WORM-SEAL: Trustworthy Data Retention and Verification for Regulatory Compliance

Tiancheng Li[1], Xiaonan Ma[2,⋆], and Ninghui Li[1]

[1] Department of Computer Science, Purdue University
{li83,ninghui}@cs.purdue.edu
[2] IBM Almaden Research Center
xiaonan.ma@gamil.com

Abstract. As the number and scope of government regulations and rules mandating trustworthy retention of data keep growing, businesses today are facing a higher degree of regulation and accountability than ever. Existing compliance storage solutions focus on providing WORM (Write-Once Read-Many) support and rely on software enforcement of the WORM property, due to performance and cost reasons. Such an approach, however, offers limited protection in the regulatory compliance setting where the threat of insider attacks is high and the data is indexed and dynamically updated (e.g., append-only access logs indexed by the creator). In this paper, we propose a solution that can greatly improve the trustworthiness of a compliance storage system, by reducing the scope of trust in the system to a tamper-resistant Trusted Computing Base (TCB). We show how trustworthy retention and verification of append-only data can be achieved through the TCB. Due to the resource constraints on the TCB, we develop a novel authentication data structure that we call Homomorphic Hash Tree (HHT). HHT drastically reduces the TCB workload. Our experimental results demonstrate the effectiveness of our approach.

1 Introduction

Today's data, such as business communications, financial statements, and medical images are increasingly being stored electronically. While digital data records are easy to store and convenient to retrieve, they are also vulnerable to malicious tampering without detection. In the wake of high-profile corporation scandals, the number and scope of government regulations mandating trustworthy information retention keep growing. Examples of such regulations include SEC rule 17a-4 [30], SOX (Sarbanes-Oxley Act) [37], and HIPAA (Health Insurance Portability and Accountability Act) [36]. As a result, businesses today are facing a higher degree of regulation and accountability than ever, and failure to comply could result in hefty fines and jail sentences.

The fundamental purpose of trustworthy record retention is to establish irrefutable proof and accurate details of past events. For example, the SEC regulation 17a-4 states that records must be stored in a non-erasable, non-rewritable format. To help organizations meet such regulatory requirements [33], the storage industry has introduced a

⋆ Currently with Schooner Information Technology, Inc.

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 472–488, 2009.

number of compliance storage solutions focusing on WORM (Write-Once Read-Many) support. While physical WORM media (such as CD-R/DVD-R and magneto-optical disks) was used in some earlier compliance systems, due to performance, capacity and cost reasons they have been replaced by recent compliance offerings [12,27,18] which are based on standard rewritable storage media. In these systems the WORM property is enforced by software. All these systems allow users to specify some retention attributes (such as expiration date) for each data object, and prevent users from modifying or removing an unexpired data object.

Existing software-based WORM approaches, however, offer only limited protection against malicious attackers who compromise the system. This weakness is particularly serious in the regulatory compliance environment where the threat of intentional insider attacks is very real, as evidenced by previous industry scandals. For example, the attacker could be a system administrator who is asked by a high-level company executive to secretly modify or hide incriminating information, when there is a threat of an audit or a legal investigation. Here, not only does the attacker have the administrative access and privileges to the data systems, he may also have enough resources to launch sophisticated attacks. These software-based WORM approaches do not provide adequate protection because: (1) they are based on the assumption that the attacker could not break into the compliance storage system; (2) an attacker could potentially bypass the WORM protection mechanisms if he manages to access the storage devices directly; (3) existing solutions is insufficient to ensure trustworthy information retrieval [17]; and (4) support for data migration is critical for long-term data retention and is needed by system updates, disaster recovery, and so on.

In this paper, we present WORM-SEAL, a secure and efficient mechanism for trustworthy retention and verification for append-only indexed data in regulatory compliant storage servers. Wle reduce the scope of trust to a TCB (Trusted Computing Base). In other words, we divide the system into a trusted base (e.g., the TCB), and a semi-trusted part which can be trusted to a lesser degree (e.g., the main system where most of the storage and management functionalities are provided). We first present an approach based on Merkle hash tree. Due to the resource constraints on the TCB, we design a novel authentication data structure (called Homomorphic Hash Tree (HHT)) which can dramatically reduce the TCB overhead. Our approach also allows a single TCB with limited resources to safeguard a great amount of data efficiently. As a result, a single TCB can be shared among many systems, or can be used to provide trust-preserving services over a wide area network.

The rest of the paper is organized as follows. We discuss our models, assumptions and design goals in Section 2. We describe the overall architecture of WORM-SEAL in Section 3 and present the Homomorphic Hash Tree (HHT) data structure and TCB-friendly solution in Section 4. Experimental results are given in Section 5. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Background

We first examine a typical usage scenario of our system. Suppose an auditor wants to locate all the emails containing a particular keyword in a compliance system, he issues

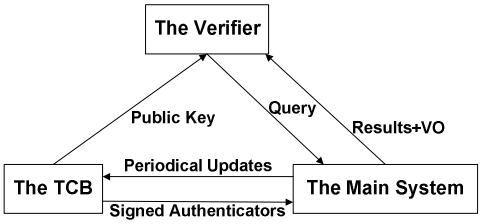


Fig. 1. The System Model for Regulatory Compliance

the query (potentially through an untrusted system administrator and an insecure communication channel) and receives five emails. With WORM-SEAL, the auditor would receive additional verification information along with the emails, which allows him to verify: (1) whether all fives emails are indeed coming from the system queried and have not been tampered with; (2) whether there are any other emails containing the same keyword which should have been included in the query result.

2.1 System Model

Figure 1 depicts the system model, which includes three distinct entities: (1) the main system, (2) the trusted computing base (TCB), and (3) the verifier. The main system hosts all the data, and provides other functionalities typically expected from a compliance storage server (such as storage management, query support, etc.). The TCB is responsible for running some trust preservation logic and maintaining a small amount of authentication information. Due to security concerns and resource constraints, it is desirable to keep the trust preservation logic as simple as possible. The integrity of data records and the correctness of query results can be verified through the verifier, which sits outside the administrative domain of the compliance server and relies on the authentication information maintained by the TCB to perform the verification.

Now we examine how the system handles updates and verification operations. When a new update request (e.g., creating a new data object) arrives, it is received by the main system, which deposits the data object and updates the related data pages accordingly. In addition, the main system also generates some authentication information which describes the data and metadata changes, and commits it to the TCB. Upon receiving such information, the TCB updates the secure authentication information it maintains.

