Techniques to Audit and Certify the Long Term Integrity of Digital Archives[1]

(Technical Report UMIACS-TR-2007-38)

Sangchul Song and Joseph JaJa

Department of Electrical and Computer Engineering

Institute for Advanced Computer Studies

University of Maryland, College Park

Abstract

A large portion of the government, business, cultural, and scientific digital data being created today needs to be archived and preserved for future use of periods ranging from a few years to decades and sometimes centuries. A fundamental requirement for a long term archive is to set up mechanisms that will ensure the authenticity of the holdings of the archive. In this paper, we develop a new methodology to address the integrity of long term archives using rigorous cryptographic techniques. Our approach involves the generation of a small-size integrity token for each digital object to be archived, and some cryptographic summary information based on all the objects handled within a dynamic time period. We present a framework that enables the continuous auditing of the holdings of the archive, as well as auditing upon access, depending on the policy set by the archive. Moreover, an independent auditor will be able to verify the integrity of every version of an archived digital object as well as link the current version to the original form of the object when it was ingested into the archive. Using this approach, a prototype system called ACE (Auditing Control Environment) has been built and tested. ACE is scalable and cost effective, and is completely independent of the archive's underlying architecture.

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 the archive's holdings throughout the lifetime of the archive. 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 computer and networking security breaches.

[1] Supported in part by the National Science Foundation and the Library of Congress, grant number IIS-0455995, under the DIGARCH program

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 fire and floods. We also must be able to handle the possibility of security breaches and 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 of transformations may be applied to a digital object during its lifetime. For example, format obsolescence can lead to a migrative 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 integrity checking techniques, such as those described in [7, 11, 13, 15], have been described in the literature. However, all these techniques fall short of the requirements of a long term digital archive. Other techniques have been developed specifically for digital archive, including those that appeared in [3, 6, 9, 10, 19] , but none seems to offer a rigorous approach that is applicable to the different emerging architectures for digital archives (including centralized, peer to peer, and distributed archives) and that is capable to continually monitor and verify the integrity of the data in a cost effective way.

The main focus of this paper is to develop a cost effective methodology for ensuring the long-term integrity of digital archives. More specifically, we introduce efficient cryptographic techniques and related procedures to continually ensure the integrity of the various objects held in the archive. In fact, our methodology allows a third-party independent auditor to verify the integrity of every version of an archived digital object as well as link the current version to the original form of the object when it was ingested into the archive.

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.

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 [15]. 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 faulty device driver or a buggy file system which can cause data to become inaccessible [15]. 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 this method is easy to implement and can be effective, it is quite expensive in terms storage use and time spent for comparisons, and has serious limitations in a distributed environment.

A well known approach used in RAID storage is based on coding techniques, the simplest of which is *parity checking* [13]. 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 [13]. The XOR parity is a very special type of *erasure codes*, which can be much more powerful [14]. 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 data integrity on storage systems, they are not designed to support regular integrity auditing as they are computationally expensive, and can only detect (and correct) certain errors but not all.

Another 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 hashing 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 [7, 11]. 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. The most common hash functions used in practice are MD5, SHA-1, SHA-256, and RIPEMD-160, none of which can be shown to be one-way functions but all of which seem to work well in practice (in spite of the recent attacks that illustrated how to break MD5 [17] and SHA-1 [18]). In addition to the one-way assumption, a key assumption of this technique is that the hash values can be stored securely with absolutely no changes introduced to these values over time. Such an assumption may be reasonable for maintaining integrity over brief periods of time but, unless some additional mechanisms are used, this assumption does not hold for digital archives especially as the number of objects (and hence the number of hashes) continues to significantly grow over time. We will elaborate on this issue later.

Techniques for Digital Archives

We now describe the most important methods that have been suggested for integrity verification for digital archives. These methods heavily depend on the basic architecture, organization, and policies assumed for the digital archive, and hence a straightforward comparison is not possible.

