrity of data stored at an untrusted storage server using only $O(1)$ space at the client. Namely, suppose that the storage server keeps a file system with n files. The client can verify the integrity of a file downloaded from the storage server in $O(\log n)$ time. Also, the client can verify the correctness and completeness of the list of k file names matching a given path prefix returned by the storage server in $O(k + \log n)$ time.

3. We implement a prototype of our integrity verification system that works with Amazon's *Simple Storage Service* (*S3*) [1].

4. We present the results of experiments on the performance of our prototype, focusing on the communication and processing overhead incurred on top that of Amazon S3. The experiments show that our system provides integrity checking while adding minimal overhead to the normal operations of Amazon S3.

Our architecture has several advantages over many previous methods. Our system requires only constant amount of storage (a single cryptographic hash value) on the client side, irrespective of the amount of outsourced data. Integrity checking is achieved efficiently, with virtually no observable overhead for file systems with hundreds of thousands of files. We maintain authentication information using an authenticated skip list (see Section 2.2), which supports simple and fast updates. Unlike some of the previous approaches, the security of our scheme is independent from probabilistic assumptions about the extent of data corruption. Instead, our system is as secure as the cryptographic hash function used. We do not assume that any component of either the storage server or the authentication server is trusted, therefore any

attack on either server will be detected, even if the two collude in an attack. Thus, the authentication server itself can be an outsourced computational resource.

Another major characteristic of our architecture is that it operates in the single-client setting, unlike other approaches such as SUNDR [15] which supports an authenticated file system in a multi-client setting, but achieves a weaker notion of consistency. This form of consistency is called *fork-consistency* and disallows anything more than the forking attack, where two clients can have a different view of the file system. In our case, full consistency of the file system in a multi-client setting can be provided either by serializing operations from different clients through a common trusted client (e.g., this can be the kernel of the file system), or by requiring each client to communicate its fresh state to all other clients after an update. The latter approach requires additional $\Omega(c)$ communication for c clients.

Our model also differs from data retrievability models such as PDP [3] in a number of ways. Our goal is not to detect corruption of data stored on the server, but to verify that the server's *responses* to the client's queries are consistent with the updates that the client has performed in the past. Thus, integrity checks are performed only when a file or list of files is requested from the storage server. The full response can then be used to verify integrity. Also, we do not require the client to keep any secret information such as a private key, an important distinction in situations where users would like to collaborate without fully trusting each other. Additionally, we are able to verify the completeness and correctness of lists returned from the server as well as the data itself. Finally, no cooperation between the client and storage server beyond the normal, unauthenticated case, is necessary. As a consequence, our integrity checking system can sit on top any existing storage service without the knowledge and cooperation of the storage server.

2. OUR APPROACH

We present a method that allows the client to manage and verify the integrity of content hosted on a remote storage server. Our method uses only a small, constant amount of storage on the client's computer, while the rest of the data needed for integrity checking is hosted on a separate authentication server (see Figure 1). Our technique assumes that both the storage server and the authentication server are untrusted. We can detect any data corruption on either server, even if the two cooperate in an attack. Our authentication server stores authentication information in an authenticated skip list, a data structure described in Section 2.2 that supports efficient updates and queries.

2.1 Skip Lists

The skip list [25] is a probabilistic data structure that maintains a set of elements, each a key-value pair, allowing searches and updates in $O(\log n)$ time with high probability (w.h.p), where n is the current number of elements. A skip list for n elements has $\log n$ levels w.h.p.: the base level is a sorted list of all of the elements; a subset of these elements also appear on the second level; for each node in a given level of the skip list, a coin flip determines whether or not it will exist in the next higher level.

We call the set of nodes associated with an element a *tower*. The *height* of the tower is the level of the highest node in that tower. Each node in the structure contains pointers

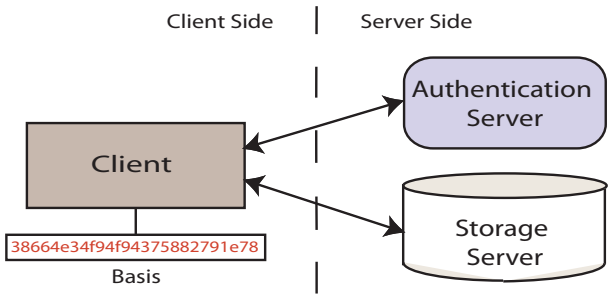

Figure 1: Reference model: The client stores only one hash value (the basis) to verify the integrity of all content on the network storage system.

to the node to its right (R) and to the node below it (B). A search of the structure for a target node T is conducted in the following way. We begin at the top left node. If R > T, then we move to B. Otherwise, we move to R. We continue this process until we are pointing to T (we have found the target), or we are pointing to a node on the base level which is greater than T (T is not contained).

2.2 Authenticated Skip Lists

The authenticated skip list structure [10] supports authenticated versions of the dictionary operations. Namely, the nodes on the base level correspond to data elements whose integrity we would like to safeguard. These data elements can be blocks of any kind: files, pieces of files, directories, etc., as long as they have keys by which they can be sorted. Each node in the structure contains a hash value which is the commutative cryptographic hash (a cryptographic hash of a pair of data, whose value is independent of the ordering of the pair) of the hash values of some pair of adjacent nodes. In this way the authenticated skip list is similar to the Merkle hash tree structure. The hash value of a particular node V in the structure is given as follows. We define V.hash, B.hash, and R.hash to be the hash values of V, and the nodes below and to the right of V, respectively, V.level to be the level of V, and V.key and R.key to be the keys of the data elements associated with V's and R's towers, respectively. The notation $h(A, B)$ indicates a commutative cryptographic hash of the elements A and B. We have:

Case 1 (V.level $= 0$)**:** If R.level $= 0$ (it has only a base level node) then V.hash $= h$(V.key, R.hash), else V.hash $= h$(V.key, R.key).

Case 2 (V.level > 0)**:** If R.level $=$ V.level then V.hash $= h$(B.hash, R.hash), else V.hash $=$ B.hash.