When a query request arrives, it is handled by the main system. To allow verification of trustworthiness of the query result, the main system includes additional correctness proof, called *Verification Object (VO)*. The VOs are generated in such a way that it reflects the state (at the time of the query execution) of the corresponding secure authentication information maintained inside the TCB. The verifier can then verify whether the returned query result matches the associated VO. If so, the result can be trusted.

2.2 Threat Model and Assumptions

We assume that the TCB is secure (for example, the IBM secure co-processors meet the very stringent FIPS 140-2 level 4 requirements). We also assume that the TCB contains a trusted clock (or has a secure mechanism to synchronize its clock with a

trusted source), and provides some basic cryptographic primitives such as secure hashing, encryption and digital signatures. In our system, the TCB is configured with a private/public key pair. The private key of the TCB is kept secret while the public key is published and made widely accessible (for example, it could be available from the system's manufacture). In particular, the public key is available to the verifier.

We assume that the TCB has limited physical resources, such as internal storage (typically on the order of megabytes or less), CPU speed and communication bandwidth (which can be orders of magnitude slower than those in the main system). For example, it would not be possible to store all of the data (or a secure one-way hash for each data record) on the secure internal storage inside the TCB. On the contrary, the main system may consist of many powerful machines, vast amount of storage, and high-speed interconnections.

2.3 Design Goals

The design goal of our system is to preserve the trustworthiness of data stored on the untrusted main system through the trusted TCB, while minimizing the workload on the TCB by shifting as much work from the TCB to the main system as possible in a secure fashion. Our security goal is stated as follows: Assume that the TCB is not compromised and the main system has not been compromised by time t, any attempt to tamper data committed before time t will be detected upon verification.

For trust preservation, we must ensure that the correctness of the query results returned by the main system can be verified. Here, by correctness, we refer to the integrity, completeness, and freshness of the query result. Integrity means every record in the query result should come from the main system in its original form, completeness means every valid record in the main system that meets the query criteria should be included in the query result, and freshness means that the query result should reflect the current state of the main system when the query was executed (or at least within an acceptable time window).

3 Overall Architecture

We present the WORM-SEAL architecture and a Merkle hash tree based approach.

3.1 Preliminaries

Collision-resistant hash function. A cryptographic hash function takes a long string (or "message") of arbitrary length as input and produces a fixed length string as output, sometimes termed a message digest or a digital fingerprint. We say that a cryptographic hash function h is collision-resistant if it is computationally difficult to find two different messages m_1 and m_2 such that $h(m_1) = h(m_2)$. Widely-used cryptographic hash functions include SHA1 and SHA256.

Digital signature. A digital signature scheme uses public-key cryptography to simulate the security properties of a signature in digital form. Given a secure digital signature scheme, it is considered computational infeasible to forge the signature of a message

without knowing the private key. A digital signature algorithm is built from, e.g., the RSA scheme or the DSA scheme.

Merkle hash tree. The Merkle hash tree [24] is a binary tree, where each leaf of the tree contains the hash of a data value, and each internal node of the tree contains the hash of its two children. The root of the Merkle hash tree is authenticated either through a trusted party or a digital signature. To verify the authenticity of a data value, the prover has to send the verifier the data value itself together with values stored in the siblings of nodes on the path from the data value to the root of the Merkle hash tree. The verifier can iteratively compute the hash values of nodes on the path from the data value to the root. The verifier can then check if the computed root value matches the authenticated root value. The security of the Merkle hash tree is based on the collision resistance of the hash function; an attacker who can successfully authenticate a bogus data value must have a hash collision in at least one node on the path from the data value to the root. In this Merkle hash tree model, the authenticity of a data value can be proven at the cost of transmitting and computing $\log_2 n$ hash values, where n is the number of leaves in the Merkle hash tree.

Append-only data pages. We consider data that is organized as a collection of append-only data pages. Each data page contains data records that have the same attribute value. When a new data record enters the system, it is appended to the corresponding data page. We can build such a data structure for each attribute of the data. One simple example of append-only data is an audit log which documents how data records are accessed (such as creation, read, deletion, etc.) in a compliance system. For the purpose of discussion, let's assume that the audit log is organized by file IDs (or file names) and can be divided into many append-only data pages, one for each file ID (Other attributes may include file owner, creation time, and etc). A typical query in this case would be to retrieve all the log entries corresponding to a specified file ID.

3.2 Basic Merkle Tree (MT) Scheme

One approach is to use an aggregated authenticator, such as a Merkle hash tree. Specifically, the main system maintains a Merkle hash tree of the data pages in the following way. The i-th leaf of the Merkle hash tree stores an authenticator $A(P_i)$ for the i-th data page P_i. Each internal node of the Merkle hash tree contains the hash of its two children and the TCB stores the root of the Merkle hash tree.

Suppose that there is a new data record d_i appended into data page P_i. To update the authentication information maintained in the TCB (i.e., the root of the Merkle hash tree), the main system transmits the following data to the TCB: (1) a secure hash of the new data record $h(d_i)$, (2) the current $A(P_i)$, and (3) all nodes that are siblings of the nodes on the path from the leaf $A(P_i)$ to the root. Upon receiving the data from the main system, the TCB first verifies the authenticity of $A(P_i)$ by recomputing the root of the Merkle hash tree and comparing it with the root stored with the TCB. If the two roots do not match, the TCB is alerted that the received authenticator may have been compromised and will reject the update request. Otherwise, the TCB is assured that the received $A(P_i)$ is authentic as well as up-to-date, and continues the update process as follows. The TCB first updates $A(P_i)$ as $A(P_i) = H(A(P_i), h(d_i))$ (H is also a secure hash function) which now covers the new data record d_i. The TCB can then compute

the new root of the Merkle hash tree based on the new $A(P_i)$ and other Merkle hash tree nodes submitted by the main system. Finally, the TCB replaces the old root value with the new one in its internal storage.

On querying data page P_i, the main system returns the following data to the verifier: all data records in P_i, all the nodes that are siblings of the nodes on the path from leaf $A(P_i)$ to the root, an up-to-date root value which is signed with the TCB's private key. The verifier can then recompute the root of the Merkle hash tree from P_i and the Merkle hash tree nodes, and compare it with the signed one issued by the TCB. The verifier is assured of the trustworthiness of data page P_i if and only if the two values match.

