A distributed Integrity Catalog for digital repositories

Nikos Chondros
University of Athens

Mema Roussopoulos
University of Athens

Abstract

Digital repositories, either digital preservation systems or archival systems, periodically check the integrity of stored objects to assure users of their correctness. To do so, prior solutions calculate integrity metadata and require the repository to store it alongside the actual data objects. This integrity metadata is essential for regularly verifying the correctness of the stored data objects. To safeguard and detect damage to this metadata, prior solutions rely on widely visible media, that is unaffiliated third parties, to store and provide back digests of the metadata to verify it is intact. However, they do not address recovery of the integrity metadata in case of damage or attack by an adversary. In essence, they do not preserve this metadata.

We introduce IntegrityCatalog, a system that collects all integrity related metadata in a single component, and treats them as first class objects, managing both their integrity and their preservation. We extend a persistent, authenticated search tree to become an authenticated dictionary for arbitrary length key/value pairs, which we use to store all integrity metadata, accessible simply by object name. Additionally, IntegrityCatalog is a distributed system that includes a network protocol that manages both corruption detection and preservation of this metadata, using administrator-selected network peers with two possible roles. Verifiers store and offer attestation on digests and have minimal storage requirements, while preservers efficiently synchronize a complete copy of the catalog to assist in recovery in case of detected catalog compromise on the local system. We describe our prototype implementation of IntegrityCatalog, measure its performance empirically, and demonstrate its effectiveness in real-world situations.

1 Introduction

Digital repositories, either digital preservation systems or simply archival systems, store a series of objects for the long term. The former, in particular, are expected to preserve their contents for centuries. A user of such a system can request a stored object a long time after its ingestion into the archive, and expect to obtain an intact copy of it.

There are basically two challenges a digital repository has to tackle, to provide assurances to its users regarding the integrity of its stored objects. The first is the physical degradation of the storage media, disk drives typically, that can result in bit rot and silent corruption. The second is an adversary, external or internal, that wishes to silently alter the content of the stored objects, in an attempt, for example, to rewrite history.

These challenges have been studied and solutions have been proposed, e.g. [14, 35, 10, 12, 24]. Many of these solutions generate integrity information for each object (such as a digest produced by a hash function), collect it in a separate file (integrity metadata file), and require the digital repository to store this extra file alongside the actual object. The system then verifies the integrity of the data objects stored by calculating fresh hashes of the objects and comparing them with the associated metadata file. To ensure that the integrity metadata is itself tamper-evident, these systems aggregate a series of such metadata files into an authenticated data structure and publish a digest of this structure in a widely visible medium, such as a newspaper or a Usenet newsgroup.

This approach is insufficient for two reasons. First, introducing the use of integrity metadata to verify and preserve data objects creates the recursive issue that integrity metadata must itself be preserved too. Thus, it is not enough to simply make it tamper-evident, there must be a mechanism to recover the damaged, compromised, or deleted integrity metadata too.

Second, offloading integrity digests to third parties
(e.g., Usenet) is not future-proof. In the long run, the organization storing the integrity metadata summaries may go out of business, become unavailable, or simply stop cooperating. For example, in the case of Usenet, there is no long-term guarantee that the company storing and maintaining the Usenet newsgroup now will continue to store integrity token summaries intact forever. To truly safeguard this metadata summary, one must distribute this information to multiple nodes, ideally in separate administrative domains, and develop a protocol that manages the distribution of integrity summaries.

In this paper, we describe the design, implementation, and evaluation of IntegrityCatalog, a system that can be integrated into any digital repository. IntegrityCatalog takes a holistic approach. It treats integrity metadata as first-class objects and serves as a tool that ensures that the integrity of the metadata themselves is preserved, verified regularly, and recovered in case of damage or adversarial attack.

IntegrityCatalog stores and manages all integrity metadata indexed and accessible by object name, alleviating the need for the digital repository to maintain and manage extra information alongside its data objects. It uses an authenticated data structure for its storage needs that is persistent and tamper-evident. Additionally, IntegrityCatalog is a distributed system that includes a network protocol that manages both corruption detection and recovery of integrity metadata using a set of administrator-selected network peers. These peers have two possible roles. Verifiers store and offer attestations on integrity tokens and have minimal storage requirements, while preservers efficiently synchronize a complete copy of the Catalog to assist in recovery in case of catalog compromise on the local system. This approach allows careful selection of the entities the repository administrator entrusts to serve as verifiers and preservers. The repository administrator has the flexibility to choose as many remote network peers as possible to serve as verifiers and preservers, to minimize the risk of a powerful adversary compromising the majority of peers, thus enforcing use of corrupt integrity information. The administrator can also change the selection of network peers over time as they become unavailable.

To implement IntegrityCatalog, we create a new authenticated data structure, called RBB-Dict by extending the RBB-Tree, introduced by Maniatis et al. 23, to become a complete authenticated dictionary for variable length key/value storage. This data structure is persistent, as it allows snapshots (of integrity metadata in our case) to be taken at arbitrary intervals and ensures that values in previous snapshots remain intact for the lifetime of the tree.

Our approach is simple, efficient and robust. It is simple because in contrast to existing solutions, it collects all integrity information into a single (but persistent) hash tree. It is efficient, because IntegrityCatalog verifies the complete data structure once, by going out on the network to verify its root digest, and then provides unlimited searches of digests within the Catalog for verification of stored data objects without requiring additional network communication. It is robust because the catalog is never used without prior verification, a task which is distributed by design to avoid single points of trust and failure.