The above rules are indicated by the arrows in Figure 2. The result is a directed map structure. The head of this graph is the top left node of the skip list, and we refer to it's hash value as the *basis* of the structure. The basis is an accumulation of all of the hashes in the whole structure, which means that if any of the base level nodes corresponding to the data are changed, we will be able to detect this change by recomputing the basis and seeing if it has changed. The authenticated skip list includes minimum and maximum nodes on either side, ensuring that a basis will exist even if our data set is empty.

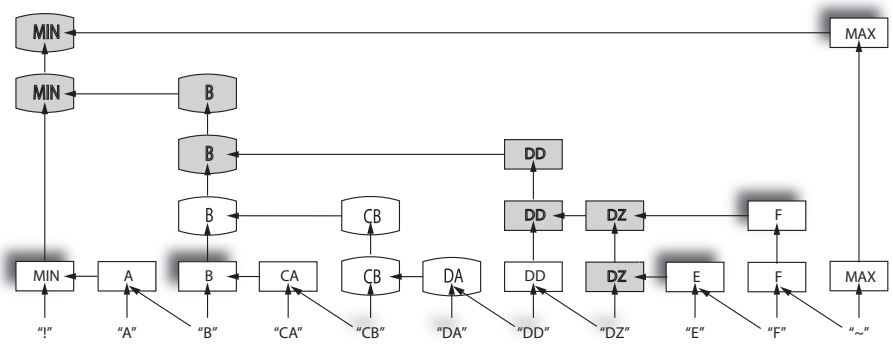

Figure 2: Diagram of an authenticated skip list. The rounded nodes describe the search path for any element greater than "CB" and less than or equal to "DA." The shaded nodes describe the search path for the element "DZ." The nodes with shadows are nodes whose information is included in the proof of integrity returned by the authentication server in response to a LIST query for elements with prefix "D".

2.3 Problem Definition

The problem we address involves two parties: an untrusted server component consisting of the storage and authentication servers, and a client. Even if the standard user identification scheme (Kerberos, for example) used by the storage server protects the client's data from outside attackers, there is still the possibility of a threat from an attacker within the storage server, for example from someone that has unrestricted access to the client's authentication information and account. How can the client be assured that his data will not be tampered with? We need to be able to detect such tampering in the following cases:

- The client requests a list of all of the objects with a given prefix that have been stored in the server, and the response is incomplete or incorrect.
- The client downloads an object from the untrusted storage server, and the content of that object has changed since the client uploaded it.

As an artifact of our architecture, we additionally must detect the case where an operation requiring authentication is performed (a list, download, upload, or deletion) and the portion of the data stored on the authentication server that is needed to authenticate the operation has been corrupted.

2.4 General Architecture

We have designed a general object-oriented software architecture for authenticated network storage services and we have implemented it in Java. A high-level view of the software architecture is shown in Figure 1. In our architecture, there are three entities, the first two of which reside on the server side, and the last of which resides on the client side:

- The *storage server*, which can be any storage service available online. The storage server is untrusted.
- The *authentication server*, which manages all of the authentication information. We run software on this server which is capable of building and maintaining an authenticated skip list structure in response to update requests received from the client, as well as responding to the client's queries about the integrity of outsourced data with proofs of authenticity or corruption. A proof consists of an ordered collection of hashes (a hash chain) and some information about the structure of the authenticated skip list. The authentication server is also untrusted.
- The *client*, who can query both the storage and authentication servers remotely and verifies the answers given to it. Verification is achieved through comparisons to a hash value stored by the client, the basis of the authenticated skip list on the authentication server. This hash (along with the software itself) is the only data which must be stored on the client, and it has constant size dependent only on the cryptographic hash function used. We assume that data stored, and operations performed on the client are entirely trusted, and as this hash value is computed and stored directly by the user when he performs an update, it is trusted. In fact, it is the only trusted value in the entire proposed solution. We run software on the client that makes use of the authentication server to authenticate the client's queries to the storage server. It is worth noting that in the most general case, this authentication software will simply provide an interface that any unauthenticated system can plug in to. Such an API has not been implemented as of now, however, and the implementation presented in this paper is more specific to the particular storage service used.

  To illustrate how this architecture functions, we describe the sequences of actions triggered by some common user requests. Suppose that a user would like to store a file in the storage server and wants to authenticate the PUT operation (see Figure 3). The following steps are performed:

  1. The user selects the file to upload
  2. Our client side software sends two different update queries, one to the storage server and the other to the authentication server.
     - The storage server query adds the user's file to the server.

– The authentication server query, which contains the hash of the file, updates the authenticated skip list on the server and retrieves a proof which allows the client to compute the correct new basis.

At a later time, the user would like to retrieve the file and wants to authenticate the GET operation (see Figure 4). Then the following steps are performed:

1. The user selects the file to download.
2. Our client side software sends two different queries, one to the storage server and the other to the authentication server.
   – The storage server query retrieves the user's file.
   – The authentication server query retrieves the proof of integrity
3. When the client receives both answers, it can verify the integrity of the file (see more details in the next section).

2.5 Algorithms and Complexity

In this section we describe the technical details of our architecture. Suppose a client stores n files (in fact, keyed data blocks of any size can be used) in a storage server, and maintains a corresponding authenticated skip list structure (refer to Section 2.2) at an authentication server. For each file in the storage server (k_i, f_i), a tuple $(k_i, h(f_i))$ is stored in the skip list, where k_i is the key (name) of the file with content f_i, and $h(f_i)$ is a cryptographic hash of f_i. The storage server and the authentication server are synchronized so that they contain the same elements. The basis of the authenticated skip list is stored locally by the client. The client now can issue four main operations which we describe and analyze below. For each of these operations, the main measures of complexity that we are interested in are the following:

1. **Query Complexity**. The time needed for the authentication server to construct the proof in response to a query (either a GET or a LIST or a PUT or a DELETE query).
2. **Verification Complexity**. The time needed for the client to process the proof in order either to verify a GET or LIST query or to update the basis after a PUT or a DELETE query.
3. **Update Complexity**. The time needed for the authentication server to perform an update (either an authenticated PUT or an authenticated DELETE).
4. **Communication Complexity**. The size of the proof (previously referred to as p or p') that must be sent over the network in response to a query.
5. **Hashing Complexity**. The number of hash computations executed during a verification or an update.