The advantage of this approach is that the TCB only needs a constant size of storage for each attribute of the append-only data structure (i.e., the storage requirement for the TCB is $O(1)$). However, to update a single data page, the amount of information transmitted between the main system and the TCB, and the number of hash operations performed by the TCB are of the complexity $O(m \cdot \log N)$ where m is the number of data pages which have been updated and N is the total number of data pages. Given that the insertion of a new data record could trigger a number of data page updates, a scalable compliance server capable of handling high data ingestion rate can easily overwhelm the resource-limited TCB.

To solve this problem, we propose a novel solution which can reduce the storage, communication and computation overhead of the TCB all to a complexity of $O(1)$ simultaneously, regardless of the number of updated data pages in an interval. In addition, this is achieved without unduly increasing the burden on the main system or the verifier. We present the details of our solution in the next section.

4 The TCB-Friendly Approach

The key idea behind our solution is to develop an authentication data structure which has the advantage of a traditional Merkle tree but also has the following property: when a leaf node in the tree is updated, the TCB can update the root of the tree directly in a secure fashion based on the update to the leaf node, without information about other internal nodes in the tree. Furthermore, if multiple leaf nodes are updated in the tree, the TCB can securely update its state information based on an aggregated authenticator covering all the changes. In particular, the aggregated authenticator can be computed by the main system.

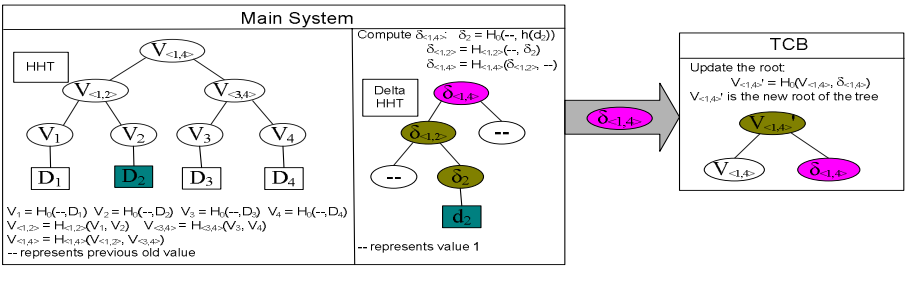


Fig. 2. Our Homomorphic Hash Tree (HHT) Scheme

With the above property, the TCB only needs to receive an aggregated authenticator from the main system in each interval, no matter how many data pages have been updated in the main system. The TCB can then perform a single operation to update its state information based on the received aggregated authenticator. This means that the communication/computation costs for the TCB in an interval are reduced to a constant.

In the following, we introduce an authentication data structure called Homomorphic Hash Tree (HHT) that satisfies the property described above. We then analyze its cost and present the security requirement. After that, we describe a construction of the HHT scheme and show how the HHT scheme is secure and achieves our design goals.

4.1 Homomorphic Hash Tree (HHT)

Our solution uses an authentication data structure that we call Homomorphic Hash Tree (HHT) shown in Figure 2. To make the discussion easier to follow, we assign a label to each node as follows: the leaf nodes are labeled numbers from 1 to N from left to right, each internal node is labeled a pair of numbers indicating the left-most descendent leaf and the right-most descendant leaf. For example, in Figure 2, the parent of the two leaf nodes labeled 1 and 2 has a label $\langle 1, 2\rangle$ and the root has a label $\langle 1, 4\rangle$.

The HHT tree is similar to a Merkle hash tree, but has several important differences. First, it uses a family of hash functions $\mathcal{H}$. While all leaf nodes use one hash function $\mathcal{H}_0$, each internal node uses a different hash function (the internal node labeled ℓ uses $\mathcal{H}_\ell$). Second, the hash functions used in the HHT satisfy the following homomorphic property. For any two hash functions $\mathcal{H}_{\ell_1}, \mathcal{H}_{\ell_2}$ in the family:

$$\mathcal{H}_{\ell_1}\left(\mathcal{H}_{\ell_2}\left(x_0, y_0\right), \mathcal{H}_{\ell_2}\left(x_1, y_1\right)\right) = \mathcal{H}_{\ell_2}\left(\mathcal{H}_{\ell_1}\left(x_0, x_1\right), \mathcal{H}_{\ell_1}\left(y_0, y_1\right)\right)$$

Third, there is an identity element 1 such that $\mathcal{H}_0\left(x, 1\right) = x$.

Our construction also uses an additional hash function h that computes the digest of new data records. This function is different from $\mathcal{H}_\ell$'s and does not need to satisfy any homomorphic property. For example, h can be the standard hash function SHA-1.

Leaf nodes. There is one leaf node for each data page P_i. This node stores the authenticator V_i for P_i $(i = 1, 2, ..., N)$. We use D_i^t to denote the contents of page P_i at the end of the t-th interval, and d_i^t to denote the new contents added to page P_i during the t-th interval. That is, $D_i^t = D_i^{t-1}||d_i^t$, where $||$ denotes concatenation. When no new content is added, $d_i^t = null$. We use δ_i^t to denote the message digest of d_i^t, defined as

$$\delta_i^t = \begin{cases} h(d_i^t) & \text{if } d_i^t \neq null \\ 1 & \text{if } d_i^t = null \end{cases}$$

The value of the authenticator for P_i at the end of the t-th internal is denoted by V_i^t, which is computed from V_i^{t-1} and δ_i^t as follows: $V_i^t = \mathcal{H}_0\left(V_i^{t-1},\ \delta_i^t\right)$. The value V_i^0 is defined as $V_i^0 = h\left(D_i^0\right)$ where D_i^0 is the initial content of P_i.

If there are no new data records for page P_i in the t-th interval, then $\delta_i^t = 1$ and therefore, $V_i^t = \mathcal{H}_0\left(V_i^{t-1}, 1\right) = V_i^{t-1}$. This means a leaf node V_i in HHT remains unchanged if there is no update to the corresponding data page P_i during an interval.

Internal nodes. Each internal node of the HHT is computed as the hash of its two children nodes.

Let V_ℓ^t denote the value of a node labeled ℓ at the end of the t-th interval. The value of each internal node ℓ is the resulted hash of its two children nodes ℓ_1, ℓ_2 as follows: $V_\ell^t = \mathcal{H}_\ell\left(V_{\ell_1}^t,\ V_{\ell_2}^t\right)$.

Update. Assume that we have the HHT for time $t-1$, where the value of a node ℓ is V_ℓ^{t-1}. Thus the root of the tree has value $V_{\langle 1,N\rangle}^{t-1}$. At time t, some leaf nodes need to be updated, and we show how to update the HHT to compute the new root. Specifically, we show that the new root $V_{\langle 1,N\rangle}^t$ can be computed from the old root $V_{\langle 1,N\rangle}^{t-1}$ and an aggregate hash $\delta_{\langle 1,N\rangle}^t$ computed by the main system.