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 some serious flaws for long term archives. Setting up a long term secure (centralized or distributed) registry that is expected to continually grow, and maintaining its integrity over time, is highly non-trivial. In fact, this is somewhat the problem we are trying to solve, except that it is slightly simpler as the hashes are

smaller and less complex than the objects themselves. Another problem with this scheme is the fact that cryptographic hashing schemes are based on the one-way function assumptions that may not hold over time for the particular hashing scheme used, and hence they may need to be replaced with more powerful schemes. There is a way to address this particular issue (to be described later) but it is computationally expensive.

Another simple 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 [10], which is 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 inter-cache protocol (LCAP) which is a cache auditing protocol. It runs LCAP continuously between 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 majority of the replies, then the object is believed to be intact. Otherwise, the content has been 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. This approach is tied to the LOCKSS inherent peer to peer distributed architecture and may not be possible to extend it to other architectures in a cost effective way. More importantly, the process is expensive and requires a significant communication overhead. Note that LOCKSS nodes can arbitrarily initiate auditing requests thereby tying up distributed resources in an unpredictable way. In fact, a compromised LOCKSS node can initiate a denial of service attack that will affect all the other nodes. One can develop security techniques to counter such attacks such as in [1, 4], but these techniques introduce an extra layer of complexity and additional costs. In general, achieving 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. One can make use of Byzantine agreement [8] strategies but these are computational expensive and difficult to implement in a cost effective way in an environment such as LOCKSS.

Another possible approach is to make use of *digital signatures* [2] 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 and threshold cryptography [8], which shed serious doubts on its practicality in a production environment. Also should the private key be compromised, the whole archive becomes at risk. Another potential problem with this scheme is its complete dependence on a third party, which may or may not exist over time, and on the secrecy of its private key. We note that schemes based on public-key cryptography are computational intensive, especially when extended to complex, large objects. A common

technique to reduce the cost is to compute a hash of the object, followed by a digital signature of the hash only.

We now introduce the time stamping technique. 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. It is clear that time stamping can be useful for the long term integrity of digital archives since our integrity notion assumes an auditable record of all the versions of the object along the temporal domain beginning from the time the object was deposited into the archive up to the present. However the temporal granularity is much coarser than that required for the common applications of time stamping. 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 [3] uses this strategy through an independent TSA. The verification procedure depends completely on the trustworthiness of the TSA. We mentioned above a number of significant problems with any approach that uses digital signatures, which show up in this scheme as well. A second approach, and the one used in our solution, is based on *linked (or chained) hashing* [6], which amounts to cryptographically chaining objects together in a certain way such that a temporal ordering among the objects can be independently verified. In the next section, we will describe this technique as used in our approach and illustrate the automatic verification and auditing procedures that will ensure the long term integrity of digital information in a cost effective way.

3. Our Approach

As can be seen from the previous section, the proposed integrity checking schemes revolves around the following techniques.

- Majority voting using replicated copies of the object or their hashes.
- Saving a digest ("fingerprint") of the 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 depends on the public key of the archive, and requires a fully trusted third party.

Our scheme is based on the notion of a cryptographic hash function and makes use of a third party that is not completely trusted, and hence can itself be audited as necessary by the archive or an independent party.

We start by introducing the 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 v, 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 next.

- *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 [11]). 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 Section 2, recent work has shown how to break the schemes based on MD5 and SHA-1, but the actual threat posed by such work is not clear and 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.

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 that 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 100KB), independent of the number of objects processed during the year. Given the small size of the witness values, they can be saved on read-only media such as CD-ROMs, and hence their integrity can be assured under reasonable assumptions about caring for the media and refreshing the content 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 CD-ROM. We next show how to counter this problem in a cost effective way.

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 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 following 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 [12], 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. 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 $o_1, o_2, o_3, ..., o_8$ (See Figure 1 for the corresponding tree).