Authenticated PUT(k, f).

The client wants to upload to the storage server a file f named k. He sends the request to both servers. The authentication server adds a new entry k associated with the element $h(f)$. At that point it also sends a proof p' back to the client (see Figure 3). In this case p' contains information that allows the client to compute the new basis. Referring to Figure 2, one can see that if we insert a node with $k =$ "D", the only nodes in the skip list whose hash values will change are the rounded ones - the search path for k. To recompute the hash values of these nodes, we need only the hash values of the nodes bordering this search path (nodes whose arrows end at a shaded node), therefore, the proof will contain those hash values. The length of the search path is $\log n$ with high probability (w.h.p). It follows that the complexity of this operation in all five of the above categories is $O(\log n)$ w.h.p. Before computing the new basis, the client validates the proof against the current basis. In this way the client is assured that the new basis he computes is correct.

Authenticated DELETE(k).

The client wants to delete a file from the storage server with name k. This procedure is similar to the procedure PUT(k, f). The complexity of this operation is also $O(\log n)$ w.h.p..

Authenticated GET(k).

The client wants to retrieve from the storage server the file contents of the file with name k. The hash of the file $h(f)$ is stored at a leaf of the authenticated skip list on the authentication server. The client makes a query to the authentication server that returns $h(f)$ along with a proof of the integrity of $h(f)$. Once again, this proof consists of information from the nodes bordering the search path, so the complexity of the operation in each of the applicable categories is also $O(\log n)$ w.h.p.. The proof can be verified against the client's stored basis, and if the verification succeeds, the client can check to see that the hash of the file received from the storage server equals $h(f)$ (see Figure 4).

Authenticated LIST(prefix).

The client wants to retrieve the names (but not the contents) of all the files whose name begins with prefix. We have developed a method for efficiently authenticating a list of k elements taken from a server containing n elements. We obtain a proof from the authentication server that includes the hashes of each of the k elements (the list body), and parts of the proofs for GET operations performed on the prefix and the last list body element (see Figure 2). Additionally, the proof contains the heights of the towers associated with each of the above nodes. We will show that the query and communication, hashing, and verification complexity of this operation is $O(k + \log n)$ w.h.p..

To determine the construction time, we assume that the only time-relevant operation is a comparison, and that this operation takes $O(1)$ time. Referring to Figure 2, one can see that the proof for a LIST operation includes elements from the proofs for GET operations on the prefix and the last element in the list. The number of comparisons performed on the server for a GET operation is $O(\log n)$ (the height of the skip list). Additionally, the proof contains information about each of the k elements making up the body of the list. The query we make for the list body portion of the proof has two steps. First we search for the prefix - this is $O(\log n)$ as well. Second, we move to the right until we reach the end of the list - an additional $O(k)$ comparisons. Summing all of the portions of the proof construction process, we see that the number of comparisons (the query time for LIST operation)

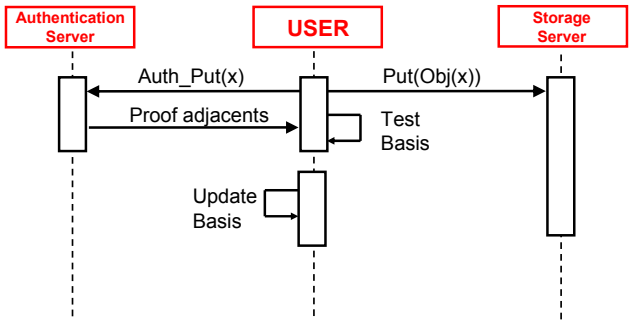

Figure 3: Authenticated PUT for a file with key $= x$.

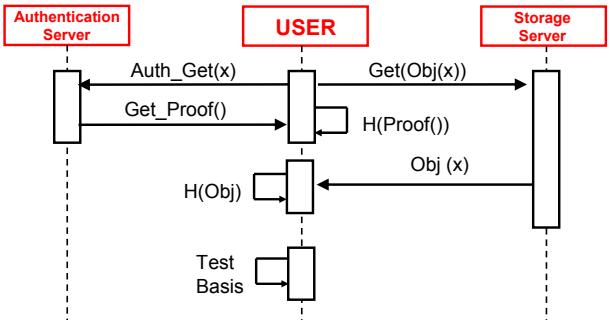

Figure 4: Authenticated GET for a file with key $= x$.

is $O(k + \log n)$. As a result, the size of the proof must also be $O(k+\log n)$, since the size of the proof cannot exceed the number of elements considered during its construction.

After the proof for a LIST query has been built and sent to the client, the client has to run a verification algorithm in order to recompute the basis (which he maintains locally) from the proof. We start with a pointer to the rightmost proof element and and maintain a stack S of proof elements as we proceed to the left. While the height of the current proof element is greater than or equal to that of the stack top, we pop the stack top and absorb its hash value into the current element using a commutative cryptographic hash function. Otherwise, we push the current element onto S, and move the pointer to the left. This verification algorithm processes a proof of size $O(k + \log n)$, and one can see that each element of the proof is passed in to the hash function exactly once. Since the computation of the hash function takes $O(1)$ time, it follows that the verification algorithm takes time $O(k + \log n)$.

2.6 Security

Our service provides protection against a wide range of attacks. An attacker may gain access to our storage server and damage or delete some of our files, or gain access to our authentication server and alter some authentication data, or do both simultaneously in order to try to deceive the client. An attacker may also intercept network communication from the client to one or both servers and change the message contents. The computations performed on the authentication server to update the authenticated skip list may also be controlled by an attacker, resulting in corrupted authentication information. In this section we will show that as long as the client itself is not compromised and the attacker is computationally bounded, the probability that *any* attack on the untrusted portion of the service will not be detected is negligible (see definition of negligible function below).

Here we give a definition of security for our protocol. We recall that a negligible function $\nu(k)$ is a function that decreases faster than any inverse polynomial $p(k)$ as k increases (k is the security parameter, in our case the length of the output of the collision-resistance/cryptographic hash function we use). We also recall that for the specific cryptographic primitive we use, i.e., the collision-resistant hash function, the probability that a computationally bounded adversary can find a collision is $\nu(k)$.