In summary, the contributions of this paper are:

- We describe the design and implementation of IntegrityCatalog, a system that can be used as a tool by digital repositories to protect, preserve, and recover integrity tokens in case of damage or adversarial attack.
- We describe the design and implementation of a protocol for efficiently distributing IntegrityCatalog snapshots to enable preservation and verification of integrity tokens at remote nodes.
- We empirically measure the performance of our IntegrityCatalog prototype and show that insertions into the catalog and searches/retrivals of object digests from the catalog are efficient. Our experiments show >1K insertions per second, and search throughput of 20K queries per second.

2 Background

Cryptographic hash functions 27 25 map a message of arbitrary length to a fixed size digest, with properties such as pre-image and collision resistance. Authenticated data structures use cryptographic hash functions to represent a data collection with a fixed-size authenticator. A proof for a membership claim, along with this authenticator, can be used by an external entity to verify this claim.

Merkle introduced the hash tree (or Merkle-tree) in 26, which is thought to be the first authenticated data structure. Here, the digest of each data element is hashed to produce a leaf node of a balanced tree with a fixed fan-out. The inner nodes of the tree are the result of hashing the digests produced at the lower level. With this organization, the root digest identifies the entire collection of data items uniquely (it is the authenticator), with the assumption that the above properties hold for the cryptographic function used to produce the digests. The proof for a membership claim consists of all digests needed to re-produce the root digest, in the path from the data element in question to the root.

Buldas et al. later introduced the Authenticated Search Tree (AST) 3, 4, which is a descendant of the hash tree. Here, all nodes (inner and leaves) contain data elements.
In this layout, the digest of any given element is produced by hashing the data element of the node as well as the digests of its children. This time the existence proof contains data elements along with the needed digests in the path from the element in question (which need not be a leaf node any more) to the root.

Both the Merkle-tree and the AST data structures are designed without secondary memory in mind. Moreover, there is no concept of persistence; that is, the ability to go back in time and see the version of the data collection that existed back then.

Maniatis et al. introduced the RBB-Tree in [23][22] to address some of these issues. RBB-Tree is an authenticated search tree, organized externally as a B-Tree but having each data page further organized as a Red-Black balanced binary tree. This layout has two benefits. First, the data structure is secondary-storage friendly because of its B-Tree organization. Second, an existence proof is minimized in size as it does not need all the data elements of all B-Tree pages from the current node to the root, but only the data elements in the path from the embedded Red-Black tree. Additionally, it uses snapshots to isolate older pages from their changed descendants, giving the ability to search the data collection in the exact structure as it was when the snapshot was taken. As an example, Figure 1 depicts the status of the tree when three snapshots are taken, where the first added items B,C,D, the second items E,F, and the third item A. As shown, page 4 which was formed during the epoch of snapshot 2, is still used in snapshot 3 as it is unchanged.

Figure 1: RBB-Tree example

3 System requirements

Integrity metadata is an essential component of many digital repositories as they perform verification of stored data objects by periodically calculating the hashes of these objects and comparing them with the associated integrity tokens. As such, we treat integrity metadata as first class objects and propose IntegrityCatalog as a tool that ensures that the integrity of the metadata themselves is verified regularly, preserved, and recovered in case of damage or adversarial attack.

The design of IntegrityCatalog system must fulfill the following requirements:

- the system should provide a succinct representation of all data objects in the digital repository and assist the repository in regularly verifying the integrity of the stored data objects
- the system should not require the digital repository to maintain any additional information, thus it should allow access to integrity metadata by object identifier
- the system should manage and store all integrity metadata in a persistent, authenticated, and tamper-evident manner
- the system should be efficient; insertion into and retrieval of metadata from the Catalog should be fast, as digital repositories often exhibit high ingress rates
- the Catalog should not be stored solely on the repository as an attacker could modify data objects and integrity metadata to match, and thus alter the content of data objects without being detected; the system administrator should be able to select the peers to replicate into
- to avoid dependence on a single point of trust or administration, digests of the Catalog should be distributed to multiple nodes regularly and in an efficient manner
- the system should not require storage of long-term secrets, making it suitable for use in digital preservation systems
- in case of damage to Catalog contents from physical storage degradation and/or adversarial attack, the system should be able to recover the integrity metadata to a prior consistent state

4 System description

4.1 IntegrityCatalog overview

IntegrityCatalog enables a node to manage the association of a locally stored object (with a unique identifier) with the digest of its contents. The collection of associations between locally stored objects and their digests forms a persistent hash tree that can be represented by an authenticator, its root digest.
The contents of this collection are protected by taking snapshots at frequent intervals and publishing a succinct token, representing the entire collection of locally stored objects, to administrator-selected peers on the network. The form of this token is \(<\text{snapshot}_\text{id}, \text{root}_\text{digest}>\), where \(\text{snapshot}_\text{id}\) is an ever increasing number identifying the current snapshot, and \(\text{root}_\text{digest}\) is the digest of the root element of the hash tree. Some of these peers, pre-selected by the administrator, are expected to further contact the local node and preserve the full contents of IntegrityCatalog for future recovery in case of damage or attack. The system is optimized to minimize the interactions for preservation to a minimum, by allowing only pages modified since the last preservation operation to be transferred.