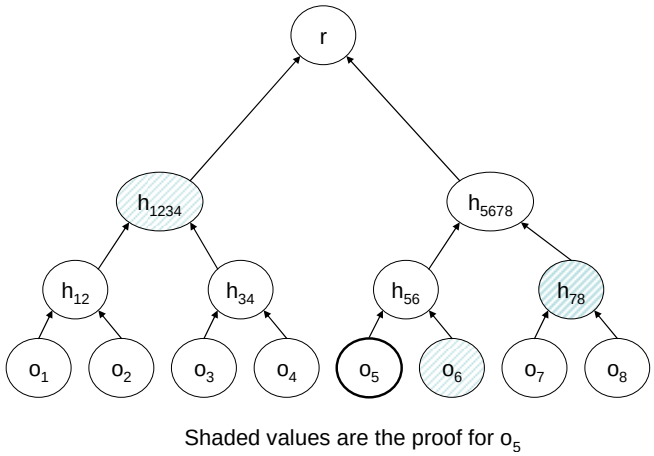

Figure 1. Merkle Tree

The values of the internal nodes are given by:

$$h_{12} = H(o_1 \| o_2); \quad h_{34} = H(o_3 \| o_4); \quad h_{56} = H(o_5 \| o_6); \quad h_{78} = H(o_7 \| o_8)$$

$$h_{1234} = H(h_{12} \| h_{34}); \quad h_{5678} = H(h_{56} \| h_{78})$$

$$r = H(h_{1234} \| h_{5678})$$

The proof of the object whose digest value is o_5 will be the following

$\{(o_6, r), (h_{78}, r), (h_{1234}, l)\}$, where *r* designates right sibling and *l* left sibling.

Given the digest of an object and its proof, we can quickly compute the round hash value by following the path defined by the proof and performing the concatenation/hash operations as appropriate. 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. Therefore, the integrity of an object can be verified effectively using the proof under the assumption that a valid copy of the round hash value was preserved somewhere. One can in principle store these round hash values on CD-ROMs, especially if we can make each round covers a whole day. However auditing an object may take an inordinate amount of time, especially that we want to allow auditing upon access of the objects as necessary. Also, the size of the corresponding proofs will become significant when the number of objects processed becomes very large. To get around these issues, we re-iterate the process by compressing the ordered set of round hash values using one of the hash linking schemes such as Merkle's tree. 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. Once determined, the witness value is stored on a read-only medium such as a CD-ROM. It also can be published on the web as a

widely observed witness. As before, we define the proof of a round hash value *r* to be the sequence of siblings in the directed graph used by the hash linking scheme on the unique path from the leaf corresponding to *r* to the root of the directed graph. In addition to storing the witness values on CD-ROMs, we assume that the round hash values are available online through a third party for use in auditing the objects of the archive. As we will show later, we don't have to completely trust this third party as its contents can be audited using the witness values. Note that the temporal ordering of the rounds as well as the objects within a round cannot be tempered with because we are using a hash linking scheme based on a one-way, collision resistant hash function (which can be different between different rounds).

Before describing the verification and auditing procedures, we summarize the types of integrity information generated by our approach. For each object, an *integrity token* is generated, which contains the proof that leads from the object's digest to the round hash value corresponding to the round in which the object participated, and the time stamp (or a sequential number) of the round. The size of the integrity token is small and depends on the logarithm of the number of objects processed during a round. A daily hash value (witness) is generated from all the round hash values processed during the day. Having created the intermediate round hash values, we assume that these values are available online for auditing purposes. We will later briefly describe our ACE (Auditing Control Environment) system that includes an independent service, which generates the integrity tokens and the witness values, and which, in addition, maintains the intermediate round hash values.

Verification and Auditing Procedures

Given an object, we can verify its integrity as follows. We start by computing the digest of the object, followed by computing the corresponding round hash value using the proof stored in the integrity token of the object. If that value is equal to the stored value for the round, the object has not been changed since the last audit. If there is any question about the stored value of the round, we can use the stored round hash values for the day to compute the witness value, and compare that value with the value stored on the CD-ROM. If the two values are equal, we are certain that the stored round values are correct since we are assuming that the witness values are always correct.