DEFINITION 1 (SECURITY). *Given a storage server S, an authentication server A and a client C that stores n files on S, we say that an integrity checking protocol is secure if:*

- *For a file f' named x stored in S, the probability is negligible that after a $\mathsf{GET}(x)$ query, A computes a proof p and S sends f' such that (p, f') passes the verification test, when in fact, the data in f is corrupted.*

- *For a list Y' of names with prefix y of files stored in S, the probability is negligible that after a $\mathsf{LIST}(y)$ query, A computes a proof p and S sends Y' such that (p, Y') passes the verification test, when in fact, Y' is either incorrect or incomplete.*

We can now prove that our protocol is secure according to Definition 1. Suppose in the beginning (when the data structure contains one element, for example) the client possesses the correct basis. Suppose he issues a $\mathsf{GET}(x)$ query. The server needs to hide the fact that it has tampered with the data of the file named x. In order for the server to do that, it must either find another file f' that has the same hash as the original file and send the correct hash and the incorrect file, or find another hash (for the incorrect file f') that will produce the same basis if included in the hashing scheme of the skip list. Neither task can be accomplished with non-negligible probability since both require finding a collision in a collision resistant hash function - in the former case, the function used to store the files in the leaves of the skip list, and in the latter case, the function used for the hashing scheme within the skip list. This argument can be applied for the LIST query as well.

However, the above is true only if the client always maintains the correct basis, even after updates take place. Indeed, for every update (either PUT or DELETE) the client runs an algorithm that takes as input the proof p' created by the authentication server, some necessary structural information which is included in the hashing scheme, and the existing basis, and outputs the new basis corresponding to the correct authenticated data structure after the update has taken place (See Figure 3). This technique ensures that the basis stored by the client is equal to the hash of the head node of the correct authenticated skip list at all times. One important result is that if an attack is made on the authentication server, altering the skip list stored there, the client will know, because the client's basis corresponds to the *correct* skip list, and the one on the server is now incorrect. This and other practical examples of attacks are discussed below.

Unlike some other security schemes that detect data corruption with some variable uncertainty [3], which basically solve a different problem, our approach guarantees that such corruption will always be detected (negligible uncertainty).

We accomplish this high level of security by maintaining the correct basis on the only trusted component of the system, the client. When an update is made and the basis needs to be changed, all of the relevant computations are also performed on the client, and their correctness is verified against the old basis. In this way, we ensure that the basis will be updated correctly on the client, even if update operation on the server is compromised by an attacker. The possession of this basis allows us to protect against all of the types of attacks mentioned earlier. Even if there is some malicious cooperation between the authentication and storage servers, the attack will be detected - either the proof provided by the authentication server will not agree with the data from the storage server, or it will not agree with the trusted basis on the client, and in either case the client will know there is a problem. Also, note that from the client's perspective the cases for which an attacker intercepts and alters network traffic between client and server are identical to those for which the actual data stored on the servers is altered, therefore our security model is equally adept at detecting them. It is worth pointing out that once we detect an attack, we will not always be able to determine *which* portion of the system was attacked. If an attacker manages to alter some data on the authentication server, the server may not be able to provide a correct proof of integrity to the client, and the client will be unable to determine whether or not an attack on the storage server has occurred as well. For clarity, we summarize these concepts by distinguishing the following cases:

1. No attack is made on either the authentication or storage servers. Result: The client can verify that integrity is preserved.
2. An attack is made on the storage server, but not the authentication server. Result: the attack is detected, and the client determines that the integrity of the data on the storage server has been compromised.
3. An attack is made on the authentication server. Result: the attack is detected, but it may not be possible for the client to determine whether or not the data on the storage server has been corrupted as well.

From a practical perspective, we view the authentication server and the storage server as a single untrusted entity, and although it would be useful to be able to determine the status of the data on the storage server even if the authentication server has been attacked, the only crucial point is that the probability that any attack on the untrusted portion of the service will not be detected is negligible. The only practical disadvantage of separating the untrusted authentication and storage components is two servers instead of one are exposed to attacks. The security of the servers themselves, however, is a topic outside the scope of this paper.

Based on the efficiency of the skip list data structure (main operations run in expected time $O(\log n)$ with high probability (w.h.p)), the results for the LIST implementation we derived before, and the proof of security above, we can summarize the main complexity and security results of this section:

THEOREM 1. *Assume the existence of a collision-resistant hash function. The presented protocol for checking the integrity of n files that reside on the storage server supports authenticated updates* PUT() *and* DELETE() *and authenticated queries* GET() *and* LIST() *and has the following properties:*

1. *The protocol is secure according to Definition 1;*
2. *The expected running time, communication complexity and hashing complexity of* PUT(), DELETE() *and* GET() *is* $O(\log n)$ *at the server and at the client w.h.p.;*
3. *The expected running time, communication complexity and hashing complexity of* LIST() *is* $O(k+\log n)$ *at the server and at the client w.h.p., where k is the size of the returned list;*
4. *The client uses space* $O(1)$*; and*
5. *The server uses expected space* $O(n)$ *w.h.p.*

Taking into account constant factors (see the definitions in [29]), the communication and hashing complexity can be shown to have an upper bound w.h.p. of $1.5 \log n$.

3. IMPLEMENTATION

To validate our software architecture for online storage authentication, we have implemented a prototype of an authenticated network storage service. Our prototype utilizes three pre-existing services/applications: Amazon Simple Storage Service is the untrusted data storage server, Amazon Elastic Compute Cloud provides our untrusted authentication server, and the prototype is built on top of an existing open source project called Jets3t Cockpit. In this section, we present some details about these three components, and then proceed to discuss the architecture of our implementation.

3.1 Amazon S3 and EC2

Amazon Simple Storage Service (S3) is a scalable, pay-per-use online storage service. Clients can store a virtually unlimited amount of data, paying for only the storage space and bandwidth that they use, with no initial start-up fee. The basic data unit in S3 is an object. Objects contain both data and meta-data. Only the meta-data portion is used by S3. The basic container for objects in S3 is called a bucket. Buckets are flat, as opposed to hierarchical; they cannot contain other buckets, only data in the form of objects. Each bucket in S3 has a unique name, and each object has a key that identifies the object within its bucket. A single object has a size limit of 5 GB, but there is no limit on the number of objects per bucket. Each client is limited to 100 buckets. Despite the flat storage scheme, it is possible to simulate hierarchical relationships through either special naming conventions (use of "/" or "." to denote directories) or use of customized object meta-data (pointers to associated files, for example). S3 supports both SOAP and REST requests.

