Int J Digit Libr
DOI 10.1007/s00799-009-0056-2

Techniques to audit and certify the long-term integrity of digital archives

Sangchul Song · Joseph JaJa

© Springer-Verlag 2009

Abstract A fundamental requirement for a digital archive is to set up mechanisms that will ensure the authenticity of its holdings in the long term. In this article, we develop a new methodology to address the long-term integrity of digital archives using rigorous cryptographic techniques. Our approach involves the generation of a small-size integrity token for each object, some cryptographic summary information, and a framework that enables cost-effective regular and periodic auditing of the archive's holdings depending on the policy set by the archive. Our scheme is very general, architecture and platform independent, and can detect with high probability any alteration to an object, including malicious alterations introduced by the archive or by an external intruder. The scheme can be shown to be mathematically correct as long as a small amount of cryptographic information, in the order of 100 KB/year, can be kept intact. Using this approach, a prototype system called ACE (Auditing Control Environment) has been built and tested in an operational large scale archiving environment.

Keywords Integrity auditing · Digital archives · Data integrity · Authenticity of digital archives

1 Introduction

A large portion of the scientific, business, cultural, and government digital information being created today needs to be maintained and preserved for future use of periods ranging from a few years to decades and sometimes centuries. Since the mid-nineties, the issue of long-term preservation of digital information has received considerable attention by major archiving communities, library organizations, government agencies, scientific communities, and individual researchers. These studies have identified major challenges regarding institutional and business models, technology infrastructure, and social and legal frameworks, which need to be addressed to achieve long-term reliable access to digital information.

One of the most challenging problems identified through these studies is how to ensure the integrity of each object of the archive's holdings throughout the lifetime of the object. Digital information is, in general, very fragile due to many potential risks ranging from hardware and software failures to major technology changes rendering current software and hardware unusable, to the ever-growing number of computer and networking security breaches. For instance, it is reported in [1] that disk drive failures alone contributed 400,000 instances of data corruption over a period of 41 months in 1.53 millions of disk drives.

Note also that most of an archive's holdings may be accessed very infrequently, and, hence, several cycles of technology evolution may occur in between accesses to digital objects, thereby causing corrupted files to go undetected until it is too late. In addition, there is also the possibility of human mishandling of the archive holdings (such as operational errors) as well as the possibility of natural hazards and disasters such as fires and floods. We also must be able to handle the possibility of malicious alterations. Most of these problems may cause unnoticeable changes to the archive, which may last for a long time before they are detected.

Two additional factors should also be taken into account when considering long-term digital archives. First, a number

S. Song (✉) · J. JaJa
Institute for Advanced Computer Studies and Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742, USA
e-mail: scsong@umd.edu

J. JaJa
e-mail: joseph@umiacs.umd.edu

Published online: 30 October 2009

of transformations may be applied to a digital object during its lifetime. For example, format obsolescence can lead to a migratory transformation to a new format. Second, cryptographic techniques, used by all current integrity checking mechanisms, are likely to become less immune to potential attacks over time, and hence they will need to be replaced by stronger techniques. Therefore, these two problems need to be also addressed in any approach to ensure the integrity of a digital archive.

A number of bit-level integrity checking techniques tailored for storage systems have been described in the literature [15–17]. However, these techniques fall short of the requirements of a long-term digital archive. Other techniques have been developed specifically for digital archives, including those that appeared in [4,11,12,21], but none seems to offer a general approach that is applicable to the different emerging architectures for digital archives (including centralized, peer to peer, and distributed archives) and that is capable of proactively monitoring and detecting any alterations to the data in a cost effective way.

The main focus of our study is the development of a rigorous methodology to certify the integrity of any object in the archive's holdings, and detect any alterations, including malicious alterations. More specifically, we introduce efficient cryptographic techniques and related procedures to periodically audit the integrity of the various objects held in the archive, which will be able, with high probability, to discover any changes made to any object in the archive, including changes introduced by a malicious user. In fact, our methodology allows a party independent of the archive to audit any object in the archive and certify its integrity with extremely high probability, as long as around 100 KB/year of cryptographic information is kept intact.

We note that a number of schemes can be used to correct errors once they are identified by our method, depending on the architecture of the archive. For a centralized archive with an isolated dark archive, a master copy can be retrieved to correct the corrupted object. For a federated or peer-to-peer distributed archive, a certified (by our scheme) remote replica can be used to replace the corrupted object using a mechanism that will depend on the technical details of the infrastructure.

