An Efficient TCB for a Generic Data Dissemination System

A. Velagapalli, S.D. Mohanty, M. Ramkumar
Department of Computer Science and Engineering
Mississippi State University, MS.

Abstract—**Several applications fall under the broad umbrella of data dissemination systems (DDS), where providers and consumers of information rely on untrusted, or even unknown middle-men to disseminate and acquire data. This paper proposes a security architecture for a generic DDS by identifying a minimal trusted computing base (TCB) for middle-men and leveraging the TCB to provide useful assurances regarding the operation of the DDS. A precise characterization of the TCB is provided as a set of simple functions that can be executed even inside a severely resource limited trustworthy boundary. A core feature of the proposed approach is the ability of even resource limited modules to maintain an index ordered merkle tree (IOMT).**

Keywords-**Merkle trees, Trusted Computing Base, Index Ordered Merkle Tree.**

I. Introduction

The information age is characterized by the unprecedented ability of entities to acquire and disseminate different types of information over a wide range of channels, using a variety of hand-held, lap-top, desk-top and rack-based computers. A common characteristic of any data dissemination system (DDS) is that the providers and consumers of data may not be able to (or desire to) interact directly with each other, and consequently, rely on middle-men.

Most often the middle-men are always-on-line servers that store data from providers for access by consumers/clients. More generally, from the perspective of the users (providers and consumers) of the DDS, the middle-men also include a plethora of other computers and components[1] in the wide area network infrastructure that enable clients to communicate with servers. Irrespective of the specific nature and purpose of the DDS, the users of the system are required to trust the middle-men to not i) modify the data, ii) replay old data, or iii) deny the presence of the data that actually exists.

A mechanism for securing the end-to-end link between the client and a server platform can eliminate the need to trust numerous other infrastructural components in the path between the client and the server. However, users are still required to trust the server. Trust in a server implies confidence in the integrity of the server software, the platform on which the server software is executed, personnel who may have control over the platform, and that secrets employed by the platform (for authenticating served data to consumers and acknowledging receipt of data to providers) are well protected.

[1]Components necessary for the functioning of a wide area network like the Internet include, for example, switches, bridges, intra and inter-domain routers, DHCP servers, DNS servers, public key infrastructure, etc.

In practice, it is infeasible to rule out hidden malicious/accidental functionality in any complex component. As servers are typically composed of numerous complex hardware and software components, there is very little rationale in practice to simply trust middle-men. The contribution of this paper is a security architecture for a generic DDS which eliminates the need to trust middle-men.

A. Trusted Computing Base for a DDS

For any system with a desired set of assurances $\mathcal{A}$, the trusted computing base (TCB) [1] is a small amount of hardware and/or software that need to be trusted in order to realize the desired assurances $\mathcal{A}$. Specifically, as long as elements in the TCB are trusted, the desired assurances will be met even if all other components in the system misbehave. In general, the lower the complexity of the elements in the TCB, the lower is the ability to hide malicious/accidental functionality in the TCB components. Consequently, in the design of any security solution it is necessary to lower the complexity of components in the TCB to the extent feasible.

This paper proposes a comprehensive security solution for a generic DDS consisting of a dynamic set of providers disseminating dynamic data through an untrusted middle-man, to a dynamic set of consumers. The paper identifies a simple TCB for a generic DDS that can be leveraged to realize all desired assurances regarding the operation of the DDS. Specifically, the TCB is a set of simple functions $F_1() \cdots F_n()$ executed inside the boundary of a trusted module $\mathbf{T}$. Our desire to simplify the TCB translates to a desire to limit the computational and storage burden required for module $\mathbf{T}$ to execute functions $F_1() \cdots F_n()$.

Central to the proposed solution is the capability of even severely resource limited modules to maintain an index ordered merkle tree (IOMT). An IOMT is a simple extension of the well known merkle hash tree [2] to provide the ability to verify non-existence. The specific contribution of this paper is a precise characterization of the TCB functionality as four functions: i) $F_{uk}()$, used to issue symmetric secrets to the users of the DDS (viz., providers and consumers of data); ii) $F_{idl}()$, used to insert or delete leaves of an IOMT; iii) $F_{upd}()$, used to update a record from a provider; and iv) $F_{qry}()$ to respond to a query by any user.

