# Securing File Storage in an Untrusted Server
## *Using a Minimal Trusted Computing Base*

Somya D Mohanty, Mahalingam Ramkumar
*Missisippi State University, MS, USA*
*sdm281@msstate.edu, ramkumar@msstate.edu*

Keywords: Merkle Trees, File Storage System, Index Ordered Merkle Trees, Authenticated Denial, Trusted Computing Base

Abstract: In applications such as remote file storage systems, an essential component of cloud computing systems, users are required to rely on untrustworthy servers. We outline an approach to secure such file storage systems by relying only on a resource limited trusted module available at the server, and more specifically, without the need to trust any component of the server or its operator(s). The proposed approach to realize a trusted file storage system (TFSS) addresses some shortcomings of a prior effort (Sarmenta et al., 2006) which employs a merkle hash tree to guarantee freshness. We argue that the shortcomings stem the inability to verify *non-existence*. The TFSS described in this paper relies on index ordered merkle trees (IOMT) to gain the ability to verify non-existence.

# 1 INTRODUCTION

In a variety of services based on the client-server paradigm, servers are middle-men that facilitate interactions between end-points that are not capable of directly interacting with each other. For example, a plurality of email servers are the middle-men involved in relaying content (email) created by one user to another. A web server relays content created by a user (the author) to web-clients. A DHCP server relays an IP address to a client from a pool of IP addresses provided by an administrator. A file storage server permits users to store their files at a central location for later access from any location - perhaps even by other users.

From a security perspective the middle-men should remain *transparent* to end-points. In other words, *any assurance that can be achieved if end-points can directly interact with each other should be realized despite the fact that they rely on middle men to do so*. This requirement is especially important for a broad class of evolving services like cloud computing, with ever increasing scope of user data residing at untrusted servers.

One approach to extend some assurances to tasks performed by servers is to mandate use of trusted platforms for running the server. In the trusted computing group (TCG) approach to realize a trusted platform, a trusted platform module (TPM) housed in a general purpose platform is leveraged to assure the integrity of the platform (tcg, 2007). The specific goal of the TCG approach is to ensure that only pre-verified and authorized software can take control of the platform. For this purpose it is however required to trust several components of the platform in addition to the TPM chip. Such additional components include the CPU, RAM, CPU-RAM bridge, and any peripheral that may have direct access to the RAM. Our inability to provide reasonable assurances to such components is the fundamental reason behind the well-known time-of-use-time-of-check (TOCTOU) (Bratus et al., 2008) problem that plagues the TCG approach. Another disadvantage of the TCG approach is the impracticality of verifying and certifying complex and dynamic software components.

For any computing system with a desired set of assurances, the trusted computing base (TCB) is "a small amount of software and hardware we rely on, and that we distinguish from a much larger amount that can misbehave without affecting secu-

rity" (Lampson et al., 1992). That the TCB *needs* to be trusted does not imply that the TCB is *worthy* of trust. Thus, it is necessary to take every possible measure to ensure that the TCB is worthy of trust, by *minimizing* the TCB to the extent feasible.

In this paper we investigate security solutions for servers which rely *only* on a trusted module like a TPM chip as the TCB - no other hardware or software is trusted. More specifically, we investigate the additional TCB functionality that needs to be offered by current TPM chips to serve as a reliable and comprehensive TCB for securing a broad class of services. As an example of leveraging such a TCB to realize trusted services, we outline the design of a trusted file storage system (TFSS).

## 1.1 TFSS Using a Trusted Module

In a file storage service user generated data from various client machines are uploaded to a centralized server, for convenient retrieval by the creator of the content, or by any entity authorized to do so by the creator, from any location. From a security perspective, users desire that the server will not manipulate user created data in unauthorized ways. Unauthorized manipulation of data can occur in various ways such as i) providing the user with a file different from the requested file, ii) providing the user with a older version of the requested file, iii) hiding the requested file, iv) denying access to the requested file (when the user is in fact authorized to access the file) etc.

In our model an untrusted server **U** (a middle man) has access to a trusted module **T**. The module **T** is trusted by all users. When users provide data to the server **U**, they expect an acknowledgement authenticated by **T**; when some information is provided by the server **U** to the user, the user expects such responses to be authenticated by **T**.

Our approach suggests that data records be accompanied with auxiliary records which specify relationships between records. An example of such a mechanism already in use today is NSEC (Weiler and Ihren, 2006) records in the DNS security protocol (DNSSEC) (Arends et al., 2005). NSEC records indicate a pair of names; a pair (A, B) conveys a specific relationship between records A and B - that B follows A in the dictionary order. More importantly, the existence of the NSEC record (A, B) is interpreted to mean that no record for a name that falls between A and B (in the dictionary order) exists.

Through well-defined interfaces exposed by the module **T**, the server **U** submits data to **T** for cryptographic authentication/verification. The data submitted by **U** will be authenticated by **T** only if the server **U** can demonstrate that some conditions necessary for this purpose have been met. The tasks executed by the module **T** to *verify that necessary preconditions have been met* is the "TCB functionality." This TCB functionality will be leveraged to ensure that untrusted servers cannot manipulate user data in unauthorized ways. It is desirable that this TCB functionality is simple enough to be implemented inside highly resource limited modules (like TPM chips).

### 1.1.1 Merkle Trees

In (Sarmenta et al., 2006) the authors suggest that enhancing the ability of current TPM chips, by including new TPM functionality to maintain a Merkle hash tree can be used for securing a wide variety of applications, including a trusted file storage system. A merkle hash tree has been employed in several realizations trustworthy computing applications to amplify trust, as this data structure can permit a resource limited module **T** to *virtually store* a large number of *dynamic* values in an untrusted location. Specifically, the leaves of the tree are dynamic values. Only a single value, the root of the tree, will need to be stored inside the module **T**. The module can verify the integrity of any of the $N$ dynamic leaves by computing $\log_2 N$ hashes (for example, 20 hash operations to verify the integrity of any of 1 million leaves).

For *true virtual storage* of a large number of values (leaves) it is *not* sufficient for the module to be able to merely verify a leaf (of a merkle tree) against the root; an equally important requirement is also the ability to verify that a specific leaf does *not* exist in the tree. We outline some replay attacks, and discuss several shortcomings of the approach in (Sarmenta et al., 2006) - all of which result from the inability of the trusted module to verify *non-existence* of leaves.

An important feature of the Merkle hash tree is that each leaf of the tree is independent. Any leaf can be inserted, verified, updated, and deleted, and the root modified to reflect the change, while ignoring the other leaves in this process. Unfortunately, this feature is also the reason for this specific inadequacy of merkle trees - viz., the inability to verify non-existence. This inadequacy can be addressed by using an index ordered merkle tree (IOMT) (Thotakura and Ramkumar, 2010) in which the leaves are *not* independent.