2 Related work

In this section, we describe some of the most common strategies used to ensure data integrity starting with the basic techniques for bit streams stored on various types of media or transmitted over a network.

2.1 Basic techniques

Data residing on storage systems or being transmitted across a network can get corrupted due to media, hardware, or software failures. Disk errors, for example, are not uncommon, and data on disk can get corrupted silently without being detected because a faulty disk controller causes misdirected writes [17]. This type of errors remains undetected because most storage software expects the media to function properly or fail explicitly rather than mis-operate at any point during its life time. The integrity of data can also get compromised because of software bugs. For example, data read from a storage device can get corrupted due to a faulty device driver or a buggy file system which can cause data to become inaccessible [17]. Moreover, data integrity can be violated because of accidental use or operational errors. Unintended user's activity might cause the integrity to be broken. For instance, deletion of a file might lead to a malfunction of specific application/system software that depends on the accidentally deleted file. As a result of this action, integrity violations may occur.

The simplest technique for implementing integrity checks is to use some form of *replication* such as mirroring. The integrity verification can then be made by comparing the replicas against each other. This method can easily detect a change in the stored data only if the modification is not carried out in all the replicas and no errors are introduced during data movement. While maintaining at least one copy of a replica is inevitably necessary to recover from a potential data corruption, performing constant bit-by-bit replica comparisons to detect integrity violation for every object in an archive is an expensive operation that is prone to errors and that cannot counter malicious alterations.

A well-known approach used in RAID storage is based on coding techniques, the simplest of which is *parity checking* [15]. The parity across the RAID array is computed using the XOR logical function. The parity value is stored together with the data on the same disk array or on a different array dedicated to the parity itself. When the disk containing the data or the parity fails, the data or parity can *sometimes* be recovered using the remaining disk and performing the XOR operation [15]. The XOR parity is a very special type of *erasure codes*, which can be much more powerful [16]. They all involve expanding the data using some types of algebraic operations in such a way that some errors may be detected and corrected. While these techniques are critical in maintaining some level of bit-level integrity on storage systems, they are not designed to support high-level data integrity since decoding will be required every time the data accessed, and they entail a significant expansion of the data. Moreover, since only certain errors can be corrected, they still require that a "master copy" be stored in some kind of a back-up system or a "dark archive".

A widely used method is based on *cryptographic hashing* (also called *checksum*) techniques. In this approach, a checksum of the bit-stream is computed and is stored persistently either with the data or separately. The checksum is calculated using a cryptographic hash algorithm. In general, a cryptographic hash algorithm takes an input of arbitrary length and converts it into a single fixed-size value known as a *digest* or *hash value*. A critical property of cryptographic hash algorithms is that they are based on *one-way* functions, that is, given the hash value of a bit-stream A, it is computationally infeasible to find a different bit-stream B that has the same hash value [8,13]. Assuming that the hash values are correct, data integrity can be verified by comparing the stored hash value with a newly computed hash from the data. Although no known hash function has been proven to be truly one-way, the most common hash functions in use are MD5, SHA-1, SHA-256, and RIPEMD-160, all of which seem to work well in practice (in spite of the recent attacks that illustrated how to break MD5 [19] and SHA-1 [20]). The major problem with this scheme is that it cannot detect malicious alterations since the hash function used by an archive is usually known, and hence an intruder or a malicious user within the archive can change an object and the corresponding hash value so that they still match.

2.2 Techniques for digital archives

We now describe some of the most notable methods that have been suggested for integrity verification for digital archives.

The most popular and, perhaps, the most important method for addressing integrity checking of digital archives is to compute a hash for each object in the archive and store the hashes in a separate, secure, and reliable registry (the hash could in addition be stored with the object as well). Integrity auditing involves periodic sampling of the content of the archive, computing the hash of each object, and comparing the computed hash with the stored hash value of the object. While such a scheme may be sufficient for small, centralized archives, it has two serious shortcomings relative to our stated goals. The first is that a malicious user within the archive or an external intruder can modify both an object and its corresponding hash value (since the hash function is known), in which case there will be no way to detect such an error. The second shortcoming is the fact that the whole scheme depends on ensuring the integrity of all the hash values, which will grow linearly with the number of objects in the archive. Even in the absence of malicious alterations, this is a non-trivial problem for large archives over the long term, especially because the hashing schemes themselves will inevitably change over time in which case we have to track the particular hashing scheme used at any specific time. In the method that we will propose, we only need to ensure the integrity of a single hash value per day, independent of the number of objects in the archive, which is a substantially easier problem to manage.

