Assuring a Cloud Storage Service

Somya D. Mohanty, Mahalingam Ramkumar
Department of Computer Science and Engineering
Mississippi State University, MS.

***Abstract*—A cloud storage assurance architecture (CSAA) for providing integrity, privacy and availability assurances regarding any cloud storage service is presented. CSAA is motivated by the fact that the complexity of components (software / hardware and personnel) that compose such a service, and lack of transparency regarding policies followed by the service makes conventional security mechanisms insufficient to provide convincing assurances to users. As it is impractical to rule out hidden undesired functionality in every component of the service, CSAA bootstraps all desired assurances from simple transformation procedures executed inside a low complexity trustworthy module; no component of the cloud storage service is trusted.**

I. Introduction

Users of a cloud storage service create files for access from multiple platforms owned by the same user, and/or for access by other users explicitly specified by the originator of the file. From a security perspective, users of the service desire assurances regarding the *integrity*, *privacy*, and *availability* of files stored at remote/unknown locations.

A practical cloud storage service [1] will include numerous elements in the form of complex hardware, software, and personnel. Unexpected (malicious or accidental) functionality in *any* hardware/software component, or malicious intent / incompetence of personnel who administer the service, can result in the violation of the desired assurances. A rogue systems administrator could delete files or expose the contents of a file to unauthorized parties; so can an attacker who may have surreptitiously gained access to a server, perhaps by exploiting a bug or a Trojan horse in some software/hardware component.

Even if the owner/deployer of the service is convinced that the system is very well designed, and all necessary steps have been taken to ensure the integrity of the service, this is a far cry from actually providing convincing proof of the integrity of the service to it's users. While it is far from practical to convince the users of the lack of undesired functionality in *every* hardware/software component and personnel of such a service, it is reasonable for the service to be able to assure the operation of a *single* trustworthy module. This is especially true if the module is deliberately constrained to possess *simple and open* functionality. In this paper, we propose a cloud storage assurance architecture (CSAA) for a generic cloud storage service, where users are assured of the integrity of the service by a trustworthy module $\mathbf{T}$. The specific contributions include an algorithmic description of module $\mathbf{T}$ functionality, and protocols that leverage simple trusted functionality to provide integrity, confidentiality and availability assurances to users.

A. Overview of CSAA

The users of any cloud storage service expect contents of files to be made privy only to users explicitly authorized by the owner of the file. They expect that only explicitly authorized users can modify their files. They expect the service to provide only the latest version of a queried file (unless explicitly queried for an older version). Users also expect that they should not be improperly denied service.

Taking advantage of the pre-image resistance property of any cryptographic hash function $h()$, the ability to assure *integrity* of any file translates to the ability to store the cryptographic hashes of each version of every file in a "protected boundary" (that is guaranteed to not be molested). The ability to assure *freshness* of the file (that the latest version is provided, even if the querier may not keep track of the version number) can be translated into the ability to maintain a monotonic counter for each file inside the secure boundary. Ensuring that only authorized users can modify the file implies the ability to store access control lists (ACL) for each file within the protected boundary.

Taking advantage of encryption schemes, the ability to ensure *privacy* of contents of files can be translated in the ability to store all file encryption keys inside the protected environment, and the ability to check the ACL before releasing the secret to authorized users. That users should not be improperly denied service (by incorrectly stating that the "file does not exist" or that the user "does not have the requisite access") implies that a process running in the secure environment should be able to unambiguously answer a simple question: "what is the access privilege of a user u for a file f?"

As the service is not trusted, file hashes, encryption secrets, and ACLs, will need to be submitted to processes running inside the protected environment by users of the service. In order to prevent users from providing deliberately inconsistent information (for example, to taint the reputation of a good cloud storage service) the information provided by users should also be verifiable, and approved by the service. For example, the service should be able to verify that the authenticated file hash submitted to the module by the user is consistent with the file submitted to the service, or that the ACL submitted to the module by the user is the same as the ACL submitted to the service, etc. In CSAA, the "protected boundary" is a resource limited module $\mathbf{T}$.

1) Clark-Wilson Model: CSAA bears some similarities to the Clark-Wilson (CW) model [2], [3] for system integrity. In the CW model,

1) all data items whose integrity need to be guaranteed are

constrained data items (CDI).
2) *integrity verification procedures* (IVP) take all CDIs as input, and outputs a binary value indicating if the system is in a valid/invalid state.
3) CDIs may be modified only by well-formed *transformation procedures* (TP).

Specifically, well-formed TPs are guaranteed to take the system into a correct state *if* the system was at a correct state *before* the execution of the TP. That a system *is* in a correct state is initially demonstrated by an IVP. From this time onwards, if only a sequence of well-formed TPs are applied, the system is guaranteed to remain in a correct state.

In the CW model, the correctness of IVPs and TPs are certified during design time by a "security officer" for the system. The only restriction on TPs imposed by CW model is by definition of CW-triples of the form (user, TP, CDIs) that specify which user process is allowed to execute a TP, and which CDIs can be modified by the TP. The CW model simply assumes run-time integrity of TPs and IVPs. The assumption that TPs executed on general purpose computers can not be molested, is far from realistic. The major departure of CSAA from the CW model is that it imposes strict limitations on TPs and IVPs to ensure that they can be executed inside the confines of a resource limited boundary offered by a trustworthy module $\mathbf{T}$. Specifically, CSAA, constrains TPs/IVPs to be simple and fixed procedures composed of only logical and hash operations, and demand small (for example, a few KB) and constant memory size. The need to limit the complexity of $\mathbf{T}$ arises from the fact that high complexity can severely limit ones ability to consummately verify and certify the integrity of module $\mathbf{T}$ [4], [5].

2) Key Elements of CSAA: In CSAA, file indexes, file hashes, ACLs, and file encryption secrets are akin to CDIs in the CW model. Not withstanding self-imposed constraints on module $\mathbf{T}$ capabilities, CSAA does not place *any* constraints on the scale of the service — the number of files, users, size of ACLs or even number of versions of each file.

Key elements of CSAA include a versatile data-structure, an Index Ordered Merkle Tree (IOMT) [7] - [9], and the concept of self-certificates [10]. The IOMT permits a module to represent a practically unlimited number of dynamic "virtual CDIs" as a single CDI — the root of an IOMT stored inside the module $\mathbf{T}$. Virtual CDIs can be stored outside the module in an untrusted location. Simple functionality consisting of a small number of hash operations afford the ability check the integrity of any virtual CDI against the single CDI stored inside the module. Self-certificates, which are "memoranda issued to oneself" — enable the module to store any number of static CDIs in an untrusted location by taking advantage of a secret privy only to module $\mathbf{T}$.

B. Organization

The rest of this paper is organized as follows. In Section II we provide an overview of the desired features and assurances regarding the operation of a cloud storage service. In Section II-C we provide an overview of CSAA. A core component of CSAA, the index ordered merkle tree (IOMT), is the subject of Section III. Section III-C outlines module generic $\mathbf{T}$ functionality required to assure the integrity of an IOMT. Section IV outlines CSAA specific IVPs and TPs, and protocols for ensuring the integrity of processes like creating new files, updating files, updating ACLs, and reporting values associated with a file to users.

II. Features and Desired Assurances

The participants in CSAA include an untrusted cloud storage service $\mathcal{S}$, a trusted module $\mathbf{T}$, and any number of users.

Service $\mathcal{S}$: The service may include one or more database servers with access to large back-end storage. The specific components of $\mathcal{S}$ are irrelevant from a security perspective, as such components are assumed to be untrusted.