The host system using IntegrityCatalog is expected to verify the catalog once, before using it to access object digests. At this point IntegrityCatalog will contact its peers and verify the root digest of the latest snapshot, thus certifying the entire collection is intact. From this point on, IntegrityCatalog provides verified replies without any further external network communication.

4.2 Design details

IntegrityCatalog requires two lists of peers to be defined for its operation. The first one is the \(\text{verifiers}\) list, containing nodes that are willing to store and later provide back the authenticator of each snapshot of the catalog. This information is small (a few dozens of bytes for each snapshot) imposing minimal storage requirements to nodes with the role of verifier.

The second one is the \(\text{preservers}\) list, which should be a subset of the \(\text{verifiers}\), and contains nodes willing to store and potentially provide back the full contents of the catalog itself. As this is an ever growing data structure, its storage requirements are more significant than the ones imposed on the \(\text{verifiers}\) for the storage of the snapshot authenticators.

Finally, if the local system is to become a \(\text{preserver}\) for another node, a third list should be constructed, called \(\text{preservees}\), with nodes for which such a preservation agreement exists. In section 6 we provide estimates for the space requirements for \(\text{preservers}\) and \(\text{verifiers}\).

The choice of nodes to act as \(\text{preservers}\) and \(\text{verifiers}\) is left to the administrator of the digital repository. Administrators have the flexibility to choose network nodes they trust and to change the set of selected nodes over time in accordance with the repository’s security policy.

The host system (i.e., the digital repository) can ask the catalog to store the digest of an object. This digest is expected to be calculated and formatted by the host system itself. The information given to IntegrityCatalog is an identifier to serve as a key and a free format sequence of characters (containing the digest) to serve as a value. The identifier is again treated as an arbitrary string, which may contain an object name, an object version or anything else appropriate for the underlying system.

The host system can terminate the current epoch of the catalog at intervals it defines itself (e.g., every day). This is achieved by invoking the \(\text{sealCatalog}\) operation. This operation takes a snapshot of the catalog and produces a new \(<\text{snapshot}_\text{id}, \text{root}_\text{digest}>\) pair. It then proceeds to publish this pair to the nodes in the \(\text{verifiers}\) list and waits for their acknowledgments. Once enough nodes have stored this information, the system considers the \(\text{sealCatalog}\) operation finished and notifies the host application about the result of the operation. In case of error, the host application is instead notified with the failure.

We expect the host system to use IntegrityCatalog to help it check the integrity of its locally stored objects periodically. To do so, it requests the stored integrity information and compares it with freshly recalculated hashes from the stored object data. Before reading the catalog, it is expected to invoke once the \(\text{verifyCatalog}\) operation. This operation requests the nodes in the \(\text{verifiers}\) list to transmit back the latest \(<\text{snapshot}_\text{id}, \text{root}_\text{digest}>\) pair each one has stored for the current node. IntegrityCatalog gathers all messages and tallies the replies by \(\text{snapshot}_\text{id}\) and \(\text{digest}\). It makes sure a minimum quorum of \(\text{verifiers}\) have replied, and tests that a winning threshold of \(\text{verifiers}\) have sent replies matching the local latest known snapshot’s digest (see sections 4.3 and 5.1 for details).

If the result of verification is successful, the host system can repeatedly query the catalog for the digest of any stored object to verify its integrity, without generating any further network communication. Before any reply to such queries is provided, IntegrityCatalog verifies it produces the same (already verified) root digest of the current snapshot, by rebuilding this root digest from the leaf containing the node all the way up to the root. This way, any tampering of the stored data pages of the catalog itself will be detected.

If, however, the verification of the catalog fails, the host system is expected to initiate the \(\text{recoverCatalog}\) operation, during which IntegrityCatalog contacts all nodes in the \(\text{preservers}\) list to achieve consensus on the latest available version of the catalog, in a manner similar to \(\text{verifyCatalog}\) above. Once IntegrityCatalog decides on a version, it selects at random one of the nodes maintaining the version and requests the initiation of a binary update. IntegrityCatalog then restores the catalog from the selected node, one snapshot at a time. When finished, it notifies all nodes in the \(\text{verifiers}\) list that the latest version of its catalog has been reset to the restored one. At this
point, the host system is expected to iterate over all data objects, identify the objects missing from the restored catalog and register them. The objects in question are the objects ingested in the local repository after the restored snapshot of the catalog was taken. This function of the host system can be summarized as follows:

```python
for each object in repository
    hash = catalog.get(object.name)
    if hash is null
        hash = calculateHash(object.data)
        catalog.put(object.name,hash)
```

Every digital preservation system that uses hash functions for verifying integrity must address the fact that eventually the specific hash function used will be phased out, hopefully before it becomes compromised. To facilitate this, while also guaranteeing that an attacker cannot replace the existing digest of an object completely, IntegrityCatalog provides for an `amend` operation. This operation allows the host system to append a new value to the existing one for a specific object. The result of this function is that the object is associated with the concatenation of the old digest and the new digest supplied. It is expected that the host system will apply some structure to the values it puts in the catalog so that, over time, it can make sense of extracted information. For example, it can prefix the hash function name to the actual digest and append a delimiter, such as 'md5:XXXXXX/sha-256:ZZZZZZ'. In any case, the format of the value attached to each object identifier is left completely to the host application. IntegrityCatalog simply does not allow a value to be overwritten, but only amended.