Another approach uses a combination of replication and hashing. In this approach, each digital object is replicated over a number of repositories. Integrity checking can be performed by computing the hash of each copy locally, and sending all the hashes to an auditor. A majority vote enables the auditor to discover the faulty copies, if any. This is the primary integrity scheme used in LOCKSS [12], which is a peer-to-peer replication system for archiving electronic journals in which each participating library collects its own copy of the journals of interest. LOCKSS uses a peer-to-peer intercache protocol (LCAP) which is a cache auditing protocol. It runs LCAP continuously among all the caches to detect and correct any damage to cached contents. The process is similar to opinion polls in which all the caches vote. When a storage peer in LOCKSS calls for an audit of a digital object, each peer that owns a replica computes the corresponding hash value and sends back the value to the audit initiator. If the computed digest agrees with the overwhelming majority of the replies, then the object is believed to be intact. If the digest disagrees with the overwhelming majority, the object is believed to be tampered with, and the copy is discarded while a new copy is fetched from the publisher or one of the caches with the right copy. As such, LOCKSS is the only scheme described in this sub-section which handles both detection and correcting simultaneously. However, this approach depends crucially on the assumption that there are many replicas for each object. While this assumption may be reasonable for archiving electronic journals at different libraries, many of the current archives do not use the peer-to-peer infrastructure, or create many replicas of each archival object. A replica voting approach can be expensive, requiring a significant communication overhead. In general, achievement of consensus among distributed nodes that do not trust each other (and some of which may be faulty) is a difficult problem that has been studied extensively in the distributed computing literature. In fact, as reported in [5], about 50 malicious nodes could abuse the LOCKSS protocol to prevent a network of 1000 nodes from auditing their contents. We note that additional set of defenses [5] including admission control, desynchronization, and redundancy can be used to counter such an attack but clearly this makes the scheme significantly more complicated and costly.

Another possible approach is to make use of *digital signatures* [3] based on *public key cryptography*. In essence, such a scheme involves a private–public key pair for performing signing/verification operations, and a supporting public-key infrastructure. The basic premise is that the private key is only known to the owner, and the public key is widely available. A message signed by a private key can be verified using the corresponding public key. The digital signature technology takes direct advantage of this property. The digital object is

signed using the private key (note that the signature depends on the digital object *and* the private key), and anybody can verify the signature using the corresponding public key. If the verification process succeeds, the digital object is considered intact (and the identity of the author of the signature verified). Hence, a possible approach to preserving the integrity of digital archives would be to sign each digital object using a private key only known to the archive. However, the certificates (public keys signed by a widely trusted certificate authority) have a finite life with a fixed expiration date. Hence, we need to have a trusted and reliable method to track the various public keys used over time. In general, this is a difficult problem that can be solved using sophisticated techniques based on Byzantine agreement protocols [9] and threshold cryptography [2], which shed serious doubts on its practicality in a production environment. Also, should the private key of the archive be compromised, the whole archive becomes at risk. This implies that a malicious user within the archive or an intruder, who gets access to the private key can easily compromise the contents of the whole archive. Another potential problem with this scheme is its complete dependence on a third party, such as certificate authorities, which may or may not exist over time.

We now introduce the time-stamping technique, which provides an alternative approach to the digital signature scheme outlined above. A time stamp of a digital object D at time T is a record that can be used any time in the future (later than T) to verify that D existed at time T. The record typically contains a time indicator (date and time) and a guarantee (that depends on the time-stamping service) that D existed in exactly this form at time T. One way to implement time stamping is through a Time Stamping Authority (TSA) that attaches a time designation to the object (or its hash) and signs it using the private key of the TSA. The British Library [4] uses this strategy through an independent TSA. With the usage of the public key of the TSA, any alteration to any object, malicious or otherwise, can be detected, which in fact achieves one of our major objectives. However, the verification procedure depends completely on the trustworthiness of a single entity, namely, the TSA. Should the TSA be compromised or disappear sometime in the future, the whole scheme breaks down completely. Moreover, this scheme is computationally expensive, and we still have to deal with the problem of tracking the various public keys used by the TSA over time.