Amazon Elastic Compute Cloud (EC2) is a pay-per-use service that provides online computing resources. A client can start a virtual machine (instance) on EC2 using any complete image of a machine. EC2 makes a number of public images available for running servers, database management systems, development environments, and so on. Clients can also run customized images.

3.2 Jets3t Cockpit

Cockpit is a subset of the open source project Jets3t. It is written in Java. It provides a graphical front-end for managing content stored on S3. The original functionality of Cockpit included support for LIST (with the option of specifying a prefix and/or delimiter) and download (GET) queries, as well as upload (PUT) and delete (DELETE) operations. Additionally, the software provides optional encryption of uploaded data and more advanced features such as generation of public URLs that allow general access to a bucket in S3 for a limited time [2].

3.3 Software Architecture

We have added integrity checking to the four basic operations of Amazon S3: the LIST, GET, PUT and DELETE. Note that these four operations form the core of any storage service. When the client triggers one of these operations, a new call is made in parallel with the original call to S3 (which is left unchanged), to an integrity checker that talks to EC2, where our authentication server resides (see Figure 5). The GUI of Cockpit has been modified to accommodate the additional authentication information.

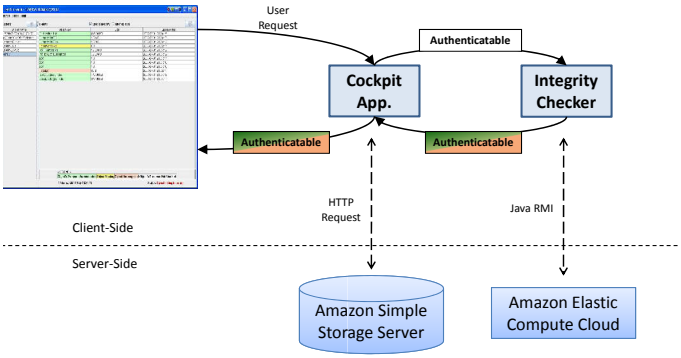

Figure 5: Software interaction architecture.

An abstract class IntegrityChecker (see Figure 6) provides the template for any integrity-checking service. It specifies four abstract methods, corresponding to the authentication of LIST, GET, PUT and DELETE operations, which must be implemented by any child class. Currently, the only implementing class is STMSIntegrityService, which delegates the authentication tasks to a service that stores and retrieves authentication data in main memory on an EC2 instance through Java Remote Method Invocation (RMI). Another service based on a database management system (DBMS) is in development and will provide identical functionality, though performance will undoubtedly differ. In fact, any integrity-checking service that can authenticate those four basic operations can easily plug-in to our prototype simply by extending IntegrityChecker. Information is passed between the GUI application and the integrity checker through objects implementing the Authenticatable (see Figure 6) interface. Implementing classes must be able to store and provide information about the authentication state of their objects' contents, as well as their objects' presence in a list.

While the authentication times for the four main operations are not insignificant compared to the time to complete the unauthenticated versions of these operations, the practical authentication time overhead depends to a large extent on the level of parallelism utilized in the implementation. To this effect, the authentication algorithms used in this prototype allow the network queries and computational operations for authentication to be conducted at the same time that data is being retrieved from S3. The approach to parallelization differs for each of the four operations:

The PUT *and* DELETE *operations.*

An important distinction to make with respect to operations that update, rather than retrieve information, is that a positive authentication result does not guarantee that the state of the relevant files on the storage server is correct. Rather, the only guarantee is that the updated basis stored on the client corresponds to the authenticated data structure in the correct updated state. In other words, to actually authenticate the contents or presence of files on the storage server, the client must make either a GET or a LIST query, respectively. The function of the authenticated update operations is simply to be sure that the stored basis is correct. The update of the storage server and the update of the authentication server are entirely separate. This fact means that it is easy to conduct both updates in parallel, simply by sending the two network requests at the same time. No comparison of results is necessary in this case.

The GET *operation.*

There are two components of the authenticated GET operation that could potentially introduce a noticeable overhead. Our first concern is that the client requests a download of a very large file (1+ GB), in which case simply computing the hash of the file's contents after the download is complete will take a considerable amount of time. To overcome this difficulty we do not wait for the entire file to be downloaded. The hash of the file is computed in pieces while the file is being downloaded, a process which effectively does not add any authentication overhead. Our second concern is that retrieving a proof for a GET operation from the authentication server may, again, take a significant amount of time. Therefore, rather than waiting for a file downloaded from S3 to be available, and subsequently computing its hash value and proving its correctness, we retrieve the correct hash of the given file from the integrity checker *while* the file is being downloaded from S3. When this approach is combined with the hashing scheme described above, the only work left to do after the download is complete (and theoretically the only operation contributing to the time overhead) is a simple comparison of the calculated hash and the one retrieved from the integrity checker.

The LIST *operation.*

For the LIST operation, rather than waiting for the results to be returned from the storage server and then authenticating them with the integrity checker, we request from the integrity checker the list that is guaranteed to be correct, and make a parallel request for the unauthenticated list. Once again, all that is left to do is compare the two lists and keep track of any discrepancies.