Trusted module $\mathbf{T}$: The module is assumed to be read-proof and write-proof [6]. In other words, a) secrets stored inside the module can not be exposed, and b) the simple TPs executed inside the module (which will be described in the rest of this paper) can not be modified.

Users: The system may possess an unlimited number of users. Every user is issued a unique identity. It is assumed that every user shares a secret with the module $\mathbf{T}$. We shall represent the secret shared between the module and a user u_i as K_i.

Users of the system employ their shared secret for securely submitting file hashes, secrets, ACLs, to the module, and for receiving authenticated file hashes, secrets and acknowledgments from the module. The service $\mathcal{S}$ serves as a middle-man between the users and the module to check correctness of all information provided by users to the module.

A. Files and ACLs

The total number of users/subscribers of the service $\mathcal{S}$ is assumed to be dynamic, and potentially unlimited. Any subscriber can create a file, choose a random secret to encrypt the file, and upload the encrypted file to $\mathcal{S}$. The original owner of the file is also expected to specify an ACL for the file, which is also uploaded to $\mathcal{S}$.

The ACL $\mathbf{A}$ for a file is assumed to be of the form

$$\mathbf{A} = \{(u_1, a_1), (u_2, a_2), \ldots, (u_l, a_l)\} \tag{1}$$

where $u_1 \cdots u_l$ represent user identities and $a_1 \cdots a_l$ their respective access permissions. We assume three types of permissions

1) $a_i = 1$ implies read-only;
2) $a_i = 2$ implies read-write access; such users may create new versions of the file;
3) $a_i = 3$ implies read-write for both the file and the ACL; such users can even modify the ACL for the file.

Collaborating users are *not* expected to possess any out-of-band mechanism for communicating file encryption secrets. Thus, file encryption secrets will also need to be securely stored by $\mathcal{S}$, and made available only to authorized user.

B. Integrity / Privacy / Availability Assurances

The CSAA approach aims to provide the following assurances regarding the operation of $\mathcal{S}$:

[A1.] $\mathcal{S}$ will not alter files; only users explicitly granted access level 2 or higher can modify the file, and in this process create a new version of the file.

[A2.] $\mathcal{S}$ is not trusted with file encryption secrets; only the module $\mathbf{T}$, and users explicitly specified in the ACL can gain clear access to file encryption secrets;

[A3.] $\mathcal{S}$ can not modify the ACL; only users with ACL level 3 for the file can do so.

[A4.] Only the latest version the file will be provided by $\mathcal{S}$ to authorized users, except when explicitly queried for an earlier version.

[A5.] After an ACL has been modified, the old ACL will not be used to determine the access privileges.

[A6.] When a user u with legitimate access rights requests a file, $\mathcal{S}$ will *not* refuse to provide access. Specifically, $\mathcal{S}$ is required to provide authenticated denial, in the form of convincing proof that "the request can not be entertained."

[A7.] In providing authenticated denial, no unsolicited information should be revealed to users.

Assurances A1 is towards *authenticity and integrity* of files; assurances A2 and A3 are required to guarantee *privacy* of file contents; assurances A4 and A5 address *replay* attacks; assurance A6 is towards *authenticated denial* — to prevent improper denial of service to a subscriber for a legitimate request.

Assurance A7 is required to address data mining issues that arise from authenticated denial. When authenticated denial is mandated, the RFSS is expected to provide a verifiable response to *every* query — either by providing the requested data, or by convincing the querier that the query *can not be entertained*. Typical strategies for authenticated denial demonstrate non existence of the requested object by ordering all objects that *do* exist, and demonstrating the presence of two adjacent objects that "cover" (or span) the requested object. Unfortunately, such an approach makes it possible for users to make random queries to gain knowledge of the existence of two other objects. For example, a user may send a query for a random file to learn about the existence of two files that *do* exist. Similarly, an user not in the ACL for a file, may get to know the identities of two users who *are* included in the ACL. The ability to gain unsolicited knowledge can motivate malicious users to make random queries for nefarious data mining purposes which can a) unduly burden the RFSS, and b) compromise the expectations of privacy of users. To prevent such attacks it should be ensured that *no unsolicited information will need to be revealed* by $\mathcal{S}$.

C. Overview of CSAA

The constrained data items necessary to guarantee all desired assurances include ACLs for each file and file-hash/file-encryption-secret for each version of every file. All CDIs can be seen as a database with one record for each file. Each record in the CDI-DB, indexed by a file index f, contains

1) a list of two-tuples that represent the ACL for the file; and
2) a list of two-tuples for each version indicating the file hash γ and file encryption secret σ.

A record $\mathbf{R}$ for index f is thus of the form

$$\begin{aligned}\mathbf{R}_a = \quad & [f\{(u_1, a_1), \ldots, (u_l, a_l)\}, \\ & \{(\gamma_1, \sigma_1)), \ldots, (\gamma_r, \sigma_r)\}]. \end{aligned} \tag{2}$$

The database is dynamic, as files may be added or deleted, the ACL for any file may be changed at any time, and new versions of a file may be added. All information necessary to construct this dynamic database comes from authenticated information asynchronously supplied by users to the module, using the service $\mathcal{S}$ as the middle-man. Every authenticated request from the user to modify the database is acknowledged by the module, indicating failure or success.

To create a new file, a user u_i and the service $\mathcal{S}$ agree on a file index f. A message from u_i, indicating the file index f is the trigger needed to create a record for f and set the ACL for f to $\{(u_i, 3)\}$, attributing user u_i with read-write privileges for the ACL.

To add a new version a user conveys values f, γ, σ' where γ is a file hash and σ' is the encrypted version of the file-encryption secret σ. If the user has access level 2 to f, a two-tuple (γ, σ) is added to the file record.

To update the ACL a user with access level 3 provides a new ACL (a list of two tuples), which replaces the old ACL. To delete the file a user with access level 3 can provide an empty ACL.

Any user may request to perform any action on any file. Even if the file does not exist, or if the user does not have the necessary access right. In all such scenarios too the user expects an authenticated response from the trusted module. Authenticated denial is necessary to prevent the untrusted service provider from improperly denying service in scenarios where the user *does* have the necessary privileges. Note that if the user has no access, or if the file does not exist, the user does not even need to know if file f does or does not exist. This is to provide assurance A7 (no unsolicited information should be revealed). The response from the trusted module in such a case is simply "illegal request." IF the use is in the ACL of the file but does not have the necessary access privilege the module's acknowledgment indicates the current access level of the user.

1) The Challenge: What makes the ability to reliably respond to queries (based on asynchronously provided authenticated information from users) challenging for the module is the fact that

1) the module is severely resource limited, and
2) we do not place any limitation on the number of files, or number of users, size of the ACL for a file, or the number of versions for a file.