In this paper we enumerate additional TCB functionality to provide TPMs with the ability to maintain IOMTs (instead of plain merkle trees as suggested in (Sarmenta et al., 2006)), and describe how this functionality can be leveraged to realize a trusted file storage system on an untrusted server by relying *only* on the TPM. Apart from addressing replay attacks on

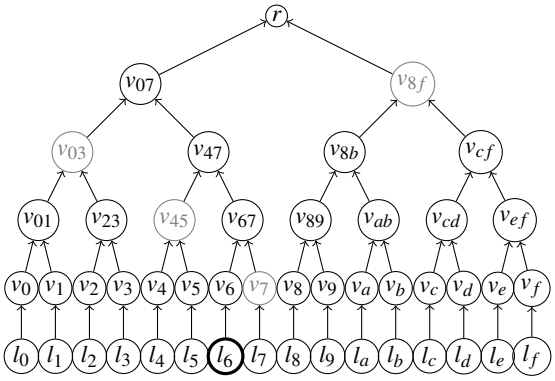

Figure 1: A Binary Merkle tree with 16 leaves. The complementary nodes for $l_6$ are $v_7$, $v_{45}$, $v_{03}$ and $v_{8f}$.

the scheme in (Sarmenta et al., 2006) the TFSS proposed in this paper has substantially enhanced functionality including i) protecting the privacy user data from untrusted servers; ii) the ability to specify and enforce sophisticated read/write access control to the files; and iii) ensuring that the untrusted server cannot deny access to authorized entities.

### 1.2 Organization

The rest of this paper is organized as follows. In Section 2 we provide a brief overview of Merkle trees and the virtual counter approach in (Sarmenta et al., 2006)) to realize a TFSS. We then outline a simple replay attack against the virtual counter scheme.

In Section 3 we provide a brief overview of the index ordered Merkle tree (IOMT), and a description of algorithms to *insert*, *delete*, *update* and *deny* leaves of the IOMT. the index ordered Merkle tree (IOMT). The mechanism for leveraging this TCB to realize a TFSS is described in Section 3.4. Related work and conclusions are offered in Section 4.

## 2 Background

In this section we provide an overview of merkle hash trees, and outline the virtual counter approach in (Sarmenta et al., 2006) to realize a secure file storage system.

### 2.1 Merkle Trees

A binary merkle hash tree is constructed using a cryptographic hash function $h()$ (for example, SHA-1). A tree of height $L$ has $N = 2^L$ leaves and $2N-1$ nodes at $l$ levels. Figure 1 displays a merkle tree with $N = 16$ leaves (height $L = 4$). The $N$ nodes $v_0 \cdots v_f$ at level 0 are obtained by hashing the leaf using the cryptographic hash function $h()$. Two adjacent nodes in each level (a left node $v_l$ and a right node $v_r$) are hashed together to yield a parent node $h(v_l \parallel v_r)$ one level above. The lone node at the top of the tree is the root $r$, which is a commitment to all leaves.

Corresponding to any leaf $l_i$ is a set of $L$ complementary nodes (one in each level). For example, the complementary nodes for $l_6$ are $v_7$, $v_{45}$, $v_{03}$ and $v_{8f}$. The complementary nodes for a leaf $l_i$ are actually *commitments for all nodes except* $l_i$.

More specifically, each complementary node is associated with a bit - representing right (0) or left (1) - depending on the position of the complementary node relative to the leaf. The $L = 4$ two-tuples $\{(v_7,0),(v_{45},1),(v_{03},1),(v_{8f},0)\}$ associated with leaf $l_6$ are the *instructions* for mapping a leaf to the root. For example, following the instructions, we can compute the root starting from $v_6 = h(l_6)$ as

$$\begin{array}{ll} v_{67} = h(v_0 \parallel v_7) & v_{47} = h(v_{45} \parallel v_{67}) \\ v_{07} = h(v_{03} \parallel v_{47}) & r = h(v_{07} \parallel v_{8f}) \end{array} \quad (1)$$

Note that the bit specifies the ordering of two nodes before hashing them together to compute the parent node. In the rest of this paper we shall denote a set of instructions for a leaf $l_i$ as $\mathbf{I}_i$ and the process of mapping a leaf to the root of the tree as

$$r = l2r(l_i, \mathbf{I}_i). \quad (2)$$

### 2.2 Merkle Trees in Trustworthy Computing

A resource limited trusted module capable of storing the root and executing function $l2r()$ can verify the integrity of a leaf against the root. The leaves and other internal nodes can be stored in an untrusted location. An untrusted entity **U** desiring to demonstrate that a leaf $l_i$ is genuine provides the instructions $\mathbf{I}_i$ necessary to map the leaf to the root.

If the module determines that $l2r(l_i, \mathbf{I}_i) = r$, the module is simultaneously assured of i) the integrity of the leaf $l_i$, *and* ii) the integrity of the instructions $\mathbf{I}_i$. More specifically, as long as the hash function $h()$ is pre-image resistant, it is infeasible to determine alternate values $\tilde{l}_i \neq l_i$, and $\tilde{\mathbf{I}}_i \neq \mathbf{I}_i$ that will satisfy $l2r(\tilde{l}_i, \tilde{\mathbf{I}}_i) = r$.

To legitimately update a leaf $l_i$ to $l_i'$ the untrusted entity **U** provides the current leaf $l_i$, the set of values $\mathbf{I}_i$ (to authenticate the current leaf against the root), and the new leaf $l_i'$ suitably authenticated by an entity responsible for the contents of the leaf $l_i$. The module **T** i) verifies that $l2r(l_i, \mathbf{I}_i) = r$; ii) verifies the authentication appended for the new leaf $l_i'$, and iii) replaces $r$ with $r' = l2r(l_i', \mathbf{I}_i)$. After the root has been modified,

the untrusted entity **U** will not be able to convince the module of the integrity of the old leaf $l_i$ (as it can no longer be authenticated against the current root).

In practice, a trusted module **T** capable of maintaining the root of a dynamic merkle tree will need to expose interfaces to i) *insert()* a leaf, ii) *delete()* a leaf, iii) *bind()* a leaf to application data (for example, hash of a file), iv) *update()* a leaf based based on authenticated data, and v) *send()* authenticated application data (bound to a leaf) to entities authorized to receive the data.

## 2.3 The Virtual Counter Approach

A typical client-server model of file storage system includes clients (users) who create files or data blobs and a server **U** with access to plentiful storage. Users upload files to the centralized server for later retrieval from any location. Files can be edited and reuploaded any number of times. As a user may employ different client machines at different locations (work, home, on the road), and as the files in different client machines may not be synchronized (except through the untrusted server), users have to trust the server to provide the latest version of the file.

In the approach based on virtual counters (Sarmenta et al., 2006) realized using a merkle tree, a trusted module **T** associated with the untrusted server **U** is intended to eliminate the need to blindly trust the server. The specific goal was to ensure that untrusted servers will not be able to replay older versions of files.

A prerequisite for the virtual counter approach is a mechanism for mutual authentication between users and the module **T** - the exact nature of the which is irrelevant for our discussions. Along with an updated file, users convey an authenticated file-hash to the server - which is intended for the module **T**. In response, users expect an authenticated confirmation by the trusted module **T** to the effect that "the requested update has been carried out."