Another approach to time stamping, which will be used as the basis for our scheme, makes use of *linked (or chained) hashing* [7], which amounts to cryptographically chaining objects together in a certain way such that a temporal ordering among the objects can be independently verified. In this approach, there is no need for a fully trusted third party or for tracking certificates over time. In an attempt to address the problem of tracking public keys in a digital signature scheme, the linked hashing technique was also suggested to time stamp the public keys [10]. Our scheme directly applies the linked hashing to target objects, thereby eliminating the necessity of maintaining the public-key infrastructure.

In the next section, we will describe the linked hashing technique as used in our approach and demonstrate its ability to achieve our goals in a cost-effective way without depending on a fully trusted archive or a third party.

3 Our approach

As can be seen from the previous section, the previous integrity checking schemes revolve around the following techniques:

- Majority voting using replicated copies of the object or their hashes.
- Computing and saving a digest (“fingerprint”) for each object, using some well-known hash functions. The auditing process consists of computing the digest from the object and comparing it to the saved digest.
- Creating a digital signature of the object and saving it “with the object.” The auditing process makes use of the public key of either the archive or a third party depending on the particular scheme used. Either way, the integrity of the scheme requires a fully trusted third party and the tracking of certificates over time.

We start by introducing the formal notion of a cryptographic hash function. Such a function compresses an arbitrarily long bit-string into a fixed length bit-string, called the hash value, such that the function is easy to compute but it is computationally infeasible to determine an input string for any given hash value. More formally, we would like our hash function H to satisfy the following two properties.

- *Preimage resistance (one-way property)*: Given any hash value x, it is computationally infeasible to find any bit-string m such that $x = H(m)$.
- *Weak collision resistance*: Given any bit-string m, it is computationally infeasible to determine a different bit-string m' such that $H(m) = H(m')$.

Another property that is sometimes a requirement of cryptographic hash functions is given here.

- *Collision resistance*: It is computationally infeasible to determine any two different strings m and m' such that $H(m) = H(m')$.

These assumptions are the basis for many well-known cryptographic algorithms, including those used in public-key

cryptography (see, for example, [13]). Unfortunately, none of the available hash functions can be shown to satisfy these properties. However, several are accepted by the community as reasonably secure and are currently in widespread use. As noted in Sect. 2, recent study has shown how to break the schemes based on MD5 and SHA-1, but the actual threat posed by such study is not clear and, moreover, there are other schemes that remain intact. It is anticipated that stronger algorithms will be developed over time and, hence, any auditing strategy for long-term digital archives has to provide mechanisms to integrate the newer algorithms without compromising the integrity of the objects that used earlier algorithms.

3.1 Constructing integrity tokens and witness values

A starting point of our approach is a scheme that computes a digest for each object and stores the corresponding digests in a separate registry. A digest is typically the result of applying a one-way hash function on the object, but for our purposes, we will not exclude other techniques for generating digests especially for multimedia objects. As mentioned earlier, a major problem with this scheme is how to ensure the integrity of the digest registry over the long term, especially because the registry grows linearly with the number of objects ingested into the archive. Clearly "attaching" the digest to the object does not solve this problem either.

One can address this problem by compressing all the digests into a small number of hash values, which we will call *witness values*, using collision-resistant, one-way hash functions. For example, we can generate one witness value per day, which cryptographically represents all the objects processed during that day, and hence the total size of all the witness values over a year is quite small (around 100 KB), independent of the number of objects processed during the year.

Given the small size of the witness values, they can be saved on reliable read-only media such as newspapers or archival quality optical media, and hence their integrity can be assured under reasonable assumptions about caring for the media and refreshing the content often as necessary. However it will be extremely time-consuming to conduct regular audits on a large scale archive using the witness values because the auditing of a single object will require the retrieval of the digests of all the objects processed during a day as well as reading the corresponding witness value from a reliable medium. We next show how to counter this problem in a cost effective way.

In order to simplify the presentation, we consider the typical scenario where the generation of the cryptographic information necessary for integrity auditing is placed at the end of the ingestion process, just before an object is archived. We organize the processing of objects into rounds, each of which covers some time interval that is dynamically determined. The length of the time interval depends on the operation of the archive, and may correspond to a fixed duration such as a minute or an hour, a number of objects between a certain minimum and a certain maximum, or may correspond to the time it takes to process a batch of objects according to the archive's schedule. During each round, digests of all the objects being processed can be compressed using any number of schemes, including, for example, the trivial scheme of hashing a concatenation of all the digests in a certain order.