B. Organization

The rest of this paper is organized as follows. In Section II we discuss the model for a generic DDS and enumerate the desired assurances $\mathcal{A}$. In Section II-A we discuss some of the current efforts to provide some assurances regarding the operation of an untrusted middle-man, which rely on the use of authenticated data structures (ADS), and some of the limitations of such approaches. In Section II-B we provide a broad overview of the proposed approach where a trusted module $\mathbf{T}$ serves as the TCB for the DDS. In Section III we provide an algorithmic overview of the index ordered merkle tree. Section IV provides an algorithmic description of the TCB functions executed by the trusted module $\mathbf{T}$. Conclusions are offered in Section V.

II. A Generic Data Dissemination System

We model a data dissemination system as being composed of a dynamic set of users $\mathcal{U}$ and a look-up server L. Users could be providers or consumers, or both. Any user may provide a succinct record $\mathbf{R} = [o, l, v, \tau]$ regarding an object to a look-up server. The object is uniquely identified by the owner o and a label l assigned by the owner, and is associated with a value v and a duration of validity τ of the record.

The owner o of a record is permitted to remove or modify the record at any time. Specifically, the owner may modify the record even while the current record has not expired. Any record may be queried by any user by specifying the owner and label. The querier expects the response to contain the most recent version of the queried record. More specifically, the querier expects the same response as she would *if* she had directly queried the owner o.

If the object itself is succinct (for example, an email address, a public key, etc.), the object may also be stored by the look-up server, and may be returned along with the response (in this case the value v may be a cryptographic hash of the email address or public key). More generally, the value v provides some information regarding the object, and could take several forms like the location of the object (a URL U), the cryptographic hash of the object (which can permit the client to verify the integrity of the object after it is obtained from location U), information necessary to establish a private channel with a data server at location U, etc.

The look-up servers simply do not care about the specific nature of the object or the purpose of a specific query. One query may be for the location $v = U$ of a file $l = F$ provided by a provider $o = P$. To establish a secret with the server U another query may be made for a record provided by $o = U$. The object F fetched from U may include components authenticated by another entity X. To verify the integrity of such components the client may desire the public key of X. This data may be disseminated through the same DDS by another entity - for example, a certificate authority $o = C$. If the CA C desires to revoke the public key of entity X, the CA may simply instruct the look-up server to remove the record indexed by $o = C$ and $l = X$.

Some of the basic desired assurances regarding the operation of any DDS are as follows:

1) Records can be modified only by owners; specifically, modifications to records by any entity other than the owner will be detectable by consumers;

2) Servers should only respond with records corresponding to the most recent update, and should not be able to replay prematurely invalidated records;

3) Servers will not be able to hide the presence of records that exist;

4) Servers will only provide a record if explicitly queried; more specifically, servers should *not* need to reveal the existence of records that were not explicitly queried.

A. Related Work

Several researchers have addressed issues in reliably querying an untrusted server using *authenticated data structures* (ADS) [3] - [9]. In such scenarios, clients who query a server trust only the originator/provider of the data (and not the server). Specifically, even while the originator of the data is not online, from a security perspective, ADS based schemes strive to provide the same assurances possible in scenarios where the queriers directly query the originator.

Broadly, an ADS can be defined by a construction algorithm $f_c()$ and a verification algorithm $f_v()$. The provider A of a set of records $\mathcal{D}_A$ computes a static summary $d = f_c(\mathcal{D}_A)$. The records $\mathcal{D}_A$ are hosted by an untrusted repository/server. Along with a response R to a query by a client, the server is expected to send a *verification object* (VO) ν satisfying $d = f_v(\nu, R)$, and the signature of the provider for the summary d. The client is now convinced that the response R would be the same *if* the client had directly queried A.