A merkle tree is used to virtually store a large number of counters $c_0 \cdots c_N$. Each counter is cryptographically bound to a file - say uniquely identified by values $id \parallel l$, where $id$ is the unique identity of a user and $l$ is a label assigned to the file by the user. Every time a file is changed the corresponding counter is incremented and the root is updated.

In (Sarmenta et al., 2006) even the root is not stored inside the module. Instead, only a non resettable monotonic counter is stored inside the module, and the root is cryptographically bound to the counter. The reason for this choice is that the latest TPM specification already includes such a non resettable monotonic counter.

If the root is $r$ when the counter is $c$ the module issues a *self-verifiable certificate* binding values $r$ and $c$. Such a certificate can be simple message authentication code (MAC) $\mu$ computed using a secret $K_s$ known only to the module **T** as

$$\mu = h(r \parallel c \parallel K_s). \tag{3}$$

The values $\mu$ and $r$ can be provided to the module by the server **U** to permit the module to recognize that the current root is $r$, as long as the counter is $c$.

Similarly, a file $id \parallel l$ with file-hash $h_{id,l}$ is bound to the $i^{\text{th}}$counter $c_i$, through a self-certificate

$$\mu_i = h(i \parallel c_i \parallel id \parallel l \parallel h_{id,l} \parallel K_s) \tag{4}$$

Every time a file is updated the counter bound to the file-hash is incremented causing the root $r$ to change. Every time the root $r$ changes the counter $c$ is incremented. New self-certificates are issued by **T** to i) bind the updated root and the counter; and ii) bind the incremented $c_i$ to the new file hash.

To send the current version of the file to a user the server requests the module to authenticate the hash of the current file by submitting the following values: i) $i \parallel c_i \parallel id \parallel l \parallel h_{id,l}$; ii) self certificates $\mu_i$ and $\mu$, and iii) instructions $\mathbf{I}_i$ to map $c_i$ to the root. The module will honor this request only if the self-certificates are consistent and $r = l2r(c_i, \mathbf{I}_i)$ is consistent with the value $r$ in $\mu = h(r \parallel c \parallel K_s)$. The module can now authenticate the values $id \parallel l \parallel h_{id,l}$ to the user.

A counter $c_i$ bound to file $id \parallel l$ will be updated by the module only if an authenticated hash $h'_{id,l}$ (authenticated by the owner $id$) is provided to the module. Apart from the authenticated hash the module will be provided inputs ($i \parallel c_i \parallel id \parallel l \parallel h_{id,l}$; self certificates $\mu_i$ and $\mu$, and instructions $\mathbf{I}_i$) required to verify the current leaf against the root. To complete this update the module

1. increments $c_i$ to $c'_i = c_i + 1$;
2. updates the root of the tree to $r' = l2r(c'_i, \mathbf{I})$ to reflect the incremented leaf;
3. increments the counter $c$ to $c' = c + 1$;
4. issues a self-certificate binding values $r'$ and $c'$;
5. issues a self certificate binding $c'_i$ with the updated hash $h'_{id,l}$; and
6. outputs an authentication token verifiable by user to the effect that the "update has been carried out."

### 2.3.1 Security Loopholes

It would seem reasonable at first sight to assume that when a user has received a confirmation from the

module **T** that the "update has been carried out" the server can no longer replay older versions of the file - as the module will not authenticate the old hash. Specifically, after the update to the root, the module will not recognize the hash of older version as authentic. Unfortunately, this is not true.

The security loop-hole in the scheme is that there is no way to prevent the server from binding the same file $id \parallel l$ to multiple (say 2) counters $c_u$ and $c_v$. Now, following the first update the server instructs the module to update counter $c_u$ and issue a confirmation to the user, and deliberately leaves counter $c_v$ intact. The old file corresponding to counter $c_v$ can be replayed by the server as $c_v$ is still a part of the tree. More generally, the server may associate a file with any number of counters (leaves) and maintain different older updates in such leaves.

A second security pitfall of the virtual counter approach is the inability to provide authenticated denial. Consider a scenario where a user submits a request for a non existent file $id \parallel l$. As no verifiable leaf bound to $id \parallel l$ exists and the module can only make reliable statements about the leaves of the tree, the module cannot authenticate a denial which conveys the non-existence of $id \parallel l$. In such a scenario the untrusted server is implicitly trusted to convey non-existence. The unfortunate side-effect is that the untrusted server can abuse this privilege by "conveying non-existence" of files that *do* exist.

At the core of both problems is the inability of the module to verify non-existence of leaves. If the server can easily verify that *no* counter has been bound to file $id \parallel l$ then the module will permit binding of $id \parallel l$ to a counter $c_i$ only if the module can verify that $id \parallel l$ does not currently belong to the tree. In such a scenario the server cannot force the module to bind $id \parallel l$ to multiple leaves. Similarly when queried for a non-existent file a module which can easily verify non-existence can provide authenticated denial, and thus eliminating the need to trust the server. This feature can be provided to merkle trees through a simple extension - the index ordered merkle trees (IOMT) (Thotakura and Ramkumar, 2010).

# 3 TCB Functions Based on IOMT

In an IOMT a leaf corresponding to some index $a$ is of the form $a \parallel \theta_a \parallel a'$, where $a'$ is also an index. For our purposes the index can be a function of the unique values that identify a file, say $a = h(id \parallel l)$. The value $\theta_a$ is a value corresponding to the index $a$. The set of uniquely indexed current leaves is an ordered list where every index points to the next higher index - the exception is the highest index which wraps around and points to the least index.

A value $x$ is enclosed by $(a,a')$ if $(a < x < a')$, or (for the wrapped around pair) if $(x < a' \leq a)$ or $(a' \leq a < x)$. That the record for $a$ indicates $a'$ as the next record is proof that "no leaf exists in the tree for an index that is *enclosed* by $a$ and $a'$." If $a = a'$ *all* values except $a$ are enclosed, and implies that $a$ is the *sole* index in the tree. A function that verifies that a value $i$ is enclosed by $(j,k)$ can be described algorithmically as

$$
\begin{array}{l}
encl((i,j),k)\ \{ \\
\text{RETURN } ((i<k<j) \vee ((i>j) \wedge (k>i)) \vee \\
\qquad\qquad\qquad ((i>j) \wedge (k<j))); \\
\}
\end{array}
$$

In an IOMT an empty leaf is of the form $(0,0,0)$. Shown below is the sequence of changes that occur as leaves corresponding to indexes $c, y$ and $d$ are inserted and then, $y$ is deleted, starting with an empty tree (with all leaves set to $(0,0,0)$).

| | |
|---|---|
| Empty | $\{(0,0,0),(0,0,0),(0,0,0),(0,0,0)\cdots\}$ |
| $c$ inserted | $\{(c,\theta_c,c),(0,0,0),(0,0,0)\cdots\}$ |
| $y$ inserted | $\{(c,\theta_c,y),(y,\theta_y,c),(0,0,0)\cdots\}$ |
| $d$ inserted | $\{(c,\theta_c,d),(y,\theta_y,c),(d,\theta_d,y)\cdots\}$ |
| $y$ removed | $\{(c,\theta_c,d),(0,0,0),(d,\theta_d,c)\cdots\}$ |