First, for all leaf nodes ($1 \le i \le N$), $V_i^t = \mathcal{H}_0\left(V_i^{t-1},\ \delta_i^t\right)$.

Second, we calculate the parent nodes. Consider the parent of leaf nodes 1 and 2, we have

$$\begin{aligned} V_{\langle 1,2\rangle}^t &= \mathcal{H}_{\langle 1,2\rangle}\left(V_1^t,\ V_2^t\right) \\ &= \mathcal{H}_{\langle 1,2\rangle}\left(\mathcal{H}_0\left(V_1^{t-1},\ \delta_1^t\right), \mathcal{H}_0\left(V_2^{t-1},\ \delta_2^t\right)\right) \\ &= \mathcal{H}_0\left(\mathcal{H}_{\langle 1,2\rangle}\left(V_1^{t-1}, V_2^{t-1}\right),\ \mathcal{H}_{\langle 1,2\rangle}\left(\delta_1^t, \delta_2^t\right)\right) \\ &= \mathcal{H}_0\left(V_{\langle 1,2\rangle}^{t-1},\ \mathcal{H}_{\langle 1,2\rangle}\left(\delta_1^t, \delta_2^t\right)\right) \end{aligned}$$

We use $\delta_{\langle 1,2\rangle}^t$ to denote $\mathcal{H}_{\langle 1,2\rangle}\left(\delta_1^t, \delta_2^t\right)$, and more generally, $\delta_\ell^t = \mathcal{H}_\ell\left(\delta_{\ell_1}^t, \delta_{\ell_2}^t\right)$, where ℓ_1 and ℓ_2 are the two children of ℓ. Therefore, we have:

$$V_{\langle 1,2\rangle}^t = \mathcal{H}_0\left(V_{\langle 1,2\rangle}^{t-1},\ \delta_{\langle 1,2\rangle}^t\right)$$

Then consider the parent of the nodes $\langle 1,2\rangle$ and $\langle 3,4\rangle$, we have

$$\begin{aligned} V_{\langle 1,4\rangle}^t &= \mathcal{H}_{\langle 1,4\rangle}\left(V_{\langle 1,2\rangle}^t,\ V_{\langle 3,4\rangle}^t\right) \\ &= \mathcal{H}_{\langle 1,4\rangle}\left(\mathcal{H}_0\left(V_{\langle 1,2\rangle}^{t-1},\ \delta_{\langle 1,2\rangle}^t\right), \mathcal{H}_0\left(V_{\langle 3,4\rangle}^{t-1},\ \delta_{\langle 3,4\rangle}^t\right)\right) \\ &= \mathcal{H}_0\left(\mathcal{H}_{\langle 1,4\rangle}\left(V_{\langle 1,2\rangle}^{t-1}, V_{\langle 3,4\rangle}^{t-1}\right),\ \mathcal{H}_{\langle 1,4\rangle}\left(\delta_{\langle 1,2\rangle}^t, \delta_{\langle 3,4\rangle}^t\right)\right) \\ &= \mathcal{H}_0\left(V_{\langle 1,4\rangle}^{t-1},\ \delta_{\langle 1,4\rangle}^t\right) \end{aligned}$$

We can iteratively compute the root of the HHT in this manner and the new root of the HHT is computed as

$$V_{\langle 1,N\rangle}^t = \mathcal{H}_0\left(V_{\langle 1,N\rangle}^{t-1},\ \delta_{\langle 1,N\rangle}^t\right).$$

The value $\delta_{\langle 1,N\rangle}^t$ is the root of another HHT (called the delta HHT) whose leaf nodes are hashes of the new data records (i.e., δ_1^t,δ_2^t,...). The delta HHT has the same height as the HHT, and the same hash function is used by an internal node in the delta HHT as the one used by its counterpart in the HHT.

In our approach, the work of computing the root of the delta HHT $\delta_{\langle 1,N\rangle}^t$ is left to the main system. At the end of each interval, the main system computes $\delta_{\langle 1,N\rangle}^t$ and sends it to the TCB. Since only hashes of new data records during an interval show up in the delta HHT as non-empty leaf nodes, the storage and computation complexity of the delta HHT is proportional to the number of updated pages in one interval and the height of the HHT.

All that the TCB needs to do is to compute the new root through one single hash operation: the new root is computed as $V_{\langle 1,N\rangle}^t = \mathcal{H}_0\left(V_{\langle 1,N\rangle}^{t-1},\ \delta_{\langle 1,N\rangle}^t\right)$. The TCB then

Table 1. Complexity comparison of the MT scheme and the HHT scheme

	Storage (TCB)	Communication (MS,TCB)	Computation (TCB)	Communication (MS, Verifier)	Computation (Verifier)
MT scheme	$O(1)$	$O(m \cdot \log N)$	$O(m \cdot \log N)$	$O(\log N)$	$O(\log N)$
HHT scheme	$O(1)$	$O(1)$	$O(1)$	$O(\log N)$	$O(\log N)$

removes the old root $V^{t-1}_{\langle 1,N\rangle}$, stores the new root $V^{t}_{\langle 1,N\rangle}$, and sends a signed version of the new root with timestamp to the main system.

Verification. The construction of a VO is similar to that in the basic Merkle tree based scheme (MT). To prove the correctness of the data page P_i, the main system returns all the data records belonging to P_i, together with the siblings of all nodes on the path from V_i to the root, and the root of the tree which is timestamped and signed by the TCB.

On receiving the data, the verifier recomputes the root from P_i and the sibling nodes. The verifier then compares the computed root with the one signed by by the TCB. The content of P_i is proved correct if and only if these two values match.

Cost analysis. Table 1 shows the complexity of our HHT scheme as compared with that of the MT scheme, assuming that updates can be batched and the number of updates to unique pages in a batch is m, the total number of pages in the data structure is N. The verification time and *VO* size refer to the computation and communication overhead for verifying the correctness of a single data page, respectively.