Note that the above procedure can also be used by a third-part independent auditor to certify the integrity of any object in the archive.

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 [5]). Such a solution will ensure our ability to verify the integrity of the object since its ingestion into the archive as articulated in earlier work. This process increases the size of the integrity token, but has no impact on the sizes of the other integrity components. In order to accommodate this update scheme, an integrity token can also contain a link to the previous integrity token, if any.

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 is

computationally demanding and too complicated to be of practical use. We make the very reasonable assumption that an archive has to preserve all the versions of an object for otherwise we won't be able to certify the authenticity of the object since its initial ingestion into the system. Hence a transformation will lead to a new version of the object, which will then participate in a hashing round to obtain its new cryptographic information using the same method as before. Different versions can be linked through the global identifier of the object, and hence it is possible to verify the integrity of all the versions of each object starting with the current one and ending with the first version ingested into the archive. 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. We may want to include the version number in the integrity token to simplify the process of linking the integrity tokens issued to the objects with different versions.

4. Putting the Ideas Together – The ACE Tool

Using the ideas described in the previous section, we present in [16] an early implementation of the ACE (Auditing Control Environment) prototype system. Another version that incorporates several significant changes will be released soon. Here we present a brief overview of the ACE architecture and illustrate its auditing processes.

ACE consists of two major components: the first, called AIMS (ACE Integrity Management System), is a third-party service provider that generates the integrity tokens upon the request of an archive. It also maintains the round hash values and publishes 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 AIMS and the archive. In particular, the AM sends requests to AIMS to generate the integrity tokens for a number of objects, and once received the AM stores the tokens in a local registry. Figure 2 shows the overall ACE architecture of the general case of a distributed archive.

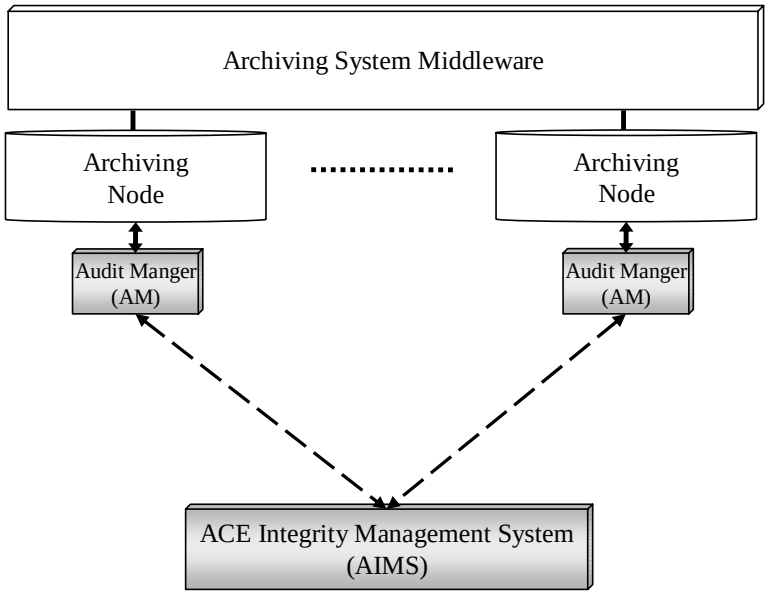

Figure 2. ACE Architecture

To audit an object (which can happen either upon accessing the object or during a regular periodic auditing process set by the archive), the AM computes the digest of the object followed by retrieving its integrity token. Using the digest and the proof in the integrity token, it computes the round hash value. It then requests that value from AIMS. If the two values agree, the object is intact. If there is any reason to question the integrity of AIMS, the AM can request the proof for the corresponding round hash value. AIMS has to aggregate all the round values for the day to determine the proof, and return the corresponding value to the AM. Using the proof, the AM can compute the corresponding witness value and compare it to the value stored on the CD-ROM. AIMS can be trusted only if the two values are equal.