In general to insert a leaf for index $c$, a leaf $a$ with pointer $a'$ such that $(a,a')$ encloses $c$ should be provided. After insertion $a$ will point to $c$ and $c$ will point to $a'$. When a leaf $x$ which currently points to $x'$ is to be deleted, a leaf $b$ that points to $x$ (or $b' = x$ should be provided. After deletion $b' = x$ is set to $b' = x'$;

Note that (apart from the insertion of the first leaf) insertion or deletion of a leaf will require two leaves to be modified simultaneously. The complementary instructions $\mathbf{I}_{lr}$ to map two leaves $\mathbf{L}_l$ and $\mathbf{L}_r$ simultaneously to the root are

1. the instructions $\mathbf{I}'_l$ to map $\mathbf{L}_l$ to $\xi_l$ where $\xi_l$ is left child of the common parent $p$ of $\mathbf{L}_l$ and $\mathbf{L}_r$;
2. the instructions $\mathbf{I}'_r$ to map $\mathbf{L}_r$ to $\xi_r$ where $\xi_r$ is right child of the common parent $p$;
3. the instructions $\mathbf{I}'_p$ to map the common parent to the root $r$.

For example, to map $l_1$ and $l_3$ (with common parent $v_{03}$ the instructions are $\mathbf{I}'_1 = \{(v_0,1)\}$, $\mathbf{I}'_3 = \{(v_2,1)\}$, and $\mathbf{I}'_p = \{(v_{47},0),(v_{8j},0)\}$. As another example, to map $l_2$ and $l_7$ (with common parent $v_{07}$, the instructions are $\mathbf{I}'_2 = \{(v_3,0),(v_{01},1)\}$, $\mathbf{I}'_7 = \{(v_6,1),(v_{45},1)\}$, and $\mathbf{I}'_p = \{(v_{8j},0)\}$.

For notational convenience we represent the three sets of instructions $\mathbf{I}'_l$, $\mathbf{I}'_r$ , and $\mathbf{I}'_p$ as $\mathbf{I}_{lr}$. We shall denote by

$$r = lvs2r(\mathbf{L}_l, \mathbf{L}_r, \mathbf{I}_{lr}) \tag{5}$$

the process to map two leaves simultaneously to the root $r$. Specifically, to modify two leaves $\mathbf{L}_l$ and $\mathbf{L}_r$ (to say $\mathbf{L}'_l$ and $\mathbf{L}'_r$) the module i) first verifies that $r = lvs2r(\mathbf{L}_l, \mathbf{L}_r, \mathbf{I}_{lr})$ and ii) computes the new root as $r' = lvs2r(\mathbf{L}'_l, \mathbf{L}'_r, \mathbf{I}_{lr})$.

## 3.1 Interface InsDel()

A trusted module which maintains an IOMT will expose an interface InsDel($\mathbf{L}_l, \mathbf{L}_r, x, \mathbf{I}_{lr}$) to add or delete leaves of the IOMT. Using the interface InsDel() the module is provided two leaves $\mathbf{L}_l = (i \parallel \theta_i \parallel i')$ and $\mathbf{L}_r = (j \parallel \theta_j \parallel j')$. If at least one of the two leaves is empty, the module interprets this as a request for inserting a leaf for an index $x$ in the place of an empty leaf (the module will honor this request only if the other leaf encloses $x$).

If both leaves are non empty and one leaf corresponds to an index $x$ (if $i = x$ or $j = x$) then that leaf is replaced by an empty leaf - but only only if the other leaf points to $x$. The pointer of the other leaf is then changed to the index that was pointed to by $x$ (the leaf to be removed). When a leaf for index $x$ is inserted the values $\theta_x$ is automatically set to zero. A leaf $x$ of the form $(x \parallel \theta_x \parallel x')$ can be deleted only if $\theta_x = 0$.

To insert a leaf the function InsDel() expects $i = x$ or $j = x$. If (say) $i = x$ the $(j, j')$ should be such that $i = x$ is enclosed - to prove that no leaf for index $x$ exists currently. Apart from the process of insertion and deletion of leaves the operations required for maintaining an IOMT are not different from those required for maintaining a plain merkle tree. In the rest of this section we describe three other functions Update(), Send(), and Deny() which together with InsDel() will serve as the TCB for a wide range of services including a trusted file storage service.

In our model we assume that every user has a unique identity and that every user shares a unique secret with the module **T**. While the module is required to expose interfaces to facilitate establishment of shared secrets with users, *we do not discuss such functionality in this paper*. We simply assume that a module **T** shares a secret $K_{id}$ with user $id$.

## 3.2 Principle of Operation

In most client-server applications users submit data-blobs to servers, and such blobs may be later retrieved by the same user or other users. To ensure that data is not modified whilst residing at the untrusted server **U**, the hash of the data is submitted to, and retrieved from, a trustworthy entity: the module **T**.

The blob retrieved from the untrusted middle man **U** is verified against the hash received from the trusted entity. Corresponding to every blob (with a unique identifier) is a leaf in the IOMT. The hash of a blob is bound to the corresponding IOMT leaf. In our approach a blob is uniquely identified by the identity $id$ of the owner and the label $l$ assigned to the blob by the owner of the blob. More specifically, the index for a blob $id \parallel l$ is $x = h(id \parallel l)$.

We also desire that when a server is queried for some blob the server should not be able to simply deny existence of the queried blob. The server is required to prove to the module **T** that the queried blob does not exist.

If some blob with index $x$ does not exist, then there must exist a leaf for some blob index $a$ with a pointer $a'$ such that $(a, a')$ encloses $x$. The server is required to prove non existence of $x$ to the module by submitting a valid leaf containing the enclosure $(a, a')$ for $x$. Only after verifying that the queried data does not exist, the module will authenticate a message conveying non existence of the queried blob.

To ensure privacy, the blob submitted to the server is encrypted. To convey the secret $S_q$ used for encrypting the $q^{\text{th}}$version of the blob $id \parallel l$, submitted by a user $id_r$ (who shares a secret $K_{id_r}$ with **T**), the value provided to the module is

$$s' = S_q \oplus h(K_{id_r} \parallel id \parallel l \parallel q). \tag{6}$$

The module **T** decrypts $s$ to obtain $S_q$ and re-encrypts $S_q$ using a secret $K_s$ known only to **T** as

$$s = S_q \oplus h(K_s \parallel id \parallel l \parallel q). \tag{7}$$

the encrypted secret $s$ is stored by the untrusted server, and is also bound to the leaf of the IOMT corresponding to index $x = h(id \parallel l)$. When a blob $id \parallel l$ is queried by a user $id'_r$ the querier submits an authenticated nonce $v$. The stored encrypted secret $s$ is conveyed to $id'_r$ as

$$\begin{aligned} s'' &= s \oplus h(K_s \parallel id \parallel l \parallel q) \oplus h(K_{id'_r} \parallel v) \\ &= S_q \oplus h(K_{id'_r} \parallel v). \end{aligned}$$

Another common requirement in many applications is for the creator of the blob $id \parallel l$ to be able to specify a list if entities who are authorized access to the blob. This is accomplished through a value $\alpha)id, l$, bound to the leaf corresponding to $id \parallel l$. The value $\alpha_{id,l}$ is actually the root of another IOMT (which we shall refer to as the access-control IOMT), with one leaf for every authorized entity.