Fortunately, the reason that a severely resource limited module *can* still meet this challenge is thanks to a versatile data structure — an Index Ordered Merkle tree (IOMT). The IOMT is a binary hash tree constructed using a standard cryptographic hash function $h()$. The IOMT enables the module $\mathbf{T}$ which

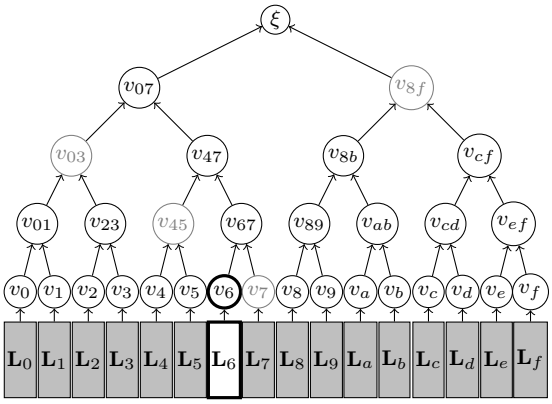

Fig. 1. A binary hash tree with 16 leaves.Nodes v_{67}, v_{47}, v_{07} and ξ are "ancestors" of v_6. v_{07} is the "immediate parent" of v_6. v_7 is the sibling of v_6, and v_{45}, v_{03} and v_{8f} are siblings of it's ancestors v_{67}, v_{47} and v_{07} respectively. The set of siblings $\{v_7, v_{45}, v_{03}, v_{8f}\}$ is the set of nodes "complementary" to v_6.

stores only a single cryptographic hash ξ inside the trusted boundary, to track the integrity of any number of databases of any size maintained by an untrusted entity. More specifically, by performing $\log_2 N$ hash operations (where N is the total number of database records) the module can a) verify the integrity of any record against the root ξ stored inside the module, or b) infer the absence of a record with a specific index, or c) update the root ξ corresponding to addition / updates to records. For our purposes, all CDIs (the database with ACL two-tuples and version three-tuples for each file index) may actually be stored by the untrusted service $\mathcal{S}$. Only a single IOMT root will be stored inside the module $\mathbf{T}$. Each event that calls for updating/adding a record or responding to a query by conveying values from a record will only call for execution of $\mathbb{O}(\log_2 N)$ hash operations inside the module.

III. Index Ordered Merkle Tree

Similar to the better known Merkle hash tree [11], an IOMT is a binary hash tree constructed using a standard cryptographic hash function like SHA-1. A tree with N *leaves* has $2N-1$ internal nodes (cryptographic hashes) distributed over $L = \log_2 N$ levels ($\frac{N}{2^l}$ nodes in levels $l = 0$ to $l = L$). The lone node at level L is the *root* of the tree, which is a cryptographic commitment to all leaves. Figure 1 depicts a binary tree with 16 leaves.

A. Nodes and Leaves

Corresponding to a leaf L_i is a leaf node v_i at level 0 of the tree, computed as

$$v_i = h(L_i). \tag{3}$$

Two adjacent nodes are each level (sibling nodes) are hashed together to compute the immediate parent at a higher level. More specifically, the parent node p of two sibling nodes u and v is computed as

$$p = H_N(u,v) \quad = \quad \begin{cases} u & \text{if } v = 0 \\ v & \text{if } u = 0 \\ h(u,v) & \text{if } u \neq 0, v \neq 0 \end{cases} \tag{4}$$

For example, $v_{67} = H_N(v_6, v_7)$, $v_{03} = H_N(v_{01}, v_{23})$, etc.

The root of the tree is a commitment to all leaves and nodes. Corresponding to any leaf node v_i it is trivial to identify a set of L *complementary* nodes $\mathbf{v}_i$ (Figure 1) depicts the set of $L = 4$ nodes complementary to v_6). The complementary nodes of v_6, together, can be seen as a commitment to all nodes *except* v_6. A simple one way function $f_{bt}()$ consisting of L $H_N()$ operations, confirms the existence of a one-way relationship between the leaf node v_i (and hence the leaf L_i) and the root ξ, through the set of complementary nodes $\mathbf{v}_i$.

$$\xi = f_{bt}(v_i, \mathbf{v}_i). \tag{5}$$

For example, v_6 and ξ are related through $\{v_7, v_{45}, v_{03}, v_{8f}\}$ through a sequence of $L = 4$ operations $v_{67} = H_N(v_6, v_7)$, $v_{47} = H_N(v_{45}, v_{67})$, $v_{07} = H_N(v_{03}, v_{47})$ and $\xi = H_N(v_{07}, v_{8f})$.

If there is a valid reason to update leaf node v_i to v_i', the new root can be computed as $\xi' = f_{bt}(v_i, \mathbf{v}_i)$. Using the same *complementary* nodes bfv_i for verifying the v_i against root ξ and updating ξ to ξ' ensures that no other leaf node is affected by the update (as $\mathbf{v}_i$ is a commitment to all nodes except v_i).

Note that $H_N()$ is a simple hash function if both inputs are non zero; else, it outputs a parent that is the same as the non zero child. The parent of two zero children is also zero (or $H_N(0,0) = 0$). Treating zero nodes in a special manner makes it possible to easily insert nodes. A tree can be seen as including *any* number of zero leaf nodes. Any zero node can be updated to a non zero value to insert a node.

1) IOMT Leaves: The leaves of an IOMT are constrained to form a *virtual circular linked list*. Specifically, an IOMT leaf is a three-tuple of the form $(\text{INDEX}, \text{NEXT\_INDEX}, \text{VALUE})$ where VALUE is a is a value associated with index INDEX.

The quantity zero is given a special interpretation in IOMTs. An index can never be zero. A like leaf $(a, a', 0)$ with zero VALUE field is a "place-holder" for index a. $(a, a', 0)$ does *not* mean that "index a is associated with value 0." A place-holder $(a, a', 0)$ indicates that "no value is associated with index a."

For all leaves in a tree $\text{NEXT\_INDEX} > \text{INDEX}$ except for the leaf with the highest index, a_{max}, for which the next index is the least index a_{min}. For a tree with a single leaf $\text{NEXT\_INDEX} = \text{INDEX}$. The constraint (to remain a linked list) is enforced when leaves are inserted are removed from the tree. A leaf is always inserted as a place-holder; only place-holders can be removed.

To insert a place-holder for an index b the verifier need to be convinced of the existence of a leaf or place-holder (b, b', ω_b) such that (b, b') encloses a. This is true if $b < a < b'$, or if $b' \leq b$ (which will be the case if b is the highest index and b' is the least index, or if $b = b'$ is the sole index) then any a that is lower than b' or greater than b is considered to be circularly enclosed by (b, b'). In other words (b, b') encloses a if

$$(b < a < b')\text{OR}(\text{b}' \leq \text{b} < \text{a})\text{OR}(\text{a} < \text{b}' \leq \text{b}) \tag{6}$$

To ensure integrity of the linked list when a place holder for a is inserted, the newly inserted place-holder will be $(a, b', 0)$, and the encloser (b, b', ω_b) will be modified to (b, a, ω_b). Recall that an IOMT can be seen as consisting of any number of

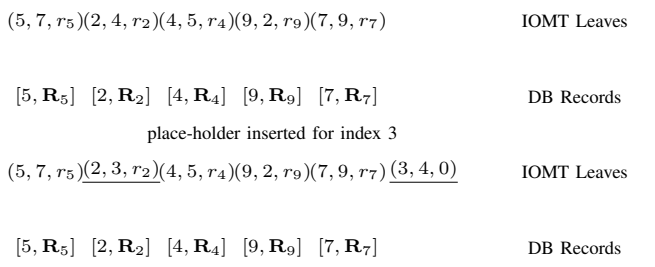

Fig. 2. Relationship between a database and IOMT leaves. $\mathbf{R}_i$ is a record for an index. r_i is a compact representation of $\mathbf{R}_i$ (for example, $r_i = h(\mathbf{R}_i)$). Insertion of a place-holder leaf has no effect on the data base even though it modifies an enclosing leaf (shown in a lighter shade).

nodes with value zero. To insert the place holder one such zero node will need to be updated to $h(a, b', 0)$; simultaneously, a node $v_b = h(b, b', \omega_b)$ will need to be updated to $v'_b = h(b, a, \omega_b)$. Correspondingly, the root of the IOMT will be updated (by executing $f_{bt}()$) to account for the modifications to two leaf nodes.