In this case, however, there is the additional difficulty that lists may be very large. If we attempt to download authenticated and unauthenticated lists tens of thousands of elements long, we must wait a considerable amount of time before we can even begin the comparison. In the interest of giving the client more immediate feedback, we retrieve

```
public abstract class IntegrityChecker
{
        /** gets the authenticated hash of the contents of the object with the given key.
         *  return a String, the correct hash, or null, if the proof returned from EC2 is incorrect. */
        protected abstract String getAuthenticatedFileHash(String key);

        /** retrieves from EC2 the correct results of a list operation with the given prefix,
         *  starting point priorLastKey, and ending point lastKey.
         *  return the correct listing. */
        protected abstract String[] getAuthenticatedList(String prefix, String priorLastKey, String lastKey);

        /** checks the integrity of the elements adjacent to the object with the given key
         *  and digest, updates EC2 to include that object's information, and stores the new basis.
         *  return true if the the correctness of the new basis is assured, false otherwise. */
        protected abstract boolean performPutUpdate(String key, String fileDigest);

         /** checks the integrity of the elements adjacent to the object with the given key
         *   updates EC2 to remove that object's information, and stores the new basis.
         *   return true if the the correctness of the new basis is assured, false otherwise. */
        protected abstract boolean performDeleteUpdate(String key);
}


public interface Authenticatable extends Comparable<Authenticatable>
{
        /** return an integer which should indicate the authentication state of this object's content,
         *  namely whether its integrity is intact, corrupted, or unchecked. */
        public int getContentAuthenticationStatus();

        /** sets the authentication state of this object's content.*/
        public void setContentAuthenticationStatus(int status);

        /** return an integer which should indicate the authentication state of this object's presence
         *  in a list. If the object is present in the list, this state should indicate whether or
         *  not its presence is authorized, and if it is not present, should indicate whether or not it should be. */
        public int getPresenceAuthenticationStatus();

        /** sets the authentication state of this object's presence. */
        public void setPresenceAuthenticationStatus(int status);

        /** return the string that is the name of the file or object that will be/has been authenticated. */
        public String getKey();

        /** sets the string that is the name of the file or object that will be/has been authenticated. */
        public void setKey(String key);
}
```

Figure 6: The IntegrityChecker abstract class and the Authenticatable interface.

both the authenticated and unauthenticated lists in smaller blocks of 1,000 elements. This approach slightly increases the total time to perform large LIST operations, but gives more regular feedback, and eliminates the possibility that we run out of memory maintaining information on tens of thousands of files.

4. EXPERIMENTS

In this section, we present preliminary experiments conducted with Amazon S3 and EC2. We show that the time overhead that is added due to the authentication service is negligible. We also demonstrate the scalability of our service.

4.1 Setup

We have implemented the authentication service in Java 1.5. Since we were not able to run the client on the same machine for all of the tests, two different machines were used. Machine 1 runs Linux, has 2G RAM, and an AMD Athlon X2 Dual Core 3800+ Processor. Machine 2 runs Windows XP, has 2G RAM and 2.16 GH Intel Core Duo processor. The authentication server runs on a virtual machine (hosted by EC2) equivalent to a computer with a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, 1.7 GB of RAM, 160 GB of disk space, and 250 MB/sec of network bandwidth. The ping time from the client to the server on EC2 is roughly 13.72 ms for machine 1, and 40.25 ms for machine 2 (average of 10 trials). We denote with n the number of elements in the authentication and storage servers. We define the workload of the experiments to be the number of files whose content and/or authentication data is requested by the client, denoted with k, together with the size of the files when their content is requested. When reviewing these results, we must keep in mind that the vast majority of the run time is attributed to network communication, making them highly susceptible to variations in network speed.

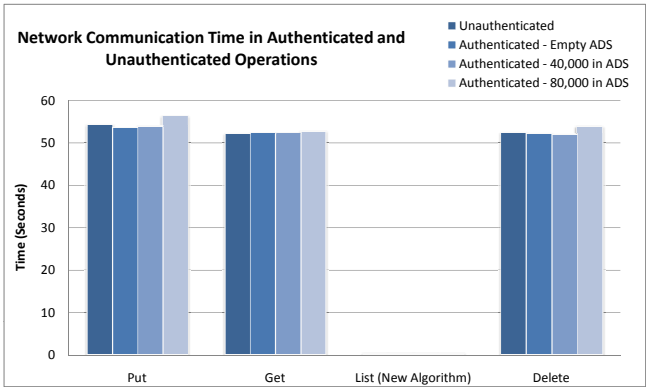

Figure 7: Comparison of non-authenticated and authenticated GET, LIST, PUT, and DELETE operations performed on a workload of 1,000 1K files, and with $n = 0$, 40,000, and 80,000, averaging over 50 trials. Our new, efficient LIST implementation is used.

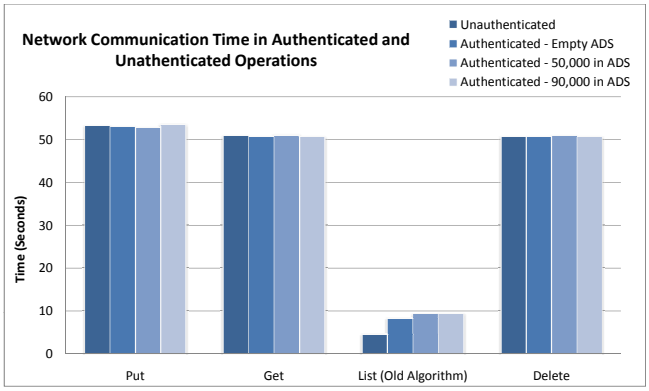

Figure 8: Comparison of non-authenticated and authenticated GET, LIST, PUT, and DELETE operations performed on a workload of 1,000 1K files, and with $n = 0$, 50,000, and 90,000, averaging over 50 trials. An old, LIST implementation is used.