4.2 Construction

Cryptographic functions. Our solution uses the following cryptographic functions:

- h : a collision resistant one-way hash function with arbitrary length input: h: $\{0,1\}^* \rightarrow Z_n$. One example of h is the SHA-1 hash function where the 160-bit output is interpreted as integers.
- $\mathcal{H}$: a hashing family $\{\mathcal{H}_\ell\}$ such that $\mathcal{H}_\ell(x,y) = x^{e_{\ell_1}} y^{e_{\ell_2}} \ mod\ n$ where n is the RSA modulus and e_{ℓ_1} and e_{ℓ_2} are the exponents. The hashing family $\mathcal{H}$ has the required homomorphic property:$\mathcal{H}_a\left(\mathcal{H}_b\left(x_0,y_0\right),\mathcal{H}_b\left(x_1,y_1\right)\right) = \mathcal{H}_b\left(\mathcal{H}_a\left(x_0,x_1\right),\mathcal{H}_a\left(y_0,y_1\right)\right).$

$\mathcal{H}_0 \in \mathcal{H}$ and $\mathcal{H}_\ell \in \mathcal{H}$. To construct $\mathcal{H}_0$ and $\mathcal{H}_\ell$, we need to instantiate the exponents e_{ℓ_1} and e_{ℓ_2} in the above definition, which is described below.

Instantiation of $\mathcal{H}_0$ and $\mathcal{H}_\ell$ hash functions. Our solution uses a set of distinct prime numbers $\{p_0, p_1, ..., p_N\}$ where p_0 is used in the instantiation of the function $\mathcal{H}_0$ and $p_1, p_2, ..., p_N$ are used in the instantiation of the functions $\mathcal{H}_\ell$. They can be chosen consecutively, in ascending order starting from, e.g., 65537.

The leaf hash function $\mathcal{H}_0$ is defined as $\mathcal{H}_0(x,y) = x \cdot y^{p_0} \ mod\ n$. We can see that $\mathcal{H}_0 \in \mathcal{H}$ and $\mathcal{H}_0(x,1) = x$. The internal hash functions $\mathcal{H}_\ell$ is defined as $\mathcal{H}_\ell(x,y) = x^{e_{\ell_1}} y^{e_{\ell_2}} \ mod\ n$ where ℓ_1 and ℓ_2 are the two children nodes of node ℓ. The following definition instantiates the exponents e_{ℓ_1} and e_{ℓ_2}.

Definition 1 (Tag Value and Exponent Value). The *tag* value of the i-th leaf is defined to be $T(i) = p_i$ for $i = 1, 2, ..., N$. The tag value of an internal node ℓ is defined as the product of the tag values of its two children, i.e., $T(\ell) = T(\ell_1)T(\ell_2)$ where ℓ_1 and

ℓ_2 are the two children nodes of ℓ. The *exponent* value e_ℓ of a node ℓ is defined the tag value of its sibling, i.e., $e_\ell = T(\bar{\ell})$ where $\bar{\ell}$ is the sibling node of ℓ.

It is easy to see that if $\ell = \langle i, j \rangle$, i.e., the leaf nodes that are descendants of ℓ are labeled from i to j, then $T(\ell) = p_i p_{i+1} \cdots p_j$. Furthermore, if a leaf k is a descendent of ℓ, then p_k doesn't divide the exponent of ℓ, since ℓ's sibling covers a different set of leaf nodes. For example, in Figure 2, the tag values of V_1 and V_2 are p_1 and p_2 respectively, and the tag value of $V_{\langle 1,2 \rangle}$ is $p_1 p_2$. The exponent values of V_1 and V_2 are p_2 and p_1 respectively, and the exponent value of $V_{\langle 1,2 \rangle}$ is $p_3 p_4$.

The verification process. The main procedure of verification is the reconstruction of the root of the HHT tree. We show how the verifier reconstructs the root of the HHT tree as follows. Consider the example in Figure 2, to verify $x = V_2$, the VO is $\{y_1 = V_1, y_2 = V_{\langle 3,4 \rangle}\}$. The root can be reconstructed as $V_{\langle 1,4 \rangle} = \mathcal{H}_{\langle 1,4 \rangle}\left(V_{\langle 1,2 \rangle}, V_{\langle 3,4 \rangle}\right) = x^{e_2 e_{\langle 1,2 \rangle}} y_1^{e_1 e_{\langle 1,2 \rangle}} y_2^{e_{\langle 3,4 \rangle}}$. Observe here, the exponent for each of x, y_1, y_2 is the product of the exponents of the nodes on the path from the corresponding node (V_2 for x, V_1 for y_1, and $V_{\langle 3,4 \rangle}$ for y_2) to the root. More generally, we define the *verification exponents* for each node in the HHT tree as follows.

Definition 2 (Verification Exponent). The *verification exponent* of a node ℓ is defined as the product of the exponents of the nodes on the path from ℓ to the root.

Note that if a leaf k is a descendent of ℓ, then p_k does not divide the verification exponent of a node ℓ. This is because this node is a descendent of every nodes on the path from ℓ to the root, and hence doesn't divide the exponent of any node on the path.

Let m be the height of the HHT tree, i.e., $m = \log N$. Let x be the value V_i. Let the verification exponent of x be F. After querying the data page content of V_i, the verifier receives a VO $\{y_1, y_2 ..., y_m\}$ from the main system. Let the verification exponents of y_i $(i = 1, 2, ..., m)$ be F_i. Then, the root of the HHT is reconstructed as

$$root = x^F \prod_{1 \le i \le m} y_i^{F_i} \tag{1}$$

The verification exponents $\{F, F_1, F_2, ..., F_m\}$ have the following property, which will be used in the security analysis in Section 4.3.

Lemma 1. Let the verification exponent of V_i be F. Let $\{y_1, y_2 ..., y_m\}$ be the VO for V_i, and $\{F_1, F_2, ..., F_m\}$ be their verification exponents. Then, we have $\gcd(F_1, F_2, ..., F_m) = p_i$ and $\gcd(p_i, F) = 1$.

Proof. One factor of F_j is the exponent of the node y_j, which is the tag value of its sibling node. As the sibling node is on the path from V_i to the root, p_i divides the tag value of this sibling node. It follows that p_i divides F_j. For any other p_k $(k \neq i)$, the leaf node whose tag value is p_k is a descendent of a node in $\{y_1, y_2, \cdots, y_m\}$, since these nodes cover all leaf nodes except for V_i. Suppose, without loss of generality, that the leaf node for p_k is covered by y_j, then p_k does not divide F_j. Therefore, $\gcd(F_1, F_2, ..., F_m) = p_i$. Finally, we note that $F = (\prod_{1 \le j \le N} p_j)/p_i$ and therefore, $\gcd(p_i, F) = 1$. The lemma holds.