B. Database Representation Using IOMT

Together, all leaves of an IOMT can be seen as a representation of a database, where each leaf corresponds to a record for an index specified by the first field. The value ω_a associated with index a can be seen as a succinct commitment to the database record for index a (for example, hash of the record).

Figure 2 depicts the relationship between leaves of an IOMT and a database represented by the IOMT. Applications employing IOMTs can be seen as two-party protocols involving a (typically resource limited) *verifier*, and a (resource rich) *prover*. The prover maintains the database of records, all IOMT leaves, and nodes. Only the root ξ of the IOMT will need to be stored by the verifier. The prover can readily identify a set of complementary nodes necessary to prove the integrity of any leaf against the root ξ.

If the prover requires to modify a record, an application specific justification will need to be provided. For example, in the case of CSAA, a justification in the form of a authenticated request from an authorized user is necessary.

As can be seen from Figure 2, from the perspective of the database represented by an IOMT, addition of a place-holder makes no difference — even while it required two leaves (and the IOMT root) to be modified. If ξ was the root before the insertion of a place holder, and ξ' the root after insertion, then ξ and ξ' are considered as *equivalent* roots. In order to update a record for an index a (change the value field in the IOMT leaf for index a) typically a database/application specific justification will be necessary. However, a module can be requested to change it's root to an equivalent root without providing an application specific justification.

C. IOMT Algorithms Inside A Trusted Boundary

Consider a trusted boundary capable of executing the algorithm $f_{bt}()$. More specifically, given a hash x and a set of hashes (complementary nodes) $\mathbf{x}$, the module can readily compute $y = f_{bt}(x, \mathbf{x})$. If the inputs x and $\mathbf{x}$ were actually chosen from an IOMT, y is guaranteed to be a node in the same IOMT. More specifically, if $\mathbf{x}$ included n values, then y is an ancestor of x that is n levels higher. If the inputs were randomly chosen it is still true that y is an ancestor of x in *some* IOMT, though not necessarily an IOMT that is actually used by anyone.

Assume a secret χ inside the trusted boundary, that was spontaneously generated, and thus not known to any entity apart from $\mathbf{T}$. Having determined a relationship between x and y (that y is an ancestor of x) the module can issue a memorandum to itself as

$$\rho = \mathrm{HMAC}([\mathrm{x}, \mathrm{y}], \chi) \tag{7}$$

where HMAC() represents a function for computing a hashed message authentication code (HMAC), executed inside the module. The values ρ, x and y can be provided back to the module at any time to convince the module that it had already verified that y is an ancestor of x (without actually having to provide the complementary values $\mathbf{x}$).

Similar to records, IOMT leaves and nodes, certificates can be stored by the untrusted prover. While dynamic values will need to be represented as leaves of a hash tree before they can be stored outside, to prevent replay attacks. However, a static values can be stored using less expensive symmetric certificates.

Simple additional functionality inside the module can then be used to *combine* two or more certificates to make higher level inferences about the IOMT, or the database represented by the IOMT, and certify such inferences. Depending on the nature of the higher level inference, each certificate can be identified by a *type*-field.

In the rest of this section we shall outline some useful certificate types (NU, EQ, RV and RU) along with a description of functions that generate such certificates of different certificate types computed as

$$\begin{aligned} \rho_{nu} &= \mathrm{HMAC}(\mathrm{NU}, [\mathrm{x}, \mathrm{y}, \mathrm{x}', \mathrm{y}'], \chi); \\ \rho_{eq} &= \mathrm{HMAC}(\mathrm{EQ}, [\mathrm{y}, \mathrm{y}'], \chi); \\ \rho_{rv} &= \mathrm{HMAC}(\mathrm{RV}, [\mathrm{x}, \mathrm{v}, \mathrm{y}], \chi); \\ \rho_{ru} &= \mathrm{HMAC}(\mathrm{RU}, [\mathrm{x}, \mathrm{v}, \mathrm{y}, \mathrm{v}', \mathrm{y}'], \chi); \end{aligned} \tag{8}$$

Certificates of type NU and EQ are oblivious to actual records represented by an IOMT. Certificate types RV and RU are not. A certificate NU-$[x, y, x', y']$ is proof that y is an ancestor of x, and if $x \to x'$, then $y \to y'$. In the rest of this paper we shall use the notation type-$[\cdots]$, for example,

$$\mathrm{NU} - [\mathrm{x}, \mathrm{y}, \mathrm{x}', \mathrm{y}'], \mathrm{EQ} - [\mathrm{z}, \mathrm{z}'], \ldots \tag{9}$$

etc., to represent the type and contents of a certificate.

A certificate EQ-$[y, y']$ is proof that the root of an IOMT can be toggled between y and y' without affecting the integrity of any record in the database represented by the IOMT. A certificate RV-$[x, v, y]$ is proof that in an IOMT with root y there exists a leaf for INDEX x with VALUE v. If $v = 0$ the implication is that no record exists with INDEX x. This may be due to a) existence of a mere place-holder for INDEX x, or absence of a leaf/place-holder with INDEX x — indicated by the presence of an encloser for index x. A certificate RU-$[x, v, y, v', y']$ is proof that an IOMT with root y there exists

a leaf for INDEX x with VALUE v, and that if $v \rightarrow v'$, then $y \rightarrow y'$.

Function $F_{nu}()$ takes inputs x, x' and $\mathbf{x}$, computes $y = f_{bt}(x, \mathbf{x})$ and $y' = f_{bt}(x', \mathbf{x})$ to create a certificate of type NU.

$$F_{nu}() : \left.\begin{array}{l} x, x', \mathbf{x} \\ y' \leftarrow y \leftarrow f_{bt}(x, \mathbf{x}) \\ \text{IF } (x \neq x') \\ \quad y' \leftarrow f_{bt}(x', \mathbf{x}) \end{array}\right\} \rightarrow \mathrm{NU[x, y, x', y']}. \quad (10)$$

$F_{nu}()$ is the only function that accepts complementary nodes as inputs. All other functions generate various certificate type by manipulating various certificate types.

Function $F_{cat}()$ can be used to combine two NU certificates to create another NU certificate. Specifically, a certificate NU-$[x, y, x', y']$ and a certificate NU-$[y, z, y', z']$ can be used to create a new certificate NU-$[x, z, x', z']$ binding x to higher level ancestor z. In other words function $F_{cat}()$ can be represented as

$$F_{cat}() : \left.\begin{array}{l} \mathrm{NU[x, y, x', y']} \\ \mathrm{NU[y, z, y', z']} \end{array}\right\} \rightarrow \mathrm{NU[x, z, x', z']}. \quad (11)$$

Recall that inserting a place-holder for index a in any IOMT requires the existence of an encloser (b, b', ω_b), and that after insertion, the leaf node corresponding to the encloser will need to be modified from $v_e = h(b, b', \omega_b) \rightarrow v'_e = h(b, a, \omega_b)$ and another leaf node (corresponding to the newly inserted leaf) will need to be modified from $0 \rightarrow v_i = h(a, b', 0)$. Given a NU certificate which states that "$v_e \rightarrow v'_e$ implies $y \rightarrow y'$" and another NU certificate stating that "$0 \rightarrow v_i$ implies $y' \rightarrow y''$, function $F_{eq}()$ can readily conclude that y and y'' are equivalent roots, and issue a certificate $\rho = \mathrm{HMAC(EQ, [y, y''], \chi)}$.