In terms of performance, we have evaluated ACE using the NARA EAP (The National Archives Electronic Access Project) Image Collection consisting of approximately 130,000 files of total size over 1.1TB. We were able to fully audit all the objects in about 15 hours while storing the data remotely on a separate server. Most of the time was spent in moving the data between the separate machines in our environment. We expect performance to be much better in a production environment since all the data movement will be carried out locally between the audit manager and the local storage.

5. Conclusion

In this paper, we presented a new methodology to address the integrity of long-term archives using rigorous cryptographic 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. More details about ACE can be found in [16].

6. References

[1] T. Aura, P. Nikander and J. Leiwo, *DOS-Resistant Authentication with Client Puzzles*, Lecture Notes in Computer Science, 2133 (2001), pp. 170+.

[2] W. Diffie and M. E. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory, IT-22 (1976), pp. 644-654.

[3] A. Farquhar, S. Martin, R. Boulderstone, V. Dooher, R. Masters and C. Wilson, *Design for the Long Term: Authenticity and Object Representation*, *Proceedings of Archiving 2005*, IS&T, 2005, pp. 104-108.

[4] T. J. Giuli, P. Maniatis, M. Baker, D. S. H. Rosenthal and M. Roussopoulos, *Attrition defenses for a peer-to-peer digital preservation system*, *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005*, USENIX Association, Berkeley, CA, USA, 2005, pp. 163-178.

[5] S. Haber and P. Kamat, *Content Integrity Service for Long-Term Digital Archives*, *Proceedings of Archiving 2006*, IS&T, 2006, pp. 159-164.

[6] S. Haber and W. S. Stornetta, *How to Time-Stamp a Digital Document*, Journal of Cryptology, 3 (1991), pp. 99-111.

[7] C. Kaufman, R. Perlman and M. Speciner, *Network security: private communication in a public world*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[8] L. Lamport, R. Shostak and M. Pease, *The Byzantine Generals Problem*, ACM Trans. Program. Lang. Syst., 4 (1982), pp. 382-401.

[9] P. Maniatis, T. Giuli and M. Baker, *Enabling the Long-Term Archival of Signed Documents through Time Stamping*, Computer Science Department, Stanford University, Stanford, CA, USA, 2001.

[10] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal and M. Baker, *The LOCKSS peer-to-peer digital preservation system*, ACM Trans. Comput. Syst., 23 (2005), pp. 2-50.

[11] A. J. Menezes, S. A. Vanstone and P. C. V. Oorschot, *Handbook of Applied Cryptography*, CRC Press, Inc., Boca Raton, FL, USA, 1996.

[12] R. C. Merkle, *Protocols for Public Key Cryptosystems*, *IEEE Symposium on Security and Privacy*, 1980, pp. 122-134.

[13] D. A. Patterson, G. Gibson and R. H. Katz, *A case for redundant arrays of inexpensive disks (RAID)*, *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, ACM Press, New York, NY, USA, 1988, pp. 109-116.

[14] J. S. Plank, *A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems*, Software – Practice & Experience, 27 (1997), pp. 995-1012.

[15] G. Sivathanu, C. P. Wright and E. Zadok, *Ensuring data integrity in storage: techniques and applications*, *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, ACM Press, New York, NY, USA, 2005, pp. 26-36.

[16] S. Song and J. JaJa, *ACE: A Novel Software Platform to Ensure the Integrity of Long Term Archives*, *Proceedings of Archiving 2007*, IS&T, 2007, pp. 90-93.

[17] X. Wang, Y. L. Yin and H. Yu, *Finding Collisions in the Full SHA-1*, *CRYPTO*, 2005, pp. 17-36.

[18] X. Wang and H. Yu, *How to Break MD5 and Other Hash Functions*, *EUROCRYPT*, 2005, pp. 19-35.

[19] H. Weatherspoon, C. Wells and J. D. Kubiatowicz, *Naming and integrity: Self-verifying data in peer-to-peer systems*, *Future Directions in Distributed Computing: Research and Position Papers*, Springer-Verlag Berlin, Berlin, 2003, pp. 142-147.