4.3 Security Analysis

The security of our construction is based on the RSA assumption: For an odd prime e and a randomly generated strong RSA modulus n (that is, $n = pq$, where $p = 2p' + 1$, $q = 2q' + 1$, and p', q' are primes), given a random $z \in Z_n^*$, it is computationally infeasible to find $y \in Z_n^*$ such that $y^e = z$. This assumption holds for any odd prime e because we use the strong RSA modulus, $\phi(n) = 4p'q'$ and we have $\gcd(e, \phi(n)) = 1$; otherwise we have factored n.

Our security proofs also use the following well-known and useful lemma, which has been used in [31,14,10].

Lemma 2. Given $x, y \in Z_n^*$, along with $a, b \in Z$, such that $x^a = y^b$ and $\gcd(a, b) = 1$, one can efficiently compute $u \in Z_n^*$ such that $u^a = y$.

To show that this lemma is true, we use the extended Euclidean algorithm to compute integers c and d such that $bd = 1 + ac$. Let $u = x^d y^{-c}$ would work:

$$u^a = x^{ad} y^{-ac} = (x^a)^d y^{-ac} = (y^b)^d y^{-ac} = y$$

We now proceed to prove the security of our scheme through the following theorem.

Theorem 1. *An attacker who breaks into the main system at time t cannot succeed in corrupting data committed before time t without being detected upon verification.*

Proof. Suppose that an attacker compromises the main system during the t-th interval. Without loss of generality, suppose that the attacker wants to change the update history of the i-th data page committed at the w-th interval, where $w < t$. The attacker tries to show that the update is d', where the actual update is d. Let $V = V_i^{w-1}$ be the value of the i-th page in HHT at time $w - 1$, and $\delta = h(d)$ be the hash of the correct update.

Assuming collision resistance of h, then the attacker must come up with a path authenticating $\delta' = h(d') \neq \delta$ as the digest of the update in this interval. Let $x = \mathcal{H}_0(V, \delta) = V\delta^{p_0}$ and $x' = \mathcal{H}_0(V, \delta') = V\delta'^{p_0}$.

The attacker succeeds if she can create a VO $\{y'_1, y'_2, ..., y'_m\}$ that authenticates x' to the TCB. Let the verification exponent of the i-th data page be F. Let the verification exponents of the siblings of nodes on the path from the leaf to the root be $F_1, F_2, ..., F_m$, respectively. Let the correct VO that authenticates x to the TCB be $\{y_1, y_2, ..., y_m\}$. Then, based on Equation 1, the attacker succeeds if she can find a VO $\{y'_1, y'_2, ..., y'_m\}$, such that

$$(V\delta^{p_0})^F \prod_{1 \leq i \leq m} y_i^{F_i} = (V\delta'^{p_0})^F \prod_{1 \leq i \leq m} y_i'^{F_i}$$

That is,

$$\left(\frac{\delta}{\delta'}\right)^{p_0 F} = \prod_{1 \leq j \leq m} \left(\frac{y'_j}{y_j}\right)^{F_j}$$

To break the security of our HHT, an adversary $\mathcal{A}$ must be able to find such $\{y'_1, y'_2, ..., y'_m\}$ for an arbitrary δ'. Note that because $\delta' = h(d')$ is the result of a cryptographic hash function, the adversary cannot control δ'; when the adversary chooses a bogus

update d', he has to authenticate a random $h(d')$. We show that it is computationally infeasible to do so by reducing this problem to the RSA problem. Given such an adversary $\mathcal{A}$, we construct an adversary that breaks the RSA problem for the modulus we use in the HHT, which is a randomly generated strong RSA modulus, as follows:

When given a random $y \in Z_n^*$, we ask $\mathcal{A}$ to come up with a VO for $\delta' = \delta/y$. If $\mathcal{A}$ succeeds, then we have

$$y^{p_0 F} = \prod_{1 \le j \le m} \left(\frac{y'_j}{y_j}\right)^{F_j}$$

As shown in Lemma 1, $\gcd(F_1, F_2, \ldots, F_m) = p_i$ and $\gcd(p_i, p_0 F) = \gcd(p_i, F) = 1$. Now let $z = \prod_{1 \le j \le m} \left(y'_j/y_j\right)^{F_j/p_i}$, then we have $z^{p_i} = y^{p_0 F}$. By Lemma 2, one can efficiently compute y^{1/p_i}, which means that we constructed an adversary that has solved the RSA problem. Therefore, our construction is secure.

4.4 Support for Regulatory Compliance

We briefly show that our solution meets our design goals for regulatory compliance. The main goal of compliant data management is to support the WORM property: once committed, data cannot be undetectably altered or deleted. As shown in the security analysis above, our solution provides secure data retention and verification. Moreover, our HHT scheme is designed for dynamic append-only data and allows efficient search over data. Our solution also provides end-to-end protection and supports data migration. Once the data has been committed to the TCB, subsequent alteration or deletion of the data will be detected upon verification. Therefore, data migration does not give the attacker additional channels for tampering the data as long as the TCB is uncompromised. Finally, as shown by the cost analysis, our HHT scheme requires a very small amount of resources on the TCB (constant storage, constant communication cost, and constant computation cost for each interval). The scheme is scalable for the TCB even there are billions or trillions of data records in the storage systems.

5 Performance Evaluation

In this section, we describe our implementation of the WORM-SEAL system and present an evaluation of the performance by comparing it with the basic MT (Merkle Tree) scheme. We implemented both the HHT scheme and the basic MT scheme in C using the OPENSSL library (version 0.9.8e). To simplify the experiments and to provide a fair comparison, we use the same hardware platform (a 3.2GHz Intel Xeon PC) to measure the performance of the main system, the TCB and the verifier. While the actual numbers in a real system will be different, we focus on the relative workload ratio here. The parameters used in our experiments are listed in Table 2.

5.1 TCB Overhead

We measure the overhead of the TCB in updating the authenticator when there are 2^M updates to unique data pages where $M = 0, 1, 2, ..., 20$ (1 to 1 million page updates)

Table 2. System Parameters and Properties

Name	Description	Value
n	RSA modulus	1024 (bits)
sData	Size of an data record	up to 512 (bits)
nInt	# of time intervals	10^6
nPagePI	# of page updates per interval	10^3
nDataPI	# of data records updated in a data page	10^2
nData	# of data records in total	10^9
nPages	# of data pages	10^6
nPQ	# of pages queried	10^3
H	Hash function	SHA-1

in a time interval and when there are 2^N pages where $N = 0, 1, 2, ..., 30$ (1 to 1 billion data pages). The overhead is measured by: (1) the number of bytes required to be transmitted from the main system to the TCB, and (2) the time by the TCB to update the authenticator.