More specifically, changing an IOMT root from $y \rightarrow y''$ corresponds to inserting a place-holder. Conversely, changing a root from $y'' \rightarrow y$ corresponds to deleting a place-holder! In the same spirit of Eq (11) function $F_{eq}()$ can be represented as

$$F_{eq}() : \left.\begin{array}{l} \mathrm{NU[v_e, y, v'_e, y']} \\ \mathrm{NU[0, y', v'_i, y'']} \\ (b, b') \text{ encloses } a \\ v_e = h(b, b', \omega_b) \\ v'_e = h(b, a, \omega_b) \\ v_i = 0; v'_i = h(a, b', 0) \end{array}\right\} \rightarrow \mathrm{EQ[y, y'']}. \quad (12)$$

Given a certificate NU-$[x, y, x, y]$ binding a value x to an ancestor y, and given a leaf (a, a', ω_a) such that the leaf hash is $x = h(a, a', \omega_a)$, function $F_{rv}()$ can conclude that (a, a', ω) is a leaf in a tree with root y and issue a certificate binding the record index a and value ω_a to the ancestor node y. Given an additional index b such that (a, a') encloses b, $F_{rv}()$ also infers that "no record for index b exists in the tree with root y." Accordingly, function $F_{rv}()$ issues a certificate of type RV (record verified) binding values $b, 0$ and y (the zero indicates that no record exists for index b).

$$F_{rv}() : \left.\begin{array}{l} \mathrm{NU[x, y, x, y]} \\ x = h(a, a', \omega_a) \\ \langle b \text{ encl. by } (a, a') \rangle \end{array}\right\} \rightarrow \begin{array}{l} \mathrm{RV[a, \omega_a, y]}. \\ \langle \mathrm{RV[b, 0, y]} \rangle \end{array} \quad (13)$$

Given a certificate NU-$[v, y, v', y']$, values (a, a', ω_a) satisfying $v = h(a, a'\omega_a)$, and ω' such that and $v' = h(a, a'\omega'_a)$, function $F_{ru}()$ can issue a certificate of type RU (record update) which states that "updating a record with index a from $\omega_a \rightarrow \omega'_a$ will require the root to be modified from $y \rightarrow y'$." In other words,

$$F_{ru}() : \left.\begin{array}{l} \mathrm{NU[v, y, v', y']} \\ v = h(a, a', \omega_a) \\ v = h(a, a', \omega'_a) \end{array}\right\} \rightarrow \mathrm{RU[a, \omega_a, y, \omega'_a, y']}. \quad (14)$$

D. Application Specific Functions

For systems/applications whose integrity is assured by a trustworthy module $\mathbf{T}$, the untrusted system is the prover that maintains one or more application specific databases and IOMTs corresponding to the databases. The module stores one or a small fixed number of IOMT roots. Apart from interfaces/functions exposed by the module to generate various types of application independent certificates, the module will expose some additional interfaces:

1) functions that accepts equivalence certificates to insert/delete place-holders;
2) transformation procedures (TPs) that accept authenticated proof of the need to modify a database record (in the case of the cloud storage service $\mathcal{S}$, an authenticated request from an authorized user is necessary to update any record in the CDI DB).
3) integrity verification procedures (IVPs) necessary to submit integrity assurances to users of the system, and possibly
4) other functions for generating additional application specific certificate types.

TPs that accept proofs for the need to change a leaf from (say) $(a, a', 0) \rightarrow (a, a', \omega_a)$ will accept a certificate like RU-$[a, 0, \xi, \omega_a, \xi']$ indicating that *if* the current root is ξ, the root should be changed to ξ', in order to modify the record for index a. IVPs that supply proof of integrity, similarly, may accept certificates of type RV to convey application specific record contents (for example, the correct hash of a file) to users. In order to interact with users, the module $\mathbf{T}$ will need to support additional functionality for establishing secrets with users of the system.

IV. CSAA Protocols

Underlying the CSAA protocols are three assumptions,

1) the integrity of simple functions executed inside the module $\mathbf{T}$,
2) privacy of secrets employed by the module to interact with the outside world, and
3) the hash function $h()$ is pre-image resistant.

The module $\mathbf{T}$ possesses in-built functionality $\mathcal{K}()$ to compute a secret it shares with any user. We shall represent by

$$K_i = \mathcal{K}(\mathbf{S}, u_i) \quad (15)$$

the secret K_i shared with user u_i. There are several ways to establish a secret between a module and users; one possibility is to employ a trusted key distribution center (KDC) who

issues a secret S to the module. The KDC also issues secrets to the users. A user u_i is issued a secret $K_i = h(S, u_i)$. In this case the functionality $K_i = \mathcal{K}(S, u_i)$ is simply $K_i = h(S, u_i)$. If it is desired to employ a plurality (say, 2) of independent KDCs, then each KDC supplies module with a a secret (say, S^1 and S^2 respectively). Each KDC issues a secret to each user (secrets of u_i are then $h(S^1, u_i)$ and $h(S^2, u_i)$, and the shared secret $K_i = \mathcal{K}(\{S^1, S^2\}, u_i) = h(S^1, u_i) \oplus h(S^2, u_i)$.

The module is also possesses in-built functionality to compute HMACs. Messages from users are authenticated by appending a HMAC. For example, to authenticate a bit-string $\mathbf{u}$ a user u_l (who shares a secret K_l with the module) computes a HMAC as

$$\mu = \mathrm{HMAC}(\mathbf{u}, \mathrm{K_l}) \tag{16}$$

Acknowledgments from the module $\mathbf{T}$ to the user u_l are also authenticated in the same manner, using a HMAC computed using secret K_l. As was discussed in the previous section, self-certificates are also computed using HMACs using the secret χ known only to the module.

The untrusted $\mathcal{S}$ maintains the CDI-DB, and the corresponding CDI-IOMT. Module $\mathbf{T}$ stores only an IOMT root ξ of the CDI-IOMT.

A. CDI-DB

The CDI-DB maintained by $\mathcal{S}$ can be seen as consisting as one record for every file index f. A record for file index f includes

1) the latest version number q_f of f;
2) a counter c_f (which is incremented every time any modification is made to record index f in the CDI-DB);
3) an ACL represented as an IOMT with root α_f; a leaf in the ACL tree is of the form (u, u', a), indicating that user u has access privilege a for file f (and that all user identities enclosed by (u, u') do not have any access privilege);
4) a certificate of type FR (file record) binding values f, c_f, α_f, q_f (certificate FR-$[f, c_f, \alpha_f, q_f]$); this certificate is re-issued by the module every time the counter c_f is incremented.

Leaves of the CDI-IOMT (with root ξ) maintained by $\mathcal{S}$ are of the form (f, f', c_f) where c_f is a counter. In addition, corresponding to every version of each file, the CDI-DB includes four values. The values corresponding to version q of file f (where $1 \leq q \leq q_f$) are

1) a value κ, which is a commitment to the file encryption secret σ; more specifically, $\kappa = h(\sigma, f)$;
2) a value λ which is a function of the hash of the encrypted file γ and the commitment κ. Specifically, $\lambda = h(\gamma, \kappa)$.
3) a value σ_s which is an encrypted version of the file encryption secret σ; specifically, $\sigma_s = h(\kappa, \chi) \oplus \sigma$ can be decrypted only by the module $\mathbf{T}$.
4) a certificate ρ_{vr} of type VR (version record) binding values f, q and λ (certificate VR-$[f, q, \lambda]$). One certificate is issued by the module every time a new version of the file is created.