4.2 Overhead Experiments

Figures 7 and 8 show the overhead added to the GET, LIST, PUT, and DELETE operations by our authentication service. We compare the completion times of the four unauthenticated operations with with those of the four authenticated operations as we vary n. The workload is 1,000 1K files. Figure 7 displays the results of the test when run on machine 1. To demonstrate the efficiency of our LIST algorithm, we ran the same test on machine 2 using an older LIST implementation, the results of which are displayed in Figure 8. The procedure used to obtain the data in these figures was as follows: beginning with the original, unauthenticated version of Cockpit, a few lines of code were added to log the system time at the beginning and the end of each operation. A workload of 1,000 files of size 1K was uploaded to S3, a list of those elements was requested, the files were downloaded, and finally, the files were deleted. These PUT, LIST, GET and DELETE operations leave our S3 space in its original state, and we obtain the unauthenticated times for each operation. We repeat until we have the desired number of trials. Next, we run through the same procedure using our authenticated Cockpit, beginning with an empty authentication and storage servers. To show some degree of scalability, we repeat again with different values of n. The results in shown in Figures 7 and 8 indicate that the time to execute the authenticated operations PUT, GET and DELETE differs by less than two seconds in each case from the time to execute the non-authenticated operations. Because of the uncertainty introduced by varying network conditions, it is difficult to say how much the authentication process contributes to the total operation time. As evidence, the authenticated time for many of the operations is actually smaller than the unauthenticated time, a result which can only be explained by variations in communication speeds. We can therefore say that within the precision range of our experiments, there is no time overhead for these operations. These results are a first indication that our service scales well (a topic that we will discuss further in Section 4.3). We would also like to highlight the improved efficiency of the LIST operation. The differences in the run conditions of the tests yielding the two graphs mean that they are not directly comparable. We can, however, compare the LIST times to the GET, PUT, and DELETE times in each individual figure. There are two main points of difference. Firstly, the new LIST completes drastically faster than the old compared to the other operations, even in the unauthenticated case. The primary cause of this change is that the new implementation has allowed us to increase the size of the list blocks from 100 to 1,000. Secondly, the older implementation of the LIST operation introduced significant authentication overhead, while our implementation appears to add no overhead at all. This result is not surprising, because as we discussed in Section 2.5, the computational and communication time for the new operation are both $O(k + \log n)$, a significant improvement over the $O(k \log n)$ bound on the older operation.

4.3 Scalability Experiments

Figures 9, 10, 11, and 12, show how varying n affects the performance of our authentication service for the LIST, PUT, and DELETE operations. Figure 9 was obtained through tests on machine 1 with a workload of 1,000 1K files, varying n from 20,000 through 400,000 at increments of 20,000, while Figures 10, 11, and 12 are results of tests run machine 2, with a workload of 100 1K files, varying n from 10,000 through 200,000, at increments of 10,000. Figures 9 and 10 describe the scalability of the new and old LIST implementations respectively.

The procedure used to obtain these figures was as follows: We began with an empty authentication and storage servers. We wanted to time the operation varying n at intervals of i. For a workload k, we first uploaded k elements, then $i - k$ elements. This step increases n by i. Next, we listed and then downloaded k elements. We repeated this operation for the desired range of n. We then deleted k elements, and then $i - k$ elements, repeating until the authentication and storage servers are empty again. We separated each of the operations (LIST, PUT, and DELETE) into four parts: regular network (retrieval of the data from S3), authentication network (retrieval of the proof from EC2), query response (processing of query on EC2), and verification (processing of the proof on the client side). While the prototype is de-

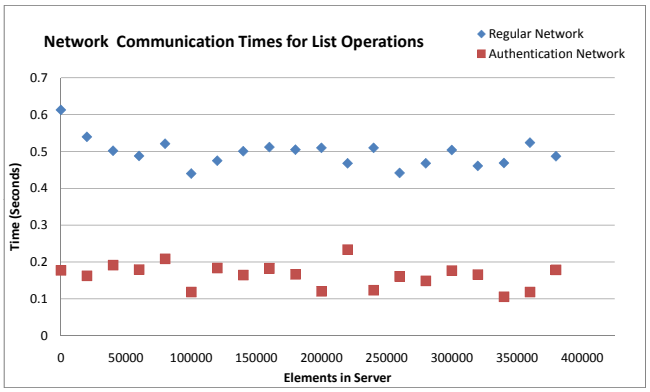

Figure 9: Authentication and regular network communication times for our new, efficiently authenticated LIST operation, varying n with a workload of 1,000 elements.

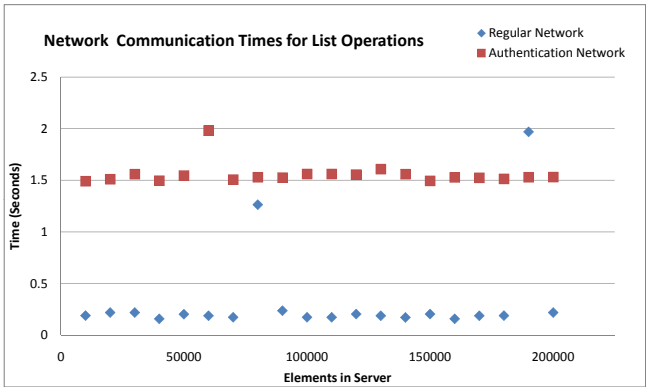

Figure 10: Authentication and regular network communication times for an old authenticated LIST operation, varying n, with a workload of 100 elements.

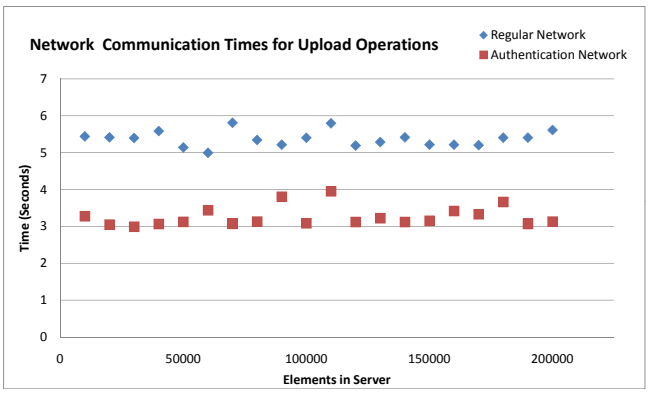

Figure 11: Times for the authentication and regular network components of an authenticated PUT operation, varying the number of elements. The workload in these experiments is 100 1K elements.