Experimental results are presented in Figure 3. Our HHT scheme performs consistently well in the experiments as its performance does not change much with respect to either M or N. The communication and computation overhead for the TCB remain constant (128 bytes and around $0.12 * 10^{-3}$ seconds) in our approach. For the basic MT scheme, the communication overhead grows quickly (almost linearly) with respect to nPagePI. The computation time is in the order of $2 * 2^M * N$. As M or N grows, the computation time also grows quickly. The performance differences between our HHT scheme and the basic MT scheme become much more signification when the tree size is large or the number of updated data pages is large.

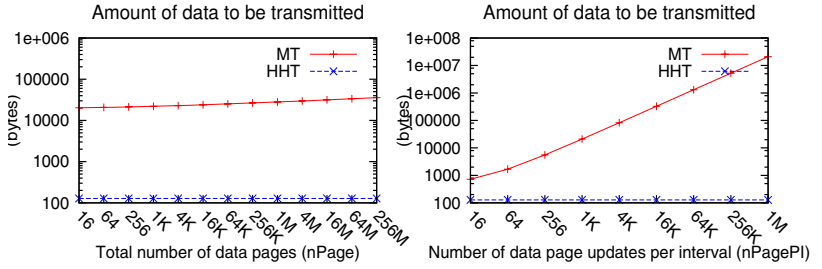
5.2 Main System Overhead

Similarly, we measure the overhead of the main system by measuring the computation time of the main system for each interval. The computation time of the main system includes: (1) the time to construct the authentication data, and (2) the time to update its authentication data structure. The total time is measured in the experiments.

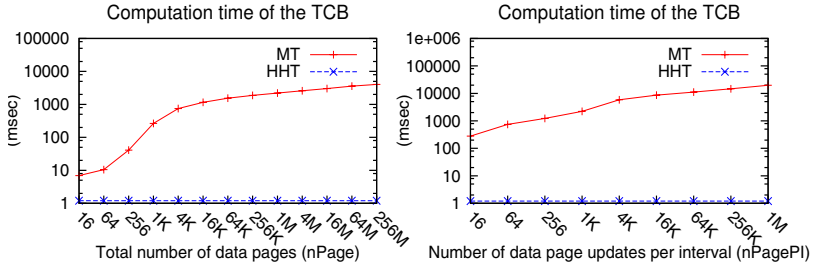
Based on the results in Figure 4, the basic MT scheme shows better performance than our approach on the side of the main system. This is not surprising as in our scheme, we shifted most of the workload on the TCB to the main system. In addition, the homomorphic hash functions used in our scheme can be more expensive than standard hash functions used in the basic MT scheme, such as SHA-1. For example, in our test system the standard hash function (SHA-1) takes around $2 * 10^{-6}$ to $3 * 10^{-6}$ seconds to compute while our homomorphic hash function takes about 10^{-3} seconds. The good news is that a real system with a large amount of data would be mostly dominated by disk IO latencies for accessing data pages and MT/HHT nodes.

5.3 Verification Cost

The verification cost is measured in terms of: (1) the time needed by the main system to construct the VO; (2) the size of the VO, i.e., the amount of additional data needs to be

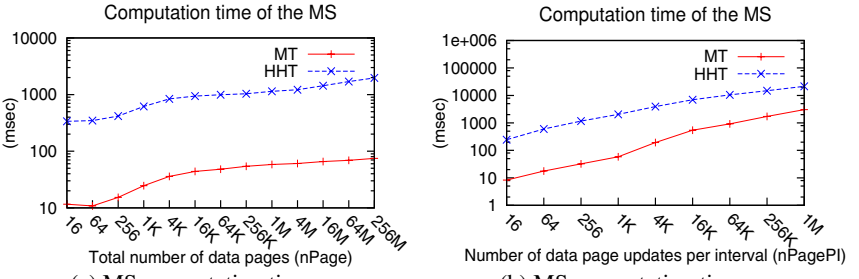


(a) Amount of transmitted data (b) Amount of transmitted data



(c) TCB update time (d) TCB update time

Fig. 3. Performance: the performance of the TCB



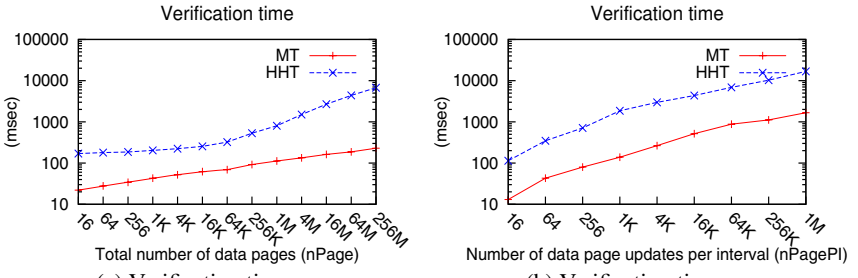
(a) MS computation time (b) MS computation time

Fig. 4. Performance: the performance of the main system

transmitted from the main system to the verifier; and (3) the verification time, i.e., the time needed by the verifier to verify the correctness of the received data.

To measure the verification time, we allow the verifier to issue to the main system random queries of the following form: returning data pages i_1, i_2, ..., i_{dim}, where dim indicates the number of data pages that are requested to be verified. For each selected parameter dim, we generate 1000 random queries Q for the experiments.

Results in Figure 5 show that verification time increases as $nPage$ or dim increases. In both experiments, the basic MT scheme shows better performances than our HHT scheme (for similar reasons mentioned in the main system overhead discussion).



(a) Verification time (b) Verification time

Fig. 5. Performance: the verification cost

However, as we have argued in Section 3, the verification process is much less frequent than the update process and thus the verification cost is a less-critical issue than the overhead on TCB.

6 Related Work

The scheme of using a small TCB to protect a scalable amount of untrusted storage has been studied [23,32]. Our model is different from the TDB model [23] in that in our model the TCB does not hae access to the actual data. The solutions are also different; we design the homomorphic hash tree (HHT) which protects append-only data structures while they use encryption and Merkle tree for protecting the sensitive state information. In Sion [32], the TCB needs to generate signatures for every VR (a collection of "similar" records) and generate another signature for expired records.

Several related problems have been studied but they are different from the problem we study in this paper. In both query verification for third-party publishing [11] and secure file services on untrusted platforms [22,20], the data owner can construct VO from the original data whereas in our model, TCB has to rely on the main system to provide update requests. Our approach does not consider secure deletion [26] and data provenance [16]. We consider append-only data that cannot be handled by POTSHARDS [35]. Many have proposed solutions for auditing logs integrity protection, including symmetric-key schemes [5,29], public-key schemes [4,19], and time-stamping [15,34]. None of these approaches have our homomorphic property.