B. Transformation Procedures

The copy of the root ξ of the CDI-IOMT stored inside the module $\mathbf{T}$ can only be modified by TPs executed inside the module.

A TP $F_{ph}()$ which accepts an equivalence certificate EQ-$[y, y']$ as input can be used to insert/delete place holders (for file indexes) in the IOMT. Specifically, if the current root is $\xi = y$, it is modified to y'; if the current root is $\xi = y'$ it is set to y.

$$F_{ph}(): \quad \mathrm{EQ}[\xi, \xi'] \text{ OR } \mathrm{EQ}[\xi', \xi] \rightarrow \{\xi \rightarrow \xi'\} \tag{17}$$

Apart from purposes of inserting/deleting place-holders (which does not affect the CDI-DB) all other modifications to ξ correspond to a due to a modification to the CDI-DB (and hence the CDI-IOMT). The justification for such modifications have to be provided as authenticated inputs from a user authorized to do so (access level $a \geq 2$ for adding a new version, and access level $a = 3$ for updating ACL or removing a file). An authenticated message from a user u_i to the module (authenticated using secret K_i)

$$\mu = \mathrm{HMAC}(\mathrm{type}, \mathrm{f}, \mathrm{c}, \mathrm{v}], \mathrm{K_i}) \tag{18}$$

includes a type (which distinguishes between updates to ACL and updates to the file), file index f, a counter c, and a value v; the value $v = \alpha$ is the rot of an ACL IOMT for messages that update the ACL. For file update messages $v = \lambda$ for the new file version. Every successful request is acknowledged by the TP. As we shall see later illegal requests (non existence of file, lack of access rights etc.) are also acknowledged using an IVP $F_{res}()$ (discussed in Section IV-C) that informs the user of the nature of the illegality of the request, while at the same time making sure that no unsolicited information is revealed.

From the perspective of the module, updating the ACL can also be for reserving a file index f or deleting a file f by conveying ACL root $\alpha = 0$.

A message $[f, 0, \alpha_0]$ from a user u_i along with a certificate RU-$[f, 0, \xi, 1, \xi']$ is sufficient to convince the module that no record exists in the CDI-DB for index f, and that adding a record with counter set to 1 will necessitate modification of root to ξ'. The module also issues a certificate FR-$[f, c_f = 1, \alpha_0, q = 0]$, which is required to make the next update to the record for file f.

For all other update requests from a user u an FR-certificate (indicating ACL root α_f) and an RV certificate RV-$[u, a, \alpha_f]$ indicating access privilege of user is necessary. The two certificates are sufficient to convince the module that *if* the current counter value for index f is c_f, *then* u_i has access level a for file f. If the user has required access level, then a certificate RU-$[f, c_f, \xi, c'_f, \xi']$ is sufficient to convince the module that file f with counter value f_c is indeed consistent with the indeed current root ξ, and that modifying the counter to c'_f will require the root to be updated to ξ'. The new counter value $c'_f = c_f + 1$ for all update operations, except for deletion of file, for which $c'_f = 0$.

All updates that increment the counter produce a new FR certificate for the updated counter value (which is needed for the next update). Updates for adding a new version also

$$
\left.\begin{array}{l}\{f, c = c_f, v\}_{u_i}\\ RU[f, c_f, \xi, c'_f, \xi']\\ FR[f, c_f, q_f, \alpha]\\ RV[u_i, a, \alpha]\\ \text{IF } (c'_f = c_f + 1);\end{array}\right\} \Rightarrow \begin{array}{l}\text{IF } (c_f = 0)\\ \quad FR[f, c'_f, 0, v = \alpha_0]\\ \quad \xi \leftarrow \xi';\\ \text{IF } (u_i = 3)\\ \quad FR[f, c'_f, q_f, v = \alpha]\\ \quad \xi \leftarrow \xi';\\ \text{IF } (u_i \geq 2)\\ \quad FR[f, c'_f, q_f + 1, v = \lambda]\\ \quad VR[f, q_f + 1, v]\\ \quad \xi \leftarrow \xi';\end{array}
$$

$$
\text{IF } (c'_f = 0) \wedge (\alpha = 0) \Rightarrow \text{IF } (u_i = 3)\ \xi \leftarrow \xi';
$$

Fig. 3. CASS TP $F_{tp}()$ that modifies CDIs. For reserving a file, updating a file/ACL the counter value is incremented ($c'_f = c_f + 1$). All these updates produce a FR certificate necessary for the next update. For deleting a file the counter value is $c'_f = 0$. No certificate is produced. The "type" field in the user request and the acknowledgment from the module are *not* shown.

produce an additional VR certificate VR-$[f, q, \lambda]$, which can be used to convey the value λ associated with any version q of file f to any user with access level 1 or higher. Figure 3 depicts the algorithm for transformation procedure $F_{tp}()$ which handles all updates. All successful updates are acknowledged (*not* shown in Figure 3)

Before invoking the the TP the service $\mathcal{S}$ is expected to ensure that the ACL submitted by the user is consistent with the value $v = \alpha$ in the message and that the counter value c in the message is consistent with the current counter value c_f for f. When reserving a file $\mathcal{S}$ may require that the user is added with level 3 access in the ACL.

Along with a message $[f, c_f, \lambda]$ from a user u_l to add the next version of file f, the user is expected to send to $\mathcal{S}$ an encrypted secret σ', a commitment κ, and the new version of the file. The service $\mathcal{S}$ hashes the (encrypted) file to ensure that the file hash is indeed γ, and verifies that $\lambda = h(\gamma, \kappa)$.

If the user chooses not to encrypt the file, then $\kappa = 0$, and $\sigma' = 0$. If $\kappa \neq 0$ $\mathcal{S}$ can not by itself verify the integrity of the secret σ' submitted by the user. This is achieved using a function $F_{rs}()$ exposed by the module. Given values u_l, f, c_f, κ, $F_{rs}()$ decrypts the secret, verifies that it is consistent with the commitment κ. Only then is the secret encrypted using χ and returned to $\mathcal{S}$ for storage as σ_s.

$$
F_{rs}: \left.\begin{array}{l} f, q, u_l, \sigma', \kappa\\ \sigma = h(f, c_f, K_l) \oplus \sigma'\\ \text{IF } (\kappa = h(f, \sigma))\\ \quad \sigma_s = h(\kappa, \chi) \oplus \sigma;\end{array}\right\} \rightarrow \sigma_s \tag{19}
$$

C. IVPs for CASS

CASS IVPs are invoked by $\mathcal{S}$ for

1) sending a negative acknowledgement to a user requesting an illegal update; and
2) sending value λ corresponding to any version of any file to an authorized user

IVP $F_{snd}()$ can only send contents determined to be consistent with the current IOMT root ξ.

Given certificates RV-$[f, c_f, \xi]$, RV-$[u_l, a, \alpha]$, FR-$[f, c_f, \alpha_f, q_f]$, RV-$[f, q, \lambda]$ an IVP $F_{snd}()$ is convinced that usrer u_l has access right a for file f. For a non existent file $c_f = 0$, and the other certificates are not necessary. If the user does no have access, or if the file does not exist, the same negative acknowledgement is created, conveying the requested file index f and no other unsolicited information.