A particular class of such schemes is based on the so-called hash linking, which was introduced to ensure that the *relative temporal ordering* of the objects processed during a round is preserved and cannot be altered without changing the final value. We will make use of the *Merkle tree* [14], which is one of the most widely used hash linking schemes. More specifically, the digests of all the objects being processed in a round form the leaves of a balanced binary tree such that the value stored at each internal node is the hash value of the concatenated values at the children. A random digest value may also be inserted into the tree at each level to ensure that the number of nodes at each level is even (except for the root). The value computed at the root of the tree is the *round hash value*, which represents the compressed value of all the digests (and objects) processed during the round. That is, a change to any of the objects will result in a different round hash value, and, moreover, it is computationally infeasible to determine another set of objects (including reordering the objects) that will yield the same round hash value.

We now define the *proof* of the digest of an object, represented in a leaf of the Merkle tree, as the sequence of the hash values of the siblings of all the nodes on the unique path from that leaf to the root.

Consider, for example, a round involving eight objects with the digest values $h_1, h_2, \ldots, h_8$ (See Fig. 1 for the corresponding tree). The values of the internal nodes are given by:

$$\begin{aligned} h_{12} &= H(h_1||h_2) & h_{34} &= H(h_3||h_4) \\ h_{56} &= H(h_5||h_6) & h_{78} &= H(h_7||h_8) \\ h_{1234} &= H(h_{12}||h_{34}) & h_{5678} &= H(h_{56}||h_{78}) \\ rh &= H(h_{1234}||h_{5678}) \end{aligned}$$

The proof of the object whose digest value is h_5 will be the following sequence:

$$PR_5 = \langle (h_6, r),\ (h_{78}, r),\ (h_{1234}, l) \rangle,$$

where r designates right sibling and l left sibling.

In general,

$$PR_i = \langle (h_j,\ r \text{ or } l) \mid h_j \text{ is the sibling of each node on the unique path from } h_i \text{ to root} \rangle$$

The proof is an essential part of the *integrity token* that is generated for each object. In essence, the integrity token

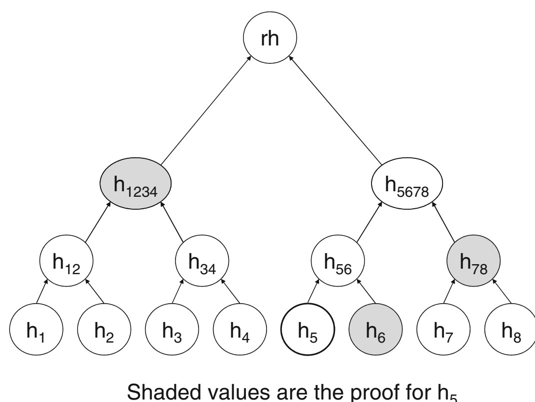

Fig. 1 Merkle tree

consists of the digest, the proof, and a time stamp of the round. It also includes other information that will be needed over the long term, which will be briefly described in the next section.

Given the integrity token of an object, we can quickly compute the round hash value by following the path defined by the proof and performing the concatenation/hash operations as appropriate. For example, with the above PR_5, the round hash value can be computed from $rh = H(h_{1234}||((h_5||h_6)||h_{78}))$. Note that the length of such a path is logarithmic in the number of objects processed during a round and, hence, it is quite small relative to the number of objects.

We reiterate the process by compressing the ordered set of round hash values using one of the hash linking schemes such as Merkle's tree. The resulting value serves as a *witness value*. The granularity of this process can be set dynamically depending on the archive's schedule. Here, we assume that all the round hash values during a day are linked together to generate a witness value. This process can of course be repeated n times, making n-layers of hash linking trees. In our prototype, we stopped at $n = 2$, since the resulting witness values were quite small (less than 100 KB a year).

Once determined, the witness values are stored in reliable read-only media. Our approach depends only on the correctness of the witness values, which is a very reasonable assumption given the total size of the witness values. Based on this assumption, we can achieve the following:

- Our scheme can detect an alteration to any digital object in the archive, malicious or otherwise.
- There is a cost-effective procedure that can periodically audit the contents of the archive to discover any alteration on any object within a short time after the alteration was made.
- Any party, independent of the archive, can audit any object in the archive and assert its integrity based on the witness values.
- No fully trusted third party is needed.

We will later describe our ACE (Auditing Control Environment) system that accomplishes all the goals stated above. We next show how our approach can be adapted to deal with object transformations and updating the hash schemes used by the archive.