Finally, we review related work in cryptography. Homomorphic hash functions can be constructed from the Pederson commitment scheme [28] or from Chaum et al. [8]. Homomorphic hash functions have been used in a number of areas, e.g., peer-to-peer content distribution [21,13]. The homomorphic property used in those schemes is simpler and does not work in our approach. Incremental hashing [2,3,9] allows the new hash $h(M')$ to be computed from the old hash value $h(M)$ and the updates to the message, instead of hashing the new message M'. Cryptographic accumulators [6,1,7] have been designed to allow proof of membership without a central trusted party. However, neither incremental hashing nor cryptographic accumulators consider the problem in the hash tree context. Merkle hash tree was used in [25] but for the purpose of constructing membership proof while not revealing information about the set.

7 Conclusion

In this paper, we have proposed a framework for trustworthy retention and verification of append-only data structures in a regulatory compliance environment. Our solution reduces the scope of trust in al compliance system to a tamper-resistant TCB. In particular, we present a TCB-efficient authenticated data structure which can greatly reduce the TCB overhead in handling updates to append-only data. Experimental results show the effectiveness of our approach, compared with a basic Merkle tree based scheme. Our solution can be integrated with existing regulatory compliance storage offerings to offer truly trustworthy end-to-end data verification.

References

1. Baric, N., Pfitzmann, B.: Collision-free accumulators and fail-stop signature schemes without trees. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 480–494. Springer, Heidelberg (1997)
2. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography: The case of hashing and signing. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 216–233. Springer, Heidelberg (1994)
3. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography and application to virus protection. In: STOC, pp. 45–56 (1995)
4. Bellare, M., Miner, S.K.: A forward-secure digital signature scheme. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (1999)
5. Bellare, M., Yee, B.: Forward integrity for secure audit logs. Technical report, University of California at San Diego, Department of Computer Science and Engineering (1997)
6. Benaloh, J.C., de Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 274–285. Springer, Heidelberg (1994)
7. Camenisch, J.L., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 61–76. Springer, Heidelberg (2002)
8. Chaum, D., van Heijst, E., Pfitzmann, B.: Cryptographically strong undeniable signatures, unconditionally secure for the signer. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 470–484. Springer, Heidelberg (1992)
9. Clarke, D., Devadas, S., van Dijk, M., Gassend, B., Suh, G.E.: Incremental multiset hash functions and their application to memory integrity checking. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 188–207. Springer, Heidelberg (2003)
10. Cramer, R., Shoup, V.: Signature schemes based on the strong rsa assumption. In: CCS, pp. 161–185 (1999)
11. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.G.: Authentic third-party data publication. In: DBSec, pp. 101–112 (2000)
12. EMC Corp. EMC Centera, `http://www.emc.com/products/family/emc-centera-family.htm`
13. Gkantsidis, C., Rodriguez, P.: Cooperative security for network coding file distribution. In: INFOCOM, pp. 1–13 (2006)
14. Guillou, L.C., Quisquater, J.-J.: A practical zero-knowledge protocol fitted to security microprocessor minimizing both transmission and memory. In: Günther, C.G. (ed.) EUROCRYPT 1988. LNCS, vol. 330, pp. 123–128. Springer, Heidelberg (1988)

15. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 437–455. Springer, Heidelberg (1991)
16. Hasan, R., Sion, R., Winslett, M.: The case of the fake picasso: Preventing history forgery with secure provenance. In: FAST, pp. 1–14 (2009)
17. Hsu, W.W., Ong, S.: Worm storage is not enough. IBM Systems Journal special issue on Compliance Management (2007)
18. IBM Corp. IBM TotalStorage DR550, http://www.ibm.com/servers/storage/disk/dr
19. Itkis, G., Reyzin, L.: Forward-secure signatures with optimal signing and verifying. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 332–354. Springer, Heidelberg (2001)
20. Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K.: Plutus: Scalable secure file sharing on untrusted storage. In: FAST, pp. 29–42 (2003)
21. Krohn, M.N., Freedman, M.J., Mazières, D.: On-the-fly verification of rateless erasure codes for efficient content distribution. In: S&P, pp. 226–240 (2004)
22. Li, J., Krohn, M., Mazières, D., Shasha, D.: Secure untrusted data repository (sundr). In: OSDI, pp. 121–136 (2004)
23. Maheshwari, U., Vingralek, R., Shapiro, W.: How to build a trusted database system on untrusted storage. In: OSDI, p. 10 (2000)
24. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988)
25. Micali, S., Rabin, M.O., Kilian, J.: Zero-knowledge sets. In: FOCS, pp. 80–91 (2003)
26. Mitra, S., Winslett, M.: Secure deletion from inverted indexes on compliance storage. In: ACM Workshop on Storage Security and Survivability (StorageSS), pp. 67–72 (2006)
27. Network Appliance, Inc. SnapLock TM Compliance and SnapLock Enterprise Software, http://www.netapp.com/products/ler/snaplock.html
28. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (1992)
29. Peterson, Z.N.J., Burns, R., Ateniese, G., Bono, S.: Design and implementation of verifiable audit trails for a versioning file system. In: FAST, pp. 93–106 (2007)
30. Securities and Exchange Commission. Guidance to Broker-Dealers on the Use of Electronic Storage Media under the National Commerce Act of 2000 with Respect to Rule 17a-4(f) (2001), http://www.sec.gov/rules/interp/34-44238.htm
31. Shamir, A.: On the generation of cryptographically strong pseudorandom sequences. TOCS 1(1), 38–44 (1983)
32. Sion, R.: Strong worm. In: ICDCS, pp. 69–76 (2008)
33. Sion, R., Winslett, M.: Regulatory-compliant data management. In: VLDB, pp. 1433–1434 (2007)
34. Snodgrass, R.T., Yao, S.S., Collberg, C.S.: Tamper detection in audit logs. In: VLDB, pp. 504–515 (2004)
35. Storer, M.W., Greenan, K.M., Miller, E.L., Voruganti, K.: Potshards: Secure long-term storage without encryption. In: USENIX Annual Technical Conference, pp. 142–156 (2007)
36. United State Department of Health. The Health Insurance Portability and Accountability Act (1996), http://www.cms.gov/hipaa
37. United States Congress. Sarbanes-Oxley Act of (2002), http://thomas.loc.gov