If file exists, and if the user has access level $a > 0$ the user is authorized to receive information about any version of f. An additional VR certificate provides information regarding version q of the file where it is possible that $q \neq q_f$ (if the user had requested an older version). The response also includes the most current version number, and the current counter value c_f for the file. IVP $F_{snd}()$ can be represented as follows:

$$
\left.\begin{array}{l}\mathrm{RV[f, c_f, \xi]}\\ \mathrm{RV[u_l, a, \alpha]}\\ \mathrm{FR[f, c_f, \alpha_f, q_f, \xi]}\\ \mathrm{VR[f, q, \lambda]}\end{array}\right\} \Rightarrow \begin{array}{l}\text{IF } ((c_f = 0) \vee (a = 0))\\ \quad \{f\}_{\mathbf{T} \rightarrow u_l}\\ \text{ELSE}\\ \quad \{f, c_f, q_f, q, \lambda\}_{\mathbf{T} \rightarrow u_l}\end{array}
$$

When a user receives the file from $\mathcal{S}$ the user also expects a value κ to be provided by $\mathcal{S}$ (module $\mathbf{T}$ does not care that $\lambda = h(\gamma, \kappa)$ or even the exact mechanism used for computing file hash γ). The user hashes the file to obtain γ and verifies that λ in the authenticated response by the module is consistent with γ and κ. If $\kappa \neq 0$ the user can easily conclude that he/she is eligible to receive a secret with commitment κ.

To relay the stored secret to an authorized user, the service employs another IVP $F_{rs}()$. Given certificates RV-$[f, c_f, \xi]$,RV-$[u_j, a \geq 1, \alpha_f]$, and FR-$[f, c_f, \alpha_f, q_f]$ it can be concluded that user u_j is eligible to receive secret σ corresponding to any version of file f. Along with the three certificates values σ_s and κ are also provided as inputs to a function $F_{rs}()$ to relay the file encryption secret to user u_j. The secret is decrypted as $\sigma = h(\kappa, \chi) \oplus \sigma_s$. The integrity of the decrypted secret is confirmed if $\kappa = h(f, \sigma)$ before it is conveyed to the user (using secret K_j shared with the user). IVP $F_{rs}()$ can be represented as follows:

$$
\left.\begin{array}{l}\mathrm{RV[f, c_f, \xi]}\\ \mathrm{RV[u_l, a, \alpha]}\\ \mathrm{FR[f, c_f, \alpha_f, q_f, \xi]}\\ \sigma_s, \kappa\\ \sigma \leftarrow h(\kappa, \chi) \oplus \sigma_s\\ \kappa = h(f, \sigma)\end{array}\right\} \Rightarrow \sigma_j = h(\kappa, K_j) \oplus \sigma.
$$

D. Summary of Module $\mathbf{T}$ Functionality

Figure 4 provides a broad summary of various certificate types, certificate functions, TPs and IVPs. The vertical dotted line separates generic IOMT functions from CASS specific functions that are aware of the (application specific) interpretation of IOMT leaves.

The two arrows from (for example) NU to EQ illustrates that two NU type certificates are required for obtaining an EQ certificate. From the figure it can be readily seen that two RV certificates are required to generate an LV certificate (certifying latest version in a tree with root θ), and the LV certificate along with an EQ certificate (for insertion of a place-holder for the next version) and RU certificate (for updating the place-holder to a leaf) are required to generate certificate type VU. Certificate generation functions are "helper functions" that are independent of CDIs.

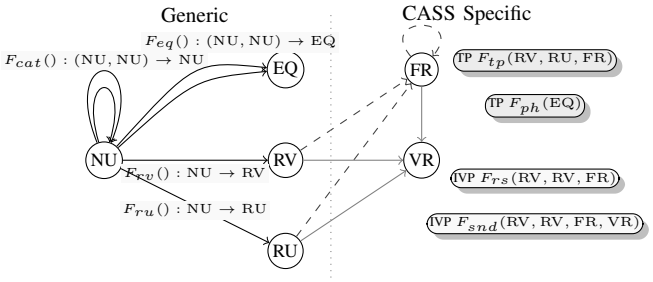

Fig. 4. Generic and CASS-specific functions and certificates.

In the right end of the figure all CASS specific TPS and IVPs are listed along with the types of certificates needed as input. $F_{tp}()$ modifies virtual CDIs, and hence the *real* CDI (IOMT root ξ) stored inside the module. The TP $F_{ph}()$ modifies the real CDI without affecting virtual CDIs (CDI-DB). IVPs do not modify CDIs.

V. Discussions, Related Work and Conclusions

The main differences between CSAA and Clark-Wilson (CW) model are as follows:

Execution of TPs and IVPs: In CSAA TPs and IVPs are constrained to be executed inside the trustworthy boundary of a sufficiently tamper-responsive module $\mathbf{T}$. It is assumed that attempts to expose secrets protected by the module or attempts to modify the module functionality will result in zeroisation of the module (where the module simply erases it's secrets) thereby rendering the module unusable for providing assurances to users. CW model makes no attempt to explicitly define the environment in which TPs are executed.

Constrained structure for CDIs: A second difference lies in deliberately constraining CDIs to be represented as a special data-structure (IOMT). This is to permit any number of CDIs to be tracked by a resource limited module $\mathbf{T}$.

Unconstrained Data Items (UDIs): Both CSAA model and the CW model recognizes the need for unconstrained data items (UDI). CDIs are actually born as UDIs. More specifically, UDIs serve as inputs to the system, and are manipulated by TPs to create CDIs. In the CW model TPs that handle UDIs are required to be verified to meet application specific "separation of duty" requirements. In CSAA, UDIs are authenticated inputs provided by users to the module. They are unconstrained, as a user can request actions on *any* file (even non existent ones) or files to which the user does not have access rights. The ability of the module to answer a simple question (what is the access right of user u for file f) even for non existent files and non existent users in the ACL (thanks to IOMT) is a very important feature required to verify the correctness of UDIs before they are converted into CDIs.

Access to TPs and IVP: The CW model was primarily intended for applications executed on a single computer that supports multiple users. Apart from specification of TPs and IVPs the CW model for a system specifies three-tupes of the form (user, TP,CDIs) that specify which user process can execute a TP, and what CDIs can be accessed/manipulated by the TP. In CSAA any one with access to the module can invoke functions exposed by the module. The untrusted service $\mathcal{S}$ is responsible to ensure that no user can directly access functions exposed by the module to provide deliberately inconsistent information to the module. If the service $\mathcal{S}$ fails to protect access to the module, the service may not be able to demonstrate it's integrity to users.

A. Scalability

Issues that affect the scalability of CSAA can be seen as belonging to two broad categories: a) the number of CDIs, and b) the rate at which events that modify CDIs occur. For a file storage system, the number of CDIs will be roughly proportional to the total number of files. A thousand fold increase (say from a billion to a trillion) will only increase the complexity of each procedure by a factor $40/30$, as the number of hash operations is proportional to $\log_2 N$, which increases from $\log_2 10^9 \approx 30$ to $\log_2 10^{12} \approx 40$.

While the complexity of handling each "event" (triggered by a user request to reserve/update/delete a file or by a request from $\mathcal{S}$ to send parameters related to a version of a file to a user) increases logarithmically, unfortunately, the frequency of events is more likely to increase linearly. A system with 10 times as many files as is likely to handle ten times as many events every second. The second issue can be addressed by deploying CSAA in parallel, using a plurality of identical modules $\mathbf{T}$, where each module tracks only a specific range of file indexes. In this case every user will need to share a key with every module.