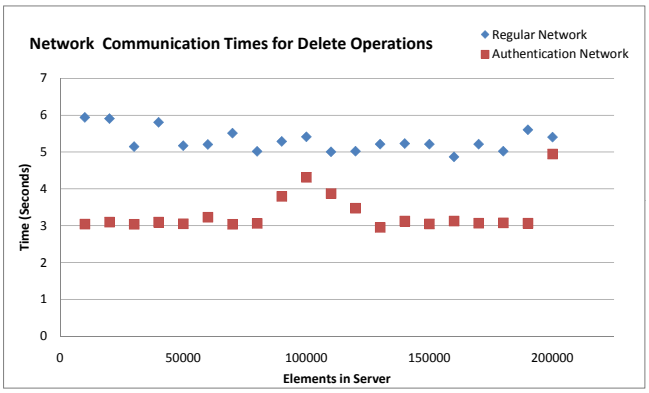

Figure 12: Times for the authentication and regular network components of an authenticated DELETE operation, varying the number of elements. The workload in these experiments is 100 elements.

signed to maximize parallelism, performing the regular and authentication network queries concurrently, for these tests we separated the two components so that they run sequentially, allowing us to time them individually. During the operations, the Java garbage collector (GC) runs periodically. We have collected the GC run times and subtracted them from the times displayed in Figures 10, 11, and 12; Figure 9 is preliminary and does not take the GC into account.

We display only the regular and authentication network times for the LIST, PUT, and DELETE operations. We were unable to obtain reliable results for the GET operation because of the complicated interaction of threads during authentication, and the query and verification times account for only around one percent of the total time of each operation, making them somewhat irrelevant when considering the performance of the service. Ignoring the few outliers, and assuming that the odd peak in Figure 12 is caused by a spike in network traffic, one can see that the overall indication of these plots is that neither the regular or the authentication network operation time for LIST, PUT, or DELETE operations is affected significantly by the number of elements stored with our service. In other words, the service seems to scale extremely well. We can compare these plots to Figures 7 and 8 and see that the total times for each operation are very close to the larger of the authenticated and regular network operation times (the workloads vary, so we are actually comparing the time per element in k). For the PUT and DELETE, and new LIST operations (Figures 9, 11, and 12), the regular network time is larger than the authentication network time, so we expect that when the regular and authentication network queries are performed in parallel, there will be no authentication overhead. In contrast, for the old LIST operation (Figure 10) the authentication network time is larger, so we expect some authentication overhead — once again the improved efficiency of the new LIST implementation is evident.

5. CONCLUSIONS

This paper presents the architecture and implementation of an integrity checking service that extends any existing online storage service. Our architecture is both space-efficient

(the user stores only a single hash value) and time efficient (a very small overhead is added to the operations of the storage service). Our implementation is built on top of Amazon's S3 and EC2 services. The experimental results confirm the negligible time overhead and scalability of our service.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation under grants IIS–0713403 and OCI–0724806, by the Center for Geometric Computing and the Kanellakis Fellowship at Brown University, by IAM Technology, Inc., nd by the Italian Ministry of Research under grant number RBIP06BZW8, project FIRB "Advanced tracking system in intermodal freight transportation".

6. REFERENCES

[1] Amazon S3 (simple storage service). `http://aws.amazon.com/s3`.

[2] JetS3t, an open source java toolkit for Amazon S3. `http://jets3t.s3.amazonaws.com/index.html`.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. ACM Conf. on Computer and Communications Security*, pp. 598–609, 2007.

[4] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. Stubblebine. Flexible authentication of XML documents. *Journal of Computer Security*, 6:841–864, 2004.

[5] G. Di Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. In *Proc. IFIP Working Conference on Data and Applications Security*, pp. 31–46, 2007.

[6] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM Trans. on Computer Systems*, 20(1):1–24, 2002.

[7] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed System Security*, pp. 131–145, 2003.

[8] M. T. Goodrich, M. Shin, R. Tamassia, and W. H. Winsborough. Authenticated dictionaries for fresh attribute credentials. In *Proc. Trust Management Conference*, pp. 332–347, 2003.

[9] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient Authentication of Outsourced File Systems. In *Proc. Information Security Conference*, pp. 80–96, 2008.

[10] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference and Exposition II*, pp. 68–82, 2001.

[11] R. C. Jammalamadaka, R. Gamboni, S. Mehrotra, K. E. Seamons, and N. Venkatasubramanian. gVault: A gmail based cryptographic network file system. In *Proc. Working Conference on Data and Applications Security*, pp. 161–176, 2007.

[12] V. Kher and Y. Kim. Securing distributed storage: challenges, techniques, and systems. In *Proc. ACM Workshop on Storage Security and Survivability*, pp. 9–25, 2005.

[13] R. Kotla, L. Alvisi, and M. Dahlin. Safestore: A durable and practical storage system. In *Proc. USENIX Annual Technical Conf.*, pp. 129–142, 2007.

[14] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pp. 121–132, 2006.

[15] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. Operating Systems Design and Implementation*, pp. 121–136, 2004.

[16] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. Operating Systems Design and Implementation*, pp. 135–150, 2000.

[17] J. Manger. Response on Jungle Disk Blog. `http://blog.jungledisk.com/2006/06/06/encryption/#comment-26`.

[18] P. Maniatis and M. Baker. Enabling the archival storage of signed documents. In *Proc. Conf. on File and Storage Technologies*, pp. 31–45, 2002.

[19] R. C. Merkle. A certified digital signature. In *Proc. Cryptology Conference*, pp. 218–238, 1989.

[20] G. Miklau and D. Suciu. Implementing a tamper-evident database system. In *Proc. Asian Computing Science Conference*, pp. 28–48, 2005.

[21] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. ACM Trans. on Storage, 2(2):107–138, 2006.

[22] M. Narasimha and G. Tsudik. Authentication of Outsourced Databases Using Signature Aggregation and Chaining. In *Proc. In. Conf. on Database Systems for Advanced Applications* , pp. 420–436, 2006.

[23] A. Oprea and M. K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *Proc. USENIX Security Symposium*, pp. 183–198, 2007.

[24] C. Papamanthou and R. Tamassia. Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures. In *Proc. Information and Communications Security*, pp. 1–15, 2007.

[25] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[26] T. S. J. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proc. Int. Conf. on Distributed Computing Systems*, 2006.

[27] H. Shacham and B. Waters. Compact proofs of retrievability. Crypto. ePrint Arch., 08/073, 2008.

[28] R. Tamassia. Authenticated data structures. In *Proc. European Symposium on Algorithms*, pp. 2–5, 2003.

[29] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *Proc. Int. Colloquium on Automata, Languages and Programming*, pp. 153–165, 2005.

[30] A. Y. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proc. Conf. on File and Storage Technologies*, pp. 77–92, 2007.