Specifically, the leaf corresponding to an entity $y$ is of the form $y \parallel w_y \parallel y'$. If such a leaf can be verified by the module to be a part of the tree with access control IOMT-root $\alpha_{id,l}$, the module recognizes that

1. entity $y$ has read and write access (to file $id \parallel l$) *if* $w_y = 1$;

2. entity $y$ has only read access if $w_y = 0$; and more importantly,
3. no entity with an identity enclosed by $(y, y')$ has *read or write* access.

The access control IOMT for a blob $id \parallel l$ is created by the owner $id$, and the entire IOMT is handed over to the untrusted server; and the root $\alpha_{id,l}$ is authenticated by the user and conveyed to the module **T** (and bound to the leaf of the main IOMT with index $x = h(id \parallel l)$, the root of which is stored by the module). When a user $id_r$ requests read or write access to a blob $id \parallel l$ the server is required to explicitly prove to the module **T** that the user $id_r$ does (or does not) have the required permission.

## 3.3 Data Structures

The module recognizes three data structures - an IOMT leaf **L**, a record **R**, and an authentication record **A**. An IOMT leaf is of the form

$$\mathbf{L} = [x \parallel \theta_x \parallel x']. \tag{8}$$

A blob-record **R** is of the form

$$\mathbf{R} = [id \parallel l \parallel q \parallel d \parallel a \parallel s] \tag{9}$$

where $q$ is a sequence number (similar to a virtual counter, $q$ is updated every time the blob $id \parallel l$ is modified); $d$ is some data associated with the blob (which can be the hash of the blob); $a$ is an auxiliary value; and $s$ is a secret associated with the blob.

The module does not care about the auxiliary value $a$. The value $a$ can be modified by users who have write access to the file. The format and interpretation of the auxiliary value is left to the users. As an example, a user $id$ may use that value to indicate to another user that he/she is currently modifying the file. The value $a$ could also be a one way function of any other auxiliary information about the file like the time of creation, submission, etc.

Also associated with blob $id \parallel l$ is the access control value $\alpha$. From the perspective of the module **T**, a record **R** and a value $\alpha$ are bound to a leaf $x \parallel \theta_x \parallel x'$ if

$$\begin{aligned} x &= h(id \parallel l) \text{ and} \\ \theta_x &= h(h(\mathbf{R}) \parallel \alpha). \end{aligned} \tag{10}$$

An authentication record **A** of the form

$$\mathbf{A} = [id_r \parallel v \parallel \tau \parallel \mu] \tag{11}$$

where $id_r$ is the identity of the entity who has authenticated values $v$ and $\tau$ using a MAC $\mu$. An authentication record is always implicitly associated with some $id \parallel l$ and the MAC is computed as

$$\mu = h(id \parallel l \parallel v \parallel \tau \parallel K_{id_r}) \tag{12}$$

The field $\tau$ in an authentication record **A** indicates the *type* of the authentication record and specifies the *nature* of the request from users. The types include

1. *INR* (initialize record);
2. *CLR* (clear record);
3. *CAC* (change access control value);
4. *RRQ* (read request); and
5. *WRQ* (write request);

The response from the module to an entity $K_{id_r}$ is a MAC computed using a secret $K_{ack} = h(K_{id_r} \parallel 1)$, and is computed over a value $v'$ and a response type $\tau'$ as

$$\mu_{ack} = h(v' \parallel \tau' \parallel K_{ack}). \tag{13}$$

More specifically, from the perspective of the module **T**, an authentication record **A** provided to the module is a query from a user. The value $\mu_{acK}$ is the response to the query from the module. To elucidate this response the untrusted server will need to provide some other values to the module **T**. Such other values typically include

1. a record **R**, a value $\alpha$, and a leaf **L** bound to **R** and $\alpha$, along with instructions **I** to map the leaf to the root, and
2. another IOMT leaf $\mathbf{L}'$ and instructions $\mathbf{I}'$ to map the leaf to the access-control root $\alpha$ - which will enable **T** to verify if the querier does (or does not) have the requested privilege.

If the query by $id_r$ is successful the response type $\tau'$ is the same as the type $\tau$ in **A**. A query can be unsuccessful only if i) the queried file does not exist (response type $\tau' = DNE$), or ii) if the user $idr_r$ is not authorized ($\tau' = NOA$); or iii) if the user has requested an illegal update ($\tau' = ILU$). In the following sections we describe the functionality of the three interfaces offered by **T**. The pseudo-code for all TCB functions are included in Appendix A.

### 3.3.1 Send$(\{\mathbf{L}, \mathbf{I}, \mathbf{R}, \alpha\}, \{\mathbf{A}\}, \{\mathbf{L}', \mathbf{I}'\})$

The first four inputs are required to check the current values for $id \parallel l$ - by verifying $x = h(id \parallel l)$, $\theta_x = h(h(\mathbf{R}) \parallel \alpha))$, and $r = l2r(\mathbf{L}, \mathbf{I})$.

The Send() function expects type $\tau == RRQ$ in $\mathbf{A} = [id_r \parallel v \parallel \tau \parallel \mu]$. In this case the value $v$ in **A** is simply a nonce. The function verifies MAC $\mu$ and proceeds to determine if $id_r$ has read access. Entity $id_r$ has read access if $id_r = id$, or $\mathbf{L}' = z \parallel w_z \parallel z'$ is such

$$z = id_r \text{ and } \alpha = l2r(\mathbf{L}', \mathbf{I}'). \tag{14}$$

If $id_r$ does *not* have read access the server is required to submit a leaf $\mathbf{L}'$ with values $(z, z')$ that encloses $id_r$.

If $id_r$ has read access, the function authenticates the record $\mathbf{R}$ to $id_r$ after decrypting and re-encrypting the secret $s$. If $id_r$ does not have read access the output is a MAC computed over values $v$ indicating type $\tau' = NOA$. The output of the function is a MAC (verifiable by $id_r$) and an encrypted secret (which can be decrypted by $id_r$).

#### 3.3.2 Deny$(\mathbf{L},\mathbf{I},\mathbf{A},id',l')$

This function first verifies that $r = l2r(\mathbf{L},\mathbf{I})$, the MAC in $\mathbf{A}$, and then verifies that $encl((x,x'),h(id' \parallel l'))$ is TRUE. The output is a MAC verifiable by $id_r$ for the value $v' = v$ and type $\tau' = DNE$.