3.2 Updating integrity information

There are two cases in which the integrity information must be updated. The first case is when the archive decides to substitute a stronger hash function for one of the hash functions currently in use because of some recently discovered potential threats. The second is when the archive decides to apply certain transformations to some of the objects (because of the possibility of a format becoming outdated, for example). There is an existing solution to deal with renewing the integrity information for the first case by re-registering each related object with the old integrity token attached to it (see, for example, [6]). Such a solution will ensure our ability to verify the integrity of the object since its ingestion into the archive as articulated in this earlier study. This process increases the size of the integrity token, but has no impact on the sizes of the other integrity components.

We now discuss how to renew the integrity information in the case when the object is subjected to a transformation. A possible solution would be to re-register the new object by concatenating the hashes of the old and the new form of the object and an identifier of the transformation, and use the resulting string as if it were the hash of an object to be registered. However, this scheme could be computationally demanding and too complicated to be of practical use. We assume that an archive has to preserve certain (sometimes all) versions of an object which can form an authenticity chain between the original version and the current version of the object. The chain may consist of the current version and the original version of the object. Since a transformation will lead to a new version of the object, and, hence, a new object with its own identifier (could be the old identifier concatenated with the version number), it will participate in a hashing round to obtain its new cryptographic information using the same method as before. However, in this case, we will include the unique identifier of the previous version in the authenticity chain in the integrity token. Note that the integrity of an object should be verified before it is transformed into a new format to ensure its authenticity at this time of its history. The inclusion of the identifiers of previous versions in the integrity tokens will enable us to go through the authenticity

chain and establish the integrity of each version as well as the validity of the corresponding transformation.

4 Putting the ideas together—the ACE tool

Making use of the ideas described in the previous section, we presented in [18], an early implementation of the ACE (Auditing Control Environment) prototype system. More recently, we released Version 1.0 of ACE, which includes some refinements to the earlier prototype. Here, we present a brief overview of the ACE architecture, illustrate its auditing processes, and report on its performance on a large scale production environment.

4.1 ACE components

ACE consists of two major components: the first, called IMS (Integrity Management System), is a third-party service provider that generates the integrity tokens upon request from an archive. A single IMS can simultaneously serve multiple archives, including multiple nodes of a distributed archive. It also maintains the round hash values and generates the witness values. In ACE, the integrity tokens contain several pieces of information in addition to the proof and the time stamp (for example, the ID of the hash algorithm used, the version number of object, and last time the object was audited). Also, ACE links consecutive round hash values sequentially. The second major component is the Audit Manager (AM), which is local to an archive and functions as a bridging component between the IMS and the archive. In particular, the AM sends requests to the IMS to generate the integrity tokens for a number of objects, and once received, the tokens are stored in a local registry. Figure 2 shows the overall ACE architecture of the general case of a distributed archive with dispersed nodes operating asynchronously.

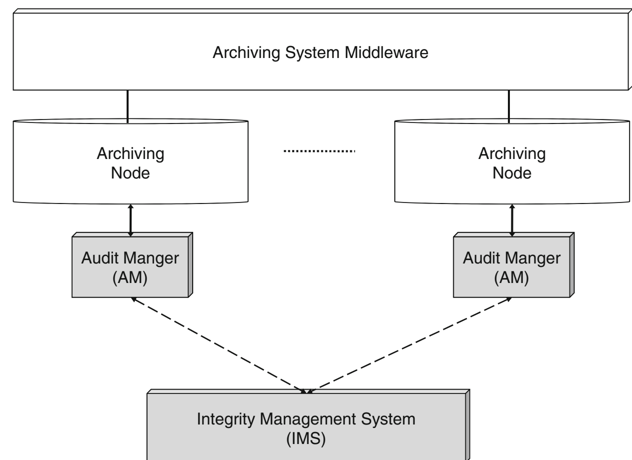

Fig. 2 ACE architecture

4.2 ACE workflow

In this subsection, we discuss a typical workflow with ACE, which includes two major operations: registration and auditing described next.