In case of object migration, due for example to format change, our system expects the object to be re-registered, with a new version identifier. This version identifier is expected to be part of the object name, thus creating a new entry in our search tree. The idea is that all different formats exist in the catalog under different names, and most probably in different catalog snapshots.

### 4.3 Assurances

The use of IntegrityCatalog, besides simplifying the management of integrity information in a digital repository, also provides the following assurances.

If object data is corrupted, but catalog data is not, the corruption will be detected in the next integrity check performed by the repository.

If catalog data gets corrupt (arbitrary data pages get corrupted or purposefully modified but without propagating the change in the hash tree), this corruption will be detected on the first catalog search operation that accesses the corrupt data pages, raising an error. This detection will happen either because the page will be damaged and the internal tree algorithms will fail, or because the validation of the root digest, performed by IntegrityCatalog before returning the result, will fail. At this point, we expect the host system to initiate a `recoverCatalog` operation to restore an intact copy of the catalog.

If IntegrityCatalog’s data is modified in a non-destructive way (from an attacker, maintaining the hash tree structure), its root digest will change (assumption from the definition of cryptographic hash functions). This change will be detected in the next `verifyCatalog` operation from the peers’ attestation.

We assume that remote nodes can display arbitrary behavior and the attacker may control and coordinate some of these nodes. Depending on the policy selected by the administrator of the digital repository, different assurances can be achieved from our network protocol. The `verifyCatalog` operation collects tuples of the form `<snapshot_id, root_digest, number_of_votes>`, which are passed along the current `<snapshot_id, root_digest>` data from the catalog’s tree to a policy-based checker.

An attacker wishing to enforce an illegitimate change in an object, will first need to break into the digital repository and alter the object’s contents and the catalog’s corresponding entries. Additionally, he will need to control enough nodes in the `verifiers` set that will: a) force a big enough quorum, to pass the quorum test, and b) force an altered catalog’s authenticator to win the voting. That is, he will need to ensure the following two inequalities hold:

\[
\frac{verifiers - absentees}{verifiers} > \text{quorum}\% \quad (1a)
\]

\[
\text{subverted} > \text{winning}\% \times (\text{verifiers} - \text{absentees}) \quad (1b)
\]

where `absentees` represents the number of faulty non-participating nodes, `subverted` is the number of faulty participating nodes, `quorum\%` and `winning\%` are the percentages selected by the current policy, and `verifiers` is the number of verifiers. Inequality (1a) holds when the attacker passes the quorum test (we assume he can prevent nodes from participating in the voting process), while (1b) holds when he controls the majority of votes. Concluding, as long as the attacker cannot enforce inequality (1b) above, he cannot force the use of corrupt catalog information from network peers.

### 4.4 Limitations

Even with all the assurances IntegrityCatalog provides, two windows of vulnerability for an attacker to modify an object’s content and go unnoticed still remain. The first one is on ingress, while the current catalog epoch is still open. Before taking the snapshot and publishing the
catalog’s authenticator to its peers, an attacker that has
taken over the local node can alter both the object and its
digest stored in IntegrityCatalog (e.g., via direct binary
modification of the catalog’s on disk image). One solu-
tion for this is to shrink the window between snapshots,
at the expense of increasing the amount of network traf-
cic as well as the storage requirements for the nodes in
the verifiers list.

Additionally, an adversary can attack immediately af-
after a recovery of the catalog. Assuming an object was
initially registered in a recent snapshot, and this snapshot
has not yet propagated to the preservers, the attacker can
corrupt the catalog and wait for the recovery of a pre-
vious version of it, where the target object was not yet
included. The attacker can then modify the object be-
fore the host system re-registers it in the catalog; this will
result in the object being registered with the wrong (al-
tered) contents. This way however, the attacker can only
target fairly recent objects. Otherwise, he would need to
control many nodes in the preservers list to be able to en-
force a version of the catalog old enough to not include
an older object.

5 Implementation

The IntegrityCatalog prototype is implemented in Java.
The only external library it uses is Oracle BerkeleyDB,
as a back-end for our authenticated search tree (detailed
in section 5.2 below).

5.1 IntegrityCatalog

IntegrityCatalog exposes a single class named Cata-
log that provides access to all described functionality
via its exposed methods. The basic operations, such
as put(key, value), amend(key, new_value), and
get(key) are self explanatory. We now describe the
seal, verify, recover, and preserve methods.

seal: The seal() method initiates the asynchronous
sealCatalog operation. Initially, it terminates the cur-
et epoch of the catalog, deriving a new snapshot. Af-
fer that, it sends a StoreMessage to each Node in the
verifiers list, carrying the new snapshot id and its root
digest. Finally, it sets a timeout to limit the time win-
dow for arrival of the replies. Each receiving Node
is expected to store this information and reply with a
StoreReplyMessage. Additionally, if the verifier node
is also a preserver, it is expected to schedule a remote
catalog preservation asynchronous operation (explained
in detail below) to preserve the current version of the cat-
alog. Each StoreReplyMessage that arrives at the cur-
rent node is collected; once all nodes have replied or the
timeout has passed, the system counts all positive replies
and decides on the success of the operation, based on
the defined quorum percent. The function used to decide
the result is replaceable by the host system and, as such,
a custom policy may be implemented. Finally, the host
system is notified with the seal operation’s result.