#### 3.3.3 Update$(\{\mathbf{L},\mathbf{I},\mathbf{R},\alpha\},\{\mathbf{R}',\mathbf{A}\},\{\mathbf{L}',\mathbf{I}'\})$

As in the Send() function, the fist four inputs are required to verify a stored record against the root. The authentication record types submitted to Update() can be of the following types: $\tau \in \{CLR,RST,INR,WRQ\}$. Authentication records with type $\tau == WRQ$ will be honored if $id_r$ is the owner ($id = id_r$), or $\mathbf{L}' = (z \parallel w_z \parallel z')$ is such that $\alpha == l2r(\mathbf{L}',\mathbf{I}')$ and

$$z = id_r \text{ and } w_z = 1. \tag{15}$$

If $id_r$ does *not* have write access, the function Update() expects a leaf $\mathbf{L}'$ with $\alpha == l2r(\mathbf{L}',\mathbf{I}')$ such that

$$\begin{aligned} &z = id_r \text{ and } w_z = 0 \text{ OR} \\ &encl((z,z'),id_r) \text{ is TRUE} \end{aligned} \tag{16}$$

For type $\tau == WRQ$ the value $\mathbf{R}' = [id' = id \parallel l' = l \parallel q' \parallel d' \parallel a' \parallel s']$ is the replacement record for $\mathbf{R}$, and the value $v$ in $\mathbf{A}$ is $v = h(\mathbf{R}')$. The update will occur only if $q' > q$.

If the field $s' = 0$ in $\mathbf{R}'$ the implication is that the new blob submitted to the server is not encrypted. All values are copied from $\mathbf{R}'$ to $\mathbf{R}$ as-is, except the field $s$. The hash $v' = h(\mathbf{R})$ for the modified $\mathbf{R}$ is computed with field $s$ set to $S_q$. Following this the field $s$ is re-encrypted for storage; the values $v'' = h(\mathbf{R})$ and $\alpha$ are used to recompute $\theta_x$, and update the root.

If the write request is successful the function returns i) the encrypted secret $s$ which will be stored by the server, and ii) a verifiable (by $id_r$) MAC for the value $v'$ to confirm that the update has been carried out successfully.

If the request is unsuccessful because $id_r$ does not have write access the function outputs a MAC indicating type $\tau' = NAU$. If the update request is illegal (for example $q' \leq q$ ) the function returns a MAC indicating type $\tau' = ILQ$.

Types $CLR,RST$ and $INR$ are honored only if the sender is the owner (or $id_r = id$). For types $CLR$ and $RST$ the inputs $\mathbf{R}'$, and $\{\mathbf{L}',\mathbf{I}'\}$ are empty.

If the type is $\tau == CLR$ the value $\theta_x$ is set to zero, and the modified leaf $\mathbf{L}$ mapped to a new root using $l2r(\mathbf{L},\mathbf{I})$.

If the type is $\tau == CAC$ the value $v$ in $\mathbf{A}$ is interpreted as the new value of $\alpha$. The value of $\theta_x$ is recomputed to reflect this change and the new root is computed.

Type $INR$ can be submitted only if the leaf $\mathbf{L}$ has not been initialized. Recall that in the function InsDel(), when a leaf for $x = h(id \parallel l)$ has been inserted, the value $\theta_x$ is set to zero. Thus, while a place holder has been created, no record has been bound to the leaf. When the type is $INR$, the first two fields $id$ and $l$ are set to the values indicated in $\mathbf{R}'$ - if $x = h(id' \parallel l')$. The other values in $\mathbf{R}'$ are ignored.

## 3.4 Realizing a TFSS

In a TFSS that leverages a trusted module $\mathbf{T}$ which offers the interfaces InsDel(), Update(), Send() and Deny() the server is responsible for providing the appropriate inputs to the module. More specifically, corresponding to *every* file (say, a file with index $id \parallel l$) the server stores

1. the file (possibly encrypted);
2. a record $\mathbf{R} = [id \parallel l \parallel q \parallel d \parallel a \parallel s]$;
3. all leaves of the access control IOMT for the file $id \parallel l$ with root $\alpha_{id,l}$;
4. an IOMT leaf of the main tree of the form $(x,\theta_x,x')$ where $x = h(id \parallel l)$ and $\theta_x = h(h(\mathbf{R}) \parallel \alpha)$

In addition, the server stores all internal nodes of the main IOMT.

#### 3.4.1 File Creation

When a user $id$ desires to create a new file $id \parallel l$ the user sends values $id \parallel l$, $v = \alpha$ the root of the access control IOMT for the file (along with all leaves of the access control IOMT), and an authentication token $\mu$ computed as $\mu = h(id \parallel l \parallel v \parallel \tau = INR \parallel K_{id})$. The server uses the interface InsDel() to insert a leaf for the index $x = h(id \parallel l)$, and then uses the function Update() to initialize the $\theta_x$ value in the newly inserted leaf as $\theta_x = h(h(\mathbf{R}) \parallel v)$ where $\mathbf{R} = [id \parallel l \parallel 0 \parallel 0 \parallel 0 \parallel 0]$ and value $\alpha = v$.

If the initialization of the leaf for the file is successful, the server returns a MAC ($\mu$) as an acknowledgement to the user where $\mu' = h(\theta_x \parallel \tau \parallel h(K_{id_r} \parallel ACK))$. This acknowledgement proves to the user that a leaf in the IOMT corresponding to the file submitted to the server has been added and initialized.

#### 3.4.2 Updating a File

Whenever an user $id_r$ modifies a file and saves it on the server, the server updates the IOMT using Update() and has to prove to the user that the updates to the file are successfully reflected.

The user sends values an encrypted file along with values $id_r$ (user identity), $id' \parallel l'$ (file to be updated), $q'$ (new sequence number), and $v$ (hash of new record $h(R')$, where $\mathbf{R}' = [id' \parallel l' \parallel q' \parallel d' \parallel a' \parallel s']$. The server verifies that $d'$ is the hash of the encrypted blob/file and extracts from its storage i) the current record for $id' \parallel l'$ ii) leaf for $id' \parallel l'$; iii) instructions for the leaf; iv) the access control value $\alpha$; v) a leaf of the access control IOMT which proves that $id_r$ does (or does not) have write access.

The user is returned an acknowledgement MAC $\mu = h(v \parallel \tau = WRQ \parallel h(K_{id_r} \parallel ACK))$ if the update was successful. If the condition $q' > q$ for sequence number is not met the ACK indicates an illegal request by setting $\tau' = ILU$. If the user does not have access rights to the file, the returned MAC is computed over $\tau' = NOA$.

#### 3.4.3 Reading a File

To read a file from the server, the requesting user ($id_r$) makes a query to the server for filename ($id \parallel l$), indicating a nonce $v$. The authentication token is computed as $\mu = h(id \parallel l \parallel v \parallel \tau = RRQ \parallel K_{id})$. If the queried file does not exist the server uses the Deny() to create an authenticated denial.