4.2.1 Registration

For an object to be registered into ACE, the audit manager creates a registration request and submits it to the IMS. When the IMS issues an integrity token for the object, the audit manager stores it locally in a special registry for the archive (local node, if it is a distributed archive). In the meantime, the IMS runs a continuous series of aggregation rounds. Each round closes either when the round receives the maximum number of registration requests, or when the maximum amount of time allocated for a round is reached, whichever comes first. These parameters are assigned by the IMS administrator. This round interval policy can be also overridden through a special object registration request, which forces the current round to immediately close and return an integrity token. Object registration requests received during a round are aggregated together along with a number of random values through the Merkel-tree hash-linking. The random values are added as necessary to ensure a minimum number of hash-linking participants in a round. The resulting round hash value is managed internally within the IMS, and an integrity token is issued to each AM who originally sent a registration request.

At the end of each day, the IMS constructs a witness value using the hash round values of the day, sends it to the participating archives, and stores it in a reliable medium (the current ACE implementation publishes witnesses to a Google group, and stores the value on a CD-ROM). These witnesses are cryptographically dependent on round hash values, which are themselves cryptographically linked to the hashes of the objects registered during that day.

4.2.2 The auditing process

ACE currently performs periodic auditing on the archive's objects following the policy set by the manager of the archive. The policy can be set at an object or collection level. For example, the policy for a certain collection could involve auditing all the objects in the collection every three months, while the policy set for another collection could be to audit all the objects in that collection every six months. A default policy will be set during registration time unless the archive manager sets it differently. Moreover, the auditing process can be invoked by the archive manager at any time on any object.

Table 1 ACE performance

Collection	No. of files	Size (GB)	Audit time
CDL	46,762	4,291	20 h 32 min
SIO-GDC	197,718	815	6 h 49 min
ICPSR	4,830,625	6,957	122 h 48 min
NC state	608,424	5,465	32 h 14 min

When applied to a specific object O, the auditing process consists of the following steps:

1. The audit manager computes the hash value of O and retrieves its integrity token.
2. Making use of the computed hash value of O and the proof contained in the integrity token, the audit manager computes the round hash value.
3. Making use of the round time stamp contained in the integrity token, the audit manager requests the corresponding round hash value from the IMS.
4. The audit manager successfully terminates the auditing process if the computed hash value in Step 1 is equal to the hash value stored in the integrity token, and the two round hash values computed in Steps 2 and 3 are equal. Otherwise, it sends an error alert to the archive manager.

It is clear that if the two hash values, as well as the two round hash values, computed through the auditing process are equal, then the object and the integrity token are intact with a very high probability. A more elaborate process, which will happen infrequently, will involve the witness values as follows. The audit manager requests from the IMS the proof for the round hash value. On receiving this request, the IMS aggregates all the round hash values for the day to determine the proof, and returns the proof to the audit manager. Making use of the proof, the AM computes the corresponding witness value of the day and compares it to the value stored on the read-only media. This elaborate process will ensure the trustworthiness of the IMS. A failure of this process will automatically invalidate the object under the auditing process.

We note that the same process can be applied by a party independent of the archive to verify the integrity of an object. The independent party will request the integrity token from the archive, and then the round hash value and its proof from the IMS. Making use of this information, she/he can quickly compute the witness value of the day on which the object was registered into ACE. She/he can then compare it with the corresponding witness value stored in the read-only media. Any alterations introduced by the archive or the IMS will be detected with very high probability.

4.3 ACE preliminary performance evaluation

ACE Version 1.0 has been deployed against a variety of collections managed by the Chronopolis archiving environment. Chronopolis is a collaboration between the San Diego Supercomputer Center(SDSC)/UCSD Libraries, National Center of Atmospheric Research (NCAR), and the University of Maryland (UMD), which involves a distributed archiving architecture with three geographic nodes at SDSC, NCAR, and UMD. Chronopolis is currently managing substantial collections from NDIIPP partners, including the California Digital Library (CDL) Web-at-Risk collection, the InterUniversity Consortium for Political and Social Research (ICPSR) collection, and collections from the Scripps Institution of Oceanography–Geological Data Center (SIO–GDC), and North Carolina Geospatial Data Archiving Project (NC State). ACE has been operational during the past six months within the Chronopolis environment. The current default ACE auditing policy is to audit files at the University of Maryland every 30 days. Table 1 illustrates the performance of a single audit manager on the collections audited at UMD, amounting to approximately 6 million files. A large fraction of the time is spent on accessing the collections across the network.