verify: The verify() method is the entry point
for the verifyCatalog asynchronous operation. The
host application is expected to call it before loop-
ting through its local objects and accessing the get() function repeatedly to perform integrity checking of
its locally stored data objects. Initially it sends a
StoredVersionRequestMessage to each Node in the
verifiers list and sets an appropriate timeout. Each
receiving node is expected to access its local reg-
istry of stored <snapshot_id, root Digest> to-
kens for the requesting catalog and reply with the
most recent one via a StoredVersionReplyMessage.
The local node collects these replies, aggregates them
by <snapshot_id, root digest> and passes them to
the policy based function, along with the current
tree snapshot and digest. This function will
reply with either success or failure. Our default
policy decides success when the quorum is greater
than 50% of the number of verifiers, and the current
<snapshot_id,root digest> has received a mini-
um of 70% of the votes of this quorum. Again, the
host application is notified with the result.

recover: The recover() method initiates the recov-
erCatalog asynchronous operation. The host applica-
tion should invoke it only when the catalog is consid-
ered damaged, as it will result in the complete replace-
ment of the local catalog with the latest available ver-
sion from one of the preserving nodes. This operation
takes place in three consecutive asynchronous phases.
In the first phase, a RecoverVersionRequestMessage
is sent to each Node in the preservers list. Each rec-
eiving node is expected to access the actual catalog it
preserves for the calling node, obtain the latest avail-
able snapshot id and its accompanying root digest and
send it via the RecoverVersionReplyMessage. The
originating node will gather all replies and decide on
the latest version to restore, in a manner similar to the
one described in the verify function. Our default im-
plementation will choose the version with more than
70% of the replies, with a minimum quorum size of
50% of the number of preservers. It will then pick at
random one of the nodes maintaining the selected ver-
sion and start the second phase, the actual catalog re-
covery. It will send a RecoverBeginMessage to the
selected node, starting with the first snapshot id, while
also recreating the local catalog from scratch. The re-
eceiving node will retrieve the first available block for
the given snapshot of the preserved remote tree and re-
ply with it via a RecoverDataMessage. The local node
will update its catalog with the received data and then
continue with a RecoverGetNextMessage until it receives a RecoverEndOfData message. At this point, it will increase the snapshot id and restart this phase until the catalog arrives at the target snapshot id. Once this goal is achieved, at phase three, it will verify the catalog’s current authenticator against the expected one and send a VersionResetMessage to all verifiers, instructing them to force the latest version to become the one just restored, regardless of each node’s local registry.

preserve: As already stated, the remote catalog preservation process is initiated when a remote node (called preserver) receives a StoredVersionRequestMessage (that is, it is a member of the verifiers list of the originating node) and finds the source node (called origin) to be in its preservers list (that is, node preserver is also a member of the preservers list of the origin). In this case, preserver will open the catalog stored locally that corresponds to origin and retrieve the latest snapshot id. It will then proceed to send an UpdateBeginMessage to origin requesting the succeeding snapshot from the discovered one. Origin will process this message and reply with an UpdateDataMessage with the contents of the corresponding page of its catalog. Preserver will then send an UpdateGetNextMessage until it receives an UpdateEndOfDataMessage from origin. Finally, preserver will repeat this process until the target snapshot id has been derived.

5.2 RBB-Dict

To meet the above functionality we developed a new data structure called RBB-Dict that leverages the RBB-Tree data structure, which provides a persistent, authenticated, tamper-evident, disk-based, search tree. It is persistent because it takes snapshots of its content and makes sure old data is never modified. It is authenticated because it forms a hash tree where the root digest serves as an authenticator for the entire collection. It is tamper-evident because this authenticator can be used to prove that a data element is indeed part of the collection by rebuilding the root digest from the path from the root to the data element in question. Finally, it is disk-based because it uses the secondary storage friendly B-Tree structure to manage its blocks.

We chose to leverage RBB-Tree, because we needed an authenticated search tree whose digest is efficient for storage and network communications, and also permits efficient searches. However, because it was designed in another context, there were shortcomings as well. The existing version is a search tree of keys only; there is no notion of associating a value with this key. It was designed to basically answer membership queries, i.e. if a key is a member of the collection or not, and to provide efficient proofs for such claims. Additionally, the existing implementation manages fixed size data elements only, which is too limiting in our context. The reason is, our keys (object names) are variable sized by nature, while our values (integrity metadata) are expected to grow over time as, for example, hash functions become obsolete and get replaced by newer ones. Finally, the issue of efficiently synchronizing different versions of the same tree over the network (thus slow data links) had not been tackled.

We created RBB-Dict by modifying the RBB-Tree concept and implementation to address these shortcomings. Our first enhancement over the previous work is the conversion of the search tree to a dictionary of arbitrary length key/value pairs. We include the value part of the data element in the calculation of the element’s digest (and consequently of the root digest as well), making sure that the value is also tamper evident. As in the RBB-Tree, all searches are performed with the key only, but RBB-Dict slightly modifies the semantics of the basic membership operation. The original semantics are:

\[ k \in T \rightarrow \text{boolean} \]

where \( T \) is the tree representing the dictionary and \( k \) is the key in question. The boolean result is either true when the key is a member of the collection and false otherwise. The new semantics are:

\[ k \in T \rightarrow (\text{boolean}, \text{value}) \]

where the value of the designated key is returned only when the result is true and is undefined otherwise. We have extended the existing RBB-Tree implementation with functionality to allow for variable length storage of both keys and values, by modifying the block management layer, as well as the tree structure accordingly. We utilize Oracle BerkeleyDB as a storage back-end for the new variable length data pages.