B. Scope of Assurances

It is important to note that the CDI databases are very different compared to the actual databases maintained by the service. For example, a real life cloud storage system may use different indexes for each file (to make it possible to readily identify the directory the file is located in, the owner of the file, etc.). The service may also maintain a database of different computers belonging to a user, and a list of folders that need to be synced in each computer. The service may also maintain a database with a record for each user indicating user quota, an the actual space utilized by the user. As the index of a file in the CDI database can be different from the indexing used by the service's databases, the service may maintain another database that enables translation between the two indexes.

Ultimately, the CSAA CDI database is designed *solely based on explicitly desired assurances*. For our purposes, for providing the seven assurances outlined in Section II-B, it is not necessary for the CDI database to track user quota, or the correctness of file syncing operations. However, *if* it is desired to provide additional assurances, like (for example) that "users should not be incorrectly denied access based on storage quotas" or that "every file in a directory is correctly synced to the latest version," then some additional CDIs will be required, demanding additional TPs and IVPs to be executed by module(s) $\mathbf{T}$.

C. Related Work

A closely related work that also aims to assure the operation of a cloud storage service by employing a trusted module is the virtual counter (VC) [12] approach. In the file storage model used in [12] users create files for access by themselves from other locations (no ACLs are used). The goal is to ensure that that only the latest version will be provided by the service (older versions should not be replayed). Specifically, a user may update a file using her computer at work. When she later tries to retrieve the file from her home, she expects the latest version of the file. The assumption here is that the user may not actually remember the previous updates she had made, and thus may not recognize the freshness of the version provided by the service. As the user does not trust the service, she expects the module to certify the file hash corresponding to the freshest version of the file.

In [12] a trusted module assures the integrity of the root of a merkle tree, where the leaves of the tree are *virtual counters* — one associated with every file. The virtual counter associated with a file (in a leaf of the tree) will be incremented by the module only when the file is updated through an authenticated message from the owner of the file. The hash of the new version of the file is bound to the incremented virtual counter value.

As a response to a query for a file, users expect an authenticated response from the module. The module will authenticate only responses that include the file hash bound to the current virtual counter value for the file.

One flaw in the virtual counter approach is that there is no mechanism to prevent the same file from being bound to multiple counters. Thus, when an update is received, the server can request the module to update one counter, while leaving a second counter bound to the older version. From the perspective of the module the other counter is still, legitimate, and can thus be replayed. In [13] it was pointed out that using an IOMT (which guarantees uniqueness of indexes) instead of a plain Merkle tree, can address this flaw.

Several authors [14] -[16] have addressed strategies for providing proof of retrievability (POR) for assuring users of the integrity of files stored in the cloud. The main motivation for such approaches stem from the fact that for large files, hashing entire files to determine hash may be an expensive operation. Most such approaches involve random checking of multiple small segments of a large file to be assured of (with a high probability) the integrity of the file. There is inherently nothing in the proposed approach that mandates that file integrity *has* to be verified by hashing a file. The module $\mathbf{T}$ does not care how the integrity of a file is demonstrated against a commitment γ. The users and the service can choose an appropriate (and convenient) strategy. Instead of a file hash, the value γ can be a concise representation (hash) of values required to verify integrity.

D. Conclusion

With ever growing complexity of hardware and software components, the presence of hidden malicious functionality is a very serious concern. It is thus essential to have strategies in place to provide assurances regarding the operation of complex systems, *without the need to rely on the integrity of complex components*. In the proposed CSAA for assuring cloud storage service $\mathcal{S}$ no component of the service $\mathcal{S}$ is trusted.

As all assurances are bootstrapped from the assumption of integrity of module $\mathbf{T}$ it is necessary to take every possible effort to ensure simplicity of the module. Lower the memory requirement inside the trusted module, the lower is the possibility that hidden malicious functionality can escape detection. The lower the power consumed by the module, the lower the need to dissipate heat, and thus the module can be shielded very well from intrusions that attempt to expose secrets from the module or modify the module functionality [4], [5]. It is for these reasons we limit the module $\mathbf{T}$ to possess a small constant memory size irrespective of the size of the database to be protected, and limit the operations performed by the module to simple sequences of hash operations.

It is important to note that the proposed approach does not seek to improve the reliability of the service $\mathcal{S}$ itself. It does not prevent, for example, a rogue system administrator or an attacker who has gained access to the system from deleting/modifying a file. All that is assured is that, *if* such an event occurs, the system can no longer prove it's integrity to any one who may query the illegally deleted/modified file. The service provider is expected to take all necessary precautions to weed out undesired functionality in their system, to rightfully gain the trust of users.

REFERENCES

[1] M. Borgmann, T. Hahn, M. Herfert, T. Kunz, M. Richter, U. Viebeg, S. Vowe, "On the Security of Cloud Storage Services," SIT Technical Report SIT-TR-2012-001, Fraunhofer Institute for Secure Information Technology, March 2012.

[2] D.D.Clark, D.R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," in Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy (SP'87), May 1987, Oakland, CA; IEEE Press, pp. 184193.

[3] X. C. Ge, F. Polack, R. Laleau, "Secure databases: An analysis of Clark-Wilson model in a database environment," Proceeding of Advanced Information Systems Engineering, 2004, pp 234-247.

[4] S. W. Smith, "Trusted Computing Platforms: Design and Applications," Springer, New York, 2005.

[5] M. Ramkumar, "Trustworthy Computing Under Resource Constraints With the DOWN Policy," IEEE Transactions on Secure and Dependable Computing, pp 49-61, Vol 5, No 1, Jan-Mar 2008.

[6] R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, T. Rabin, "Tamper Proof Security: Theoretical Foundations for Security Against Hardware Tampering," Theory of Cryptography Conference, Cambridge, MA, February 2004.

[7] V. Thotakura, M. Ramkumar, "Minimal TCB For MANET Nodes," 6th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2010), Niagara Falls, ON, Canada, September 2010.

[8] S. Mohanty, A. Velagapalli, M. Ramkumar," An Efficient TCB for a Generic Content Distribution System," International Conference on Cyber-enabled distributed computing and knowledge discovery, Oct 2012.

[9] A. Velagapalli, S. Mohanty, M. Ramkumar,"An Efficient TCB for a Generic Data Dissemination System," International Conference on Communications in China: Communications Theory and Security (CTS), ICCC12-CTS, 2012.

[10] D.Davis, R. Swick, "Network Security via Private-Key Certificates," SIGOPS Oper. Syst. Rev. **24** (4) pp 64–67, Oct. 1990.

[11] R.C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," Advances in Cryptology CRYPTO '87. Lecture Notes in Computer Science 293. 1987.

[12] F.G. Sarmenta, M.V.Dijk, C.W. ODonnell, J. Rhodes, S. Devadas, ”Virtual monotonic counters and count-limited objects using a TPM without a trusted OS,“ *Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing (STC06*, pages 27–42, 2006.

[13] S. D. Mohanty, M. Ramkumar, “Securing File Storage in an Untrusted Server Using a Minimal Trusted Computing Base,” First International Conference on Cloud Computing and Services Science, Noordwijkerhout, The Netherlands, May 2011.

[14] A. Juels, J. Burton S. Kaliski, “PORs: Proofs of Retrievability for Large Files,” Proc. of CCS 07, pp. 584597, 2007.

[15] H. Shacham, B. Waters, “Compact Proofs of Retrievability,” Proc. of Asiacrypt 08, Dec. 2008.

[16] G. Ateniese, R. D. Pietro, L. V. Mancini, G. Tsudik, “Scalable and Efficient Provable Data Possession,” Proc. of SecureComm 08, pp. 1 10, 2008.