If the queried file exists, the server fetches the record for $id \parallel l$ along with the value $\alpha$, IOMT leaf, instructions, a leaf of the access control IOMT, and instructions to map the leaf to $\alpha$. If successful, the output of the module is a MAC for type $\tau' = RRQ$, authenticating the hash of the current record for $id \parallel l$, along with an encrypted secret $s'$.

The server returns the encrypted blob along with the a record $\mathbf{R}$ for the file, a MAC, and the encrypted secret, to the user. The user first verifies the integrity of the blob by verifying that the hash of the blob is consistent with the a value in the record $\mathbf{R}$, and that the record $\mathbf{R}$ is consistent with the MAC. After this the user can decrypt the secret and use the secret to decrypt the encrypted blob.

#### 3.4.4 Modifying Access Control and Deletion

An owner ($id == id_r$) of a file ($l$) can change its access privileges for different users by submitting a query to the server. The leaves of the access IOMT and its root $v$ are given to the server along with the query. The authentication token generated $\mu = h(id \parallel l \parallel v \parallel \tau = CAC \parallel K_{id})$, is used to update the value of $\theta_x = h(\mathbf{R}) \parallel \mathbf{v})$ after verification of ownership. The user is issued an acknowledgement MAC $\mu = h(v \parallel \tau = CAC \parallel h(K_{id_r} \parallel ACK))$ by the server to demonstrate that the update has been carried out.

To delete a file, the requesting user ($id_r$) must be the owner ($id$) of the file. The user ($id_r = id$) submits a query to delete a file ($id \parallel l$) to the server, which in turn uses the interface Update() to clear out values from the leaf of the IOMT. The authentication token $\mu$ computed as $\mu = h(id \parallel l \parallel v \parallel \tau = CLR \parallel K_{id})$. The function verifies $id_r == id$ and sets $\theta_x = 0$. An IOMT leaf for $x$ with $\theta_x = 0$ can be deleted by calling the InsDel() function.

## 4 Related Work and Conclusions

There have been many approaches towards improving trustworthiness of servers by leveraging a trusted module such as a TPM (tcg, 2007). Some solutions like the TCG-TPM approach rely on having a certain amount of trust in other components of the system such as Operating System, BIOS, CPU and possibly other peripherals which have direct access to the RAM. The AEGIS (Arbaugh et al., 1997) system was the precursor to the TCG-TPM approach of building a chain of trust to prove the integrity of the system from startup. The main pitfalls of the TCG approach include i) the time-of-check-time-of-use (TOCTOU) (Bratus et al., 2008) problem resulting from the fact that there are a variety of ways in which a code that has been measured and loaded, could be modified before it is actually executed; and ii) the impracticality of thorough verification of functionality of complex and dynamic software components. Strategies like NGSCB (Peinado et al., 2004) and Terra (Garfinkel et al., 2003) that partially address the second issue attempt to remove large chunks of code from the chain of trust by employing virtual machines such as Xen (Barham et al., ), Disco (Bugnion et al., 1997).

Merkle trees have found substantial attention as they provide a computationally efficient way to verify the integrity of dynamic data stored in untrusted locations. In the Terra application model (Garfinkel et al., 2003) the leaves of Merkle tree represent chunks of Virtual Disk representing an application, and the root verifies the integrity of the disk. Data entities, be it a file in a storage server (Sarmenta et al., 2006) or a record in a database (Maheshwari et al., 2000; Mykletun et al., 2004) can be represented as a part of a merkle tree to prevent replay attacks by an untrusted middle-man. Log based scheme along with merkle trees provide a efficient way of dealing sequential

atomic updates in Trusted Databases (Maheshwari et al., 2000). Refs. (Sarmenta et al., 2006; Dijk et al., 2006; Dijk et al., 2007) use the single monotonic counter of a TPM to generate virtual monotonic counters for each record or entity that needs to be verified in the system. Various data-structures such as log based scheme and Merkle trees are used to provide integrity verification of the data objects.

## 4.1 Conclusions

Servers acting as middle-men are central to the functioning of large class of network based applications. A current approach to secure a server is to run the server software on a TCG trusted platform equipped with a trusted platform module (TPM). The TCG-TPM exposes hundreds of fixed functions, which (amongst other things) facilitate storing and extending hashes of software in platform configuration registers (PCRs) inside the TPM, and binding secrets to specific PCR values. The goal of the TCG platform is to ensure that only certified code will be loaded and executed on the platform. Unfortunately, to achieve this goal the TCG model is forced to include untrustworthy components in the TCB.

Strategies that rely only on a trusted module - like the approach proposed in this paper and the virtual counter approach in (Sarmenta et al., 2006) require some additional functionality to become part of future TPM versions. Ref. (Sarmenta et al., 2006) suggests that adding the ability to maintain a merkle tree can address the needs of many application scenarios. We enumerated some shortcomings of the virtual counter approach and suggest that a index ordered merkle tree should be used instead of a merkle tree. The ability to verify non-existence is a crucial requirement if we do not desire to rely on the integrity of servers.

The TFSS system outlined in this paper is but one possible application of rich TCB functionality. From a high level the TCB functionality can securely a dynamic hash and/or a secret from the creator (or anyone authorized by the creator) to any authorized entity. For a web server the blob can be a static web page; for a DNS server the blob can be a DNS record; for an email server the blob can be an email.

Our ongoing work is focused on modifications and/or additions to the TCB functionality to make it better suited for securing a wide range of servers. Some possible additions include a) more sophisticated access control policies like i) different IOMT roots for read and write; ii) interpreting access control root as corresponding to a list of explicitly denied nodes (instead of explicitly allowed nodes), and b) efficient mechanisms for setting up shared secrets between users and servers.

# REFERENCES


(2007). Tcg specification: Architecture overview, specification revision 1.4.

Arbaugh, W. A., Farbert, D. J., and Smith, J. M. (1997). A secure and reliable bootstrap architecture. In *IN PROCEEDINGS OF THE 1997 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 65–71. IEEE Computer Society.

Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. (Mar. 2005). DNS Security Introduction and Requirements. *IETF RFC 4033*.

Barham, P., Dragovic, B., Fraser, K., Steven, H., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *In SOSP (2003*, pages 164–177.

Bratus, S., Sparks, E., and Smith, S. W. (2008). Toctou, traps, and trusted computing. In *In Trust 08: Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies*, pages 14–32.

Bugnion, E., Devine, S., and Rosenblum, M. (1997). Disco: Running commodity operating systems on scalable multiprocessors. In *ACM Transactions on Computer Systems*, pages 143–156.

Dijk, M. V., Sarmenta, L. F. G., Odonnell, C. W., and Devadas, S. (2006). Proof of freshness: How to efficiently use on online single secure clock to secure shared untrusted memory. Technical report.

Dijk, M. V., Sarmenta, L. F. G., Rhodes, J., and Devadas, S. (2007). Securing shared untrusted storage by using tpm 1.2 without requiring a trusted os. Technical report.

Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. (2003). Terra: a virtual machine-based platform for trusted computing. pages 193–206. ACM Press.