Our second enhancement is the introduction of an update operation to allow for the value corresponding to a key to change over time; the original RBB-Tree is defined as an append-only data structure. Such a change operation leaves the existing value intact in all previous snapshots, while modifying it with the new one in the current epoch and onward.

Our third enhancement is to enable efficient synchronization of a tree with an existing previous version of it. This allows us to propagate IntegrityCatalog’s underlying data structures to selected peers on the network efficiently, as we already described in the network operations of Section 5.1. The RBB-Dict interface includes the following methods to allow for this synchronization:

- getFirstBlockOfSnapshot(long snapshot) →<blockNumber, blockSize>
• getNextBlockOfSnapshot(long snapshot, long blockNumber) \rightarrow \langle\text{blockNumber}, \text{blockData}\rangle
• binaryUpdateBegin(long snapshot)
• binaryUpdateBlock(long snapshot, long blockNumber, MemoryBuffer blockData)
• binaryUpdateCommit(long snapshot)

To explain, assume an application that has access to two trees, one called updated and the other stale, with obvious roles. The application will start by asking the updated tree for the first block of the next snapshot than the latest of the stale tree (getFirstBlockOfSnapshot). It will initiate an update operation on the stale tree (binaryUpdateBegin) and update it with the retrieved block (binaryUpdateBlock). It will then ask the updated tree for the next block (getNextBlockOfSnapshot) until it receives and End-of-Data reply, at which point it will conclude updates to the stale tree (binaryUpdateCommit). It will continue this update loop for any snapshots existing at the updated tree and not yet present on the stale one, until the last one is processed. At this point, the stale tree will be semantically equal to the updated one.

6 Evaluation

In this section, we evaluate IntegrityCatalog’s performance. We focus on the performance characteristics of RBB-Dict, as the major IntegrityCatalog operations (insert and search) are dominated by its behavior. In all tests, we use a single computer with an Intel Core i5-3570 running at 3.40GHz, 16GB memory, and a 2 TB hard disk drive (HGST Deskstar 5K300, 32MB cache, 5400rpm). The hard disk is slow enough to identify any performance problems of our disk based data structures.

We first measure the speed of insertions to the tree. Since file or URL names are what we expect to be used as object identifiers in the catalog, we selected a relatively high key size of 160 bytes; we kept the value size to 25 bytes, representing a typical SHA-1 signature with some kind of hash function id prefix.

The RBB-Dict module has a structural property that potentially affects performance, that is the maximum number of keys for each page in the tree. In our benchmarks, we vary this attribute (Keys per Page: KPP) from 250 keys per page up to 1000, with increments of 250. Furthermore, we vary the number of elements that are added before a snapshot is taken (Keys per Snapshot: KPS), with values of one thousand, ten thousand and one hundred thousand; this represents potential usage scenarios for the catalog. We extract and plot the average insertion time for each snapshot, which shows how insertion time increases when the tree size grows. To achieve that, we measure the time it takes to insert KPS elements in the tree and divide it by KPS; then we take a snapshot of the tree. We repeat this process until the tree has 1 million data elements. This illustrates the effect of RBB-Dict’s persistence, the fact that data pages formed in one snapshot are never modified but copied over for structural tree changes.

We first show the effect of choosing keys per page in Figure 2. Figure 2(a) shows the results of a snapshot size of 1K (KPS) running with all four settings for KPP, 2(b) for 10K KPS and 2(c) for 1M KPS. In all cases, the smallest selection of 250 for KPP gives the best results, as smaller-sized pages are read in the tree operations needed for insertion. In any case, the best time is less than 2 ms, while the worst is less than 30 ms.

![Figure 3: Effect of snapshot size in RBB-Dict insertion speed](image-url)

For this optimal setting of 250 for KPP, we show the effect of snapshot size variance in Figure 3, where insertion times for all 3 snapshot sizes are plotted. The small snapshot size of 1K produces the worst times, as it causes a lot of pages to be copied over from previous epochs because of copy-on-write; performance is a lot more stable for the larger snapshot sizes of 10K and 100K. In any case, the worst time is less than 9 ms, while the best is at 1 ms. We measure from 170 up to 237 insertions per second, figures that should satisfy the needs of our use case. We also performed benchmarks on a solid state disk device (a Samsung 840 Pro Series 256GB, on the same remaining hardware), where the insertions per second range from 1164 to 1366, a six-fold speedup demonstrating that the bottleneck is the slow hard disk device.

This performance analysis becomes more apparent once we study the on-disk size of the tree for each experiment. We show its size in GBs in Table 1, where we see that the more frequent snapshot intervals produce far larger files. In our optimal 250 KPP setting, a snapshot every one thousand elements produces a disk size of 43
GBs (for 1 million elements), while the more infrequent snapshot size of one hundred thousand produces an on-disk image of less than 2 GBs for the same number of elements. This behavior clearly favors heavier (more insertions per snapshot) usage.