During the audit period on the CDL collection, a single audit manager was able to run at the rate of about 60 MB per second on average, almost fully utilizing the file transfer bandwidth available. For the other collections, where there were more small files, the audit speed was further limited by the overhead accessing each file. For example, on the ICPSR collection, the audit manager ran at the rate of 13 MB per second, having to open up each of about 4.8 million files. These results indicate that the actual time spent by an audit manager to perform the core audit process is negligible. It is small enough to be effectively hidden by the unavoidable overhead for accessing the collections. We note that multiple audit managers can be run concurrently on different collections to increase the performance almost linearly as necessary.

5 Conclusion

In this article, we presented a new methodology to address the integrity of long-term archives using rigorous crypto-

graphic techniques. Our approach depends only on the use of hash functions and linking schemes, and is independent of an external infrastructure such as PKI. The computational requirements of our approach are minimal and the overall solution can be implemented on any archive architecture. We built ACE as a complete prototype that executes this strategy and showed its effectiveness on large collections in Chronopolis. More details about ACE can be found in [18].

Acknowledgement This research study was supported in part by the National Science Foundation and the Library of Congress, grant number IIS-0455995, under the DIGARCH program.

References

1. Bairavasundaram, L.N., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Goodson, G.R., Schroeder, B.: An analysis of data corruption in the storage stack. ACM Trans. Storage **4**(3), 1–28 (2008). http://doi.acm.org/10.1145/1416944.1416947
2. Desmedt, Y.G., Frankel, Y.: Threshold cryptosystems. In: CRYPTO '89: Proceedings on advances in cryptology, pp. 307–315. Springer-Verlag, New York, Inc., New York, NY, USA (1989)
3. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theory **IT-22**(6), 644–654 (1976)
4. Farquhar, A., Martin, S., Boulderstone, R., Dooher, V., Masters, R., Wilson, C.: Design for the long term: Authenticity and object representation. In: Proceedings of Archiving 2005. IS&T, pp. 104–108 (2005)
5. Giuli, T.J., Maniatis, P., Baker, M., Rosenthal, D.S.H., Roussopoulos, M.: Attrition defenses for a peer-to-peer digital preservation system. In: ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005, pp. 163–178. USENIX Association, Berkeley, CA, USA (2005)
6. Haber, S., Kamat, P.: Content integrity service for long-term digital archives. In: Proceedings of Archiving 2006. IS&T, pp. 159–164 (2006)
7. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. J. Cryptol. **3**(2), 99–111 (1991)
8. Kaufman, C., Perlman, R., Speciner, M.: Network Security: Private Communication in a Public World. 2nd edn. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2002)
9. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Program Lang. Syst. **4**(3), 382–401 (1982). http://doi.acm.org/10.1145/357172.357176
10. Maniatis, P., Baker, M.: Enabling the archival storage of signed documents. In: FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies, p 3. USENIX Association, Berkeley, CA, USA (2002)
11. Maniatis, P., Giuli, T., Baker, M.: Enabling the long-term archival of signed documents through time stamping. Technical Report arXiv:cs.DC/0106058, Computer Science Department, Stanford University, Stanford, CA, USA (2001)
12. Maniatis, P., Roussopoulos, M., Giuli, T.J., Rosenthal, D.S.H., Baker, M.: The LOCKSS peer-to-peer digital preservation system. ACM Trans. Comput. Syst. **23**(1), 2–50 (2005). http://doi.acm.org/10.1145/1047915.1047917
13. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA (1996)
14. Merkle, R.C.: Protocols for public key cryptosystems. In: IEEE Symposium on Security and Privacy, pp. 122–134. (1980)
15. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (RAID). In: SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data, pp. 109–116. ACM Press, New York, NY, USA (1988). http://doi.acm.org/10.1145/50202.50214
16. Plank, J.S.: A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Softw. Pract. Exp. **27**(9), 995–1012 (1997)
17. Sivathanu, G., Wright, C.P., Zadok, E.: Ensuring data integrity in storage: techniques and applications. In: StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability, pp. 26–36. ACM Press, New York, NY, USA (2005). http://doi.acm.org/10.1145/1103780.1103784
18. Song, S., JaJa, J.: Ace: a novel software platform to ensure the integrity of long term archives. In: Proceedings of Archiving 2007, pp. 90–93. IS&T (2007)
19. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: EUROCRYPT. 19–35 (2005)
20. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: CRYPTO. 17–36 (2005)
21. Weatherspoon, H., Wells, C., Kubiatowicz, J.: Naming and integrity: self-verifying data in peer-to-peer systems. In: Future Directions in Distributed Computing, pp. 142–147 (2003)