Lampson, B., Abadi, M., Burrows, M., and Wobber, E. (1992). Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310.

Maheshwari, U., Vingralek, R., and Shapiro, W. (2000). How to build a trusted database system on untrusted storage. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 10–10, Berkeley, CA, USA. USENIX Association.

Mykletun, E., Narasimha, M., and Tsudik, G. (2004). Authentication and integrity in outsourced databases.

Peinado, M., Chen, Y., Engl, P., and Manferdelli, J. (2004). Ngscb: A trusted open system. In *In Proceedings of 9th Australasian Conference on Information Security and Privacy ACISP*, pages 86–97. Springer.

Sarmenta, L. F. G., Dijk, M. V., Odonnell, C. W., Rhodes, J., and Devadas, S. (2006). Virtual monotonic counters and count-limited objects using a tpm without a trusted os. In *In Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing (STC06*, pages 27–42.

Thotakura, V. and Ramkumar, M. (2010). Minimal tcb for manet nodes. In *6th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2010)*. Niagara Falls, ON, Canada.

Weiler, S. and Ihren, J. (2006). Minimally Covering NSEC Records and DNSSEC On-line Signing. RFC 4470 (Proposed Standard).

# A Module T Interfaces

InsDel($\mathbf{L}_l, \mathbf{L}_r, x, \mathbf{I}_{lr}$) {
IF $(i == 0) \vee (j == 0)$ //inserting a leaf
  $x = insert(\mathbf{L}_l, \mathbf{L}_r, x, 0, \mathbf{I}_{lr}, r)$
ELSE IF $(((i == x) \wedge (\theta_i == 0)) \vee ((j == x) \wedge (\theta_j == 0)))$
  $x = delete(\mathbf{L}_l, \mathbf{L}_r, x, \mathbf{I}_{lr}, r)$;
IF $(x \neq ERROR) \quad r = x$;
RETURN $x$;
}

Deny($\mathbf{L}, \mathbf{I}, \mathbf{A}, id', l'$){
IF $(r \neq l2r(\mathbf{L}, \mathbf{I}))$ RETURN ERROR;
IF $(x \neq h(id \parallel l)$ RETURN ERROR;
$K_{ack} = h(K_{id_r} \parallel ACK)$;
IF $encl((x, x'), h(id' \parallel l'))$
  RETURN $\mu = h(v \parallel DNE \parallel K_{ack}))$;
ELSE
  RETURN ERROR;
}

Send($\mathbf{L}, \mathbf{I}, \mathbf{R}, \alpha, \mathbf{A}, \mathbf{L}', \mathbf{I}'$){
IF $((\tau \neq RRQ) \vee (r \neq l2r(\mathbf{L}, \mathbf{I})) \vee (x \neq h(id \parallel l))$
  RETURN ERROR;
IF $((h(h(\mathbf{R}) \parallel \alpha) \neq \theta_x) \vee (\mu \neq h(id \parallel l \parallel v \parallel \tau \parallel K_{id_r})))$
  RETURN ERROR;
IF $((\alpha \neq l2r(\mathbf{L}', \mathbf{I}')) \vee (id \neq id_r))$ RETURN ERROR;
$K_{ack} = h(K_{id_r} \parallel ACK)$;
IF $((id == id_r) \vee (z == id_r)) \quad rp = 1$;
ELSE IF $(encl(z, z', id_r)) \quad rp = 0$;
ELSE RETURN ERROR;
IF $(rp)$
  $K_e = h(K_{id_r} \parallel v); K_d = h(K_s \parallel id \parallel l \parallel q)$;
  $s = s \oplus K_d \oplus K_e$; $v' = h(\mathbf{R})$;
  RETURN $s, \mu = h(v \parallel \tau \parallel K_{ack}))$;
ELSE RETURN $\mu = h(v \parallel NOA \parallel K_{ack}))$;
}

Update($\mathbf{L}, \mathbf{I}, \mathbf{R}, \alpha, \mathbf{R}', \mathbf{A}, \mathbf{L}', \mathbf{I}'$){
IF $((r \neq l2r(\mathbf{L}, \mathbf{I}) \vee (\mu \neq h(id \parallel l \parallel v \parallel \tau \parallel K_{id_r})))$
  RETURN ERROR;
$K_{ack} = h(K_{id_r} \parallel ACK)$;
IF $((\tau == INR))$;
  IF $((id_r \neq id) \vee (\theta_x \neq 0) \vee (x \neq h(id' \parallel l')))$
    RETURN ERROR;
  $\mathbf{R} = [id' \parallel l' \parallel 0 \cdots \parallel 0]; v' = h(\mathbf{R})$;
  $v' = h(v' \parallel v)$; $\theta_x = v'$; $(r = l2r(\mathbf{L}, \mathbf{I}))$;
  RETURN $\mu' = h(v' \parallel \tau \parallel K_{ack})$;
IF $(h(h(\mathbf{R}) \parallel \alpha) \neq \theta_{\mathbf{x}})$ RETURN ERROR;
IF $((\tau == CLR) \vee (\tau == CAC))$
  IF $(id \neq id_r)$ RETURN $\mu' = h(v \parallel NOA \parallel K_{ack})$;
  IF $(\tau == CLR)$
    $\theta_x = 0$; $(r = l2r(\mathbf{L}, \mathbf{I}))$;
    RETURN $\mu' = h(v \parallel \tau \parallel K_{ack})$;
  ELSE IF $(\tau == CAC)$
    $\theta_x = h(\mathbf{R} \parallel v)$; $(r = l2r(\mathbf{L}, \mathbf{I}))$;
    RETURN $\mu' = h(v \parallel \tau \parallel K_{ack})$;
IF $((id \neq id') \vee (l \neq l'))$ RETURN ERROR;
IF $(v \neq h(\mathbf{R}'))$ RETURN ERROR;
IF $((\alpha \neq l2r(\mathbf{L}', \mathbf{I}')) \vee (id \neq id_r))$ RETURN ERROR;
IF $(\tau == WRQ)$
  IF $(id == id_r) \quad wp = 1$;
  ELSE IF $((z == id_r) \wedge (b == 1)) \quad wp = 1$;
  ELSE IF $((z == id_r) \wedge (b == 0)) \quad wp = 0$;
  ELSE IF $(encl(z, z', id_r)) \quad wp = 0$;
  ELSE RETURN ERROR;
IF $(wp)$
  IF $(q' > q)$
    $K_e = h(K_s \parallel id \parallel l \parallel q'); K_d = h(K_{id_r} \parallel id \parallel l \parallel q')$;
    $h = h'$; $q = q'$; $s = s \oplus K_d \oplus K_e$;
    $\theta_x = h(h(\mathbf{R}) \parallel \alpha); r = l2r(\mathbf{L}, \mathbf{I})$;
    RETURN $\mu = h(v \parallel \tau \parallel K_{ack}))$;
  ELSE RETURN $\mu = h(v \parallel ILR \parallel K_{ack}))$;
ELSE RETURN $\mu = h(v \parallel NOA \parallel K_{ack}))$;
}