Our second experiment regards search speed; to measure it we selected a 250 KPP tree layout and took snapshots every 1000 insertions, which is the worst measured case. At every snapshot we stop and search for a random sample of ten thousand already inserted elements, regardless of the snapshot they participated in. We also include in the measurement the verification of the search result, that is the re-calculation of the root digest from the proof that is returned. We average this measured time and illustrate it at every snapshot point in Figure 4. We see that search speed is constantly measured below 0.12 ms, with most results below 0.06 ms, across all tree sizes. In total, this represents approximately 20,000 searches per second, a figure certainly fast enough for IntegrityCatalog not to become a bottleneck in a repository object verification process.

Regarding our network protocol, the network traffic generated for each operation, per node, is presented in Table 2, where we are assuming a SHA-1 hash function (with a digest size 20 bytes). The frequent seal and verify operations are very efficient as only digests travel across nodes, while the heavier preserve and recover are dominated by the size of the snapshot and tree size they carry, respectively. For the heavier ones, every network message carries a single tree page. Analyzing a bit further the 43 GB size (the worst for our optimal 250 KPP case), we see it represents an average 43 MB per snapshot, the data expected to travel on the wire for a preservation synchronization operation of an up-to-date preserver node. Even at the very modest network speed of 10 Mbit/sec, it will take an average of 43 seconds for a (fairly common) snapshot transfer, and a total of 12 hours for a complete catalog restore in the (rare) case of a needed recovery.

7 Related work

The most obvious way to safeguard the integrity of an object in an archive is to maintain multiple (at least 3) copies of it; to verify its integrity, one can simply compare all copies and decide based on the majority’s point of view. However, the cost becomes quickly pro-

---

**Table 1: Disk size in GB for each setting of KPP (rows) and KPS (columns)**

| KPP  | 1K kps | 10K kps | 100K kps |
|------|--------|---------|----------|
| 250 kpp | 43     | 15      | 1.86     |
| 500 kpp | 67     | 16      | 1.83     |
| 750 kpp | 86     | 16      | 1.77     |
| 1000 kpp | 100    | 16      | 1.79     |

**Table 2: Network traffic generated by the catalog**

| operation | local | remote |
|-----------|-------|--------|
| seal      | <100 bytes | <100 bytes |
| verify    | <100 bytes | <100 bytes |
| preserve  | snapshot size | small |
| recover   | small | tree size |

Figure 4: Search time measured at every snapshot point
hibitve as the archive grows in size. Another alternative is to use Redundant Array of Inexpensive Disks (RAID) [29, 5, 2, 28] where either full copies are maintained or parity bits are automatically calculated and stored along with data, allowing recovery as long as a minimum number of disks are intact. Again, a significant space overhead is imposed to achieve integrity. These solutions however, besides detecting damage, also help repair it.

Another approach is to calculate a checksum for an object and store it along with it. To achieve this, one can use error detecting techniques [16], such as cyclic redundancy checks [30], the most famous being CRC32 [19]. Although they may be attractive for messages on communication channels and very fast to compute, they are clearly inadequate for long term storage, as they don’t provide strong pre-image resistance (an attacker can fairly easily calculate a second message with the same CRC as an existing one).

Another alternative for a “summary” function are algebraic formulas that produce signatures [17, 32, 18]. Schwarz et al. [33] proposed their use for proof of remote data possession when the owner no longer holds the original data. Although potentially faster to calculate than cryptographic hash functions, they again lack the pre-image resistance properties required to be applicable to the long-term digital preservation and archival systems we target.

Finally, cryptographic hash functions [27], such as SHA-* [11], may be used to produce strong digests, which can be stored along the objects themselves. This is today the strongest type of summary information that can be produced, and this is the scheme we build upon.

All aforementioned techniques however, where all integrity checking related artifacts are stored locally (either multiple copies of an object, or its checksum/digest), fail when a malicious entity attacks aiming to alter the existing content unnoticed. The attacker can simply alter all related elements and the changes will go unnoticed by any of the above techniques.

A different approach is demonstrated by the LOCKSS [24] peer-to-peer digital preservation system, which assumes multiple nodes store the same object and uses them for verification (and also for damage repair). This is a valid assumption for this system as its purpose is to preserve academic journals, that is, widely visible data. Verification is done by one node initiating a poll and multiple others, owners of the same object, participating in it. However, even this way, each node participating in the poll will need to read and process the complete object in order to produce a digest for it. This results in heavy load for the system as a whole; our approach can complement the existing polling protocol by allowing each node to verify its own content. This can reduce the polling rate per object and thus reduce overall system load.

Another method is to maintain copies of the object, or its digest, in write-once media; even this however becomes quickly unmanageable as the number of objects increases, as one needs to locate the proper media to access the required object. Additionally, as there is no guarantee what you read is exactly what you have written, you still need multiple copies. However, this approach would prohibit an adversary from altering an object unnoticed.

Haber et al. [14] propose a Content Integrity Service (CIS) based on hash trees along with hash-linking. The system defines epochs of operation. At the end of each epoch, the root digest, which is dependent upon the one of the previous epoch, becomes the ‘witness’ value and is expected to be safely published to a “widely available medium”. This witness can then be retrieved to verify proofs of membership. Such proofs, consisting of all needed tree nodes starting from the data element in question up to the root of the current epoch are expected to be stored by the archival system along with the object itself.

The CIS scheme was later enhanced by Song et al. [35] in the ACE system by introducing another epoch, above the aforementioned one, before arriving to the witness value. More specifically, the first epoch, which is dynamically sized by the system, forms again a hash tree on ingest. Once it is closed, the root value is incorporated in a hash-chain for the user-defined period (e.g. a week) of the second epoch. When this second epoch closes, the authenticator for this hash chain becomes the witness value to be published, or in any case preserved as securely as possible. The archive is again requested to store the membership proofs along the data objects and the integrity management system now stores (and verifies) the hash-chains. The two epoch approach assists in minimizing the proof size, but leaves part of the proof validation in the integrity management system, further complicating the solution compared to CIS.

Both schemes deal with change in the used hash function the same way; the object is expected to be re-registered along with the old proof, thus linking the new proof with the old one.

Both works try to establish and maintain the time of ingress for an object, doing so however for objects of the local node in isolation; our work does not yet focus on this, although it can be extended to provide more global timeline information, such as ordering the times of ingress of the same data object across different archival nodes.

The major drawback of both systems is that the repository is now asked to safely preserve the membership proofs for each data element while the integrity service only maintains hash trees of digests, organized by time.
and not indexed by object name. Thus, the simplest form of attack for an adversary is to erase the membership proof of an object, and then proceed to modify it, leaving the repository unable to verify the integrity of the object in question.

On this subject, the European Telecommunications Standards Institute (ETSI) defines a specification [10] based on Timestamping Authorities (TSA) and a Public Key Infrastructure (PKI), which addresses both object integrity and certification of an object’s existence before a certain point in time. This specification takes into account the phasing out of underlying algorithms by renewing integrity information with newer timestamps obtained with newer keys or from a newer TSA. This solution however depends on the reliability of the TSAs used, which now become single points of failure for integrity information while they also remain outside the control of the digital preservation system.

Another approach, proposed in RFC 4998 [12] uses mostly the same techniques as the ETSI standard above, but also provides for the use of hash trees to timestamp multiple objects at the same time. The interesting element of this solution is the anticipation of change of the hash tree’s hash function, which is performed by recreating the hash tree with the same data elements while concatenating the previous digest as input to the new hash function. If this process is performed before the old hash function is compromised, it safeguards against possible malicious tampering of the hash tree. While this solution, from our point of view, is not ideal because it relies on third parties (the TSA), the hash tree recreation is a technique that may be valuable as future work in our approach.

In our work, we organize all integrity metadata in a single data structure, addressable by object name. We make sure this data structure is tamper-evident and we also give a complete solution to its preservation and recovery, as well as its own integrity checking. This way, the host digital preservation system does not need to track any extra information alongside the objects. Additionally, when the host system asks for the integrity information of an object (by name), it is assured this information is intact. Because of the underlying RBB-Tree data structure, our proofs also do not increase in size as the number of snapshots increase, because the proof is defined for a specific snapshot only. Finally, our approach does not directly use a PKI; however it does need secure message delivery to and from the nodes comprising the verifiers and preservers lists. It is up to the host system to choose the right method for message authentication and integrity checks, however it does not have to store any secrets along the objects for the long term.

Our system is complementary to digital preservation and distributed storage systems and may be used as a tool by each storage node in the system to proactively verify the integrity of its contents. As such, it is more suitable to systems where the complete file is stored in different nodes, such as LOCKSS [24], FreeNet [6], FastSite [1], and Publius [37]. Systems that break files in pieces, such as Venti [31], OceanStore [20], CFS [9], Pastiche [7], Samsara [8], GridSharing [36], PASIS [13], and Glacier [15] can still use IntegrityCatalog to proactively validate the pieces they own.

Our solution is orthogonal to external audits by the owner of the objects, such as [21] or by an external auditor [34].

We make no assumptions about the host system’s approach on object privacy, however IntegrityCatalog does not expose any object data outside the system, as it only exposes hashes.

8 Conclusion

In this paper, we presented IntegrityCatalog, a novel distributed system that can be integrated into digital repositories that use integrity tokens to periodically verify the correctness of their locally stored data objects. Unlike prior approaches that focus only on making integrity metadata tamper-evident, IntegrityCatalog takes a holistic approach. It treats integrity metadata as first-class objects and provides a distributed network protocol that ensures that integrity tokens are preserved, verified regularly, and recovered in case of damage or adversarial attack, without requiring storage of long-term secrets. Our approach uses a new data structure, called RBB-Dict which provides a persistent, authenticated dictionary of arbitrary length keys and associated values, that is also efficient to synchronize over the network. Performance measurements of our IntegrityCatalog prototype show that insertions of digests of objects into the catalog can be sustained at a rate of >1K insertions per second, while search throughput is measured at 20K queries per second.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUTEUX, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In Proc. 5th Symposium on Operating System Design and Implementation (5th OSDI’02) (Boston, Massachusetts, USA, Dec. 2002), D. E. Culler and P. Druschel, Eds., USENIX Association.

[2] BITTON, D., AND GRAY, J. Disk shadowing. In vldb (Aug. 1988), pp. 331–338.

[3] BULDAS, A., LAUD, P., AND LIPMAA, H. Accountable certificate management using undeniable attestations. IACR Cryptology ePrint Archive 2000 (2000), 27.