In ADS based approaches the owner A of the set of records $\mathcal{D}_A$ constructs a *hash tree* like data-structure with the records as leaves and the succinct summary d as the root of the tree. From this perspective, the ADS construction algorithm $f_c()$ can be seen as the algorithm to insert a leaf into the tree, and the VO can be seen as a set of hashes required to verify the integrity of any leaf against the root d using the verification algorithm $f_v()$. Most commonly used hash tree data structures for ADS applications include skip-lists, red-black trees and B-trees, all of which provide the capability to *order* records in a set (based on some index).

The purpose of ordering records is to permit succinct responses to i) queries for non existing indexes, ii) maximum/minimum value queries and iii) range queries. For example, if a querier seeks a record for an index X that does not exist, a two adjacent records in the tree can be sent (along with VOs to verify the two records against the root d signed by the originator) - one for an index x and one for the next index y such that $x < X < y$. As the tree is constructed by the originator, to the extent the querier trusts the originator of the data, the querier is assured that the queried index does *not* exist. When queried for all records in a range X to Y the server provides all records that fall in the range (each accompanied by an independent VO), and in addition, to prove completeness (to assure the querier that the server has *not* omitted any existing record) the server provides two additional records - a record for index $x < X$ indicating X as the next record and a record indicating that $y > Y$ is the index that follows Y.

1) Limitations of Existing Approaches: Some limitations of the ADS based approach render it unsuitable for several practical services with any of the following characteristics:

a) *Multiple Independent Providers:* In scenarios with multiple independent providers a record for an index X may be provided by and entity A and the record with the next higher index Y may be provided by an independent entity B. Clearly, neither A nor B can construct the hash tree.

b) *Truly dynamic data:* In most ADS based applications it is assumed that whenever any record is modified, the new root is signed by the originator and issued to the server. In scenarios where future modifications are unforeseen, the originator needs to sign the roots with short enough validity durations to ensure that the old root cannot be replayed by the server. Thus, the originator needs to send fresh signatures for the current root periodically, even if no updates were performed. In scenarios where the originator desires to be involved *only* for purposes of providing updates, existing ADS schemes are unsuitable.

c) *Revealing unsolicited information:* In many application scenarios it is desirable that servers should not be required to provide unsolicited information to queriers. As an example, in the case of the domain name system (DNS) [10], [11], to prove that no record with the queried DNS name exists, a DNS server is required to provide two name names that cover the queried name, thus revealing unsolicited information. This is the cause of the well known "DNS walk" or "zone enumeration" issue associated DNSSEC [12],[13].

d) *Low bandwidth overhead is desired:* In some scenarios, the overhead for the verification object (usually a sequence of hashes) required by clients for verifying any record may be unacceptable.

e) *Entrusting secrets to servers:* In some application scenarios the data to be conveyed to the client may be a secret. While ADSs can ensure integrity of data stored at untrusted servers, they do not address issues related to privacy of the data. Thus, in scenarios where middle-men need to be entrusted with secrets, conventional ADS schemes cannot be used.

B. Salient Features of the Proposed Approach

All the above inadequacies can be overcome if ADSs are *constructed and verified* by a *trusted third party* (TTP). Specifically, as typical ADS construction and verification algorithms involve only simple sequences of cryptographic hashing and logical operations, the TTP can be a low complexity trustworthy module $\mathbf{T}$. As the intent of the TTP is to ensure that middle-man cannot violate rules, module $\mathbf{T}$ functionality for constructing/verifying ADSs is the TCB for a middle-man.

In the proposed security model, the look-up servers are untrusted. However, a look up server has access to a trusted module $\mathbf{T}$ which performs some trivial functions $f_1() \cdots f_n()$ that constitute the TCB for the system. While a look-up server may be required to maintain and serve a dynamic number (say n) of records where n could be millions or even billions, the module $\mathbf{T}$ is assumed to possess only modest computational ability and small constant $\mathbb{O}(1)$ storage capability.

Records submitted by providers are stored as leaves of an index ordered merkle tree (IOMT), uniquely indexed as a function of the owner o and a label l (more specifically, $h(o, l)$, where $h()$ is a standard cryptographic hash function like SHA-1). The module stores only the root of the IOMT (a single hash). By performing simple sequences of hash operations, the module can verify the integrity of any record against the root of the tree.

In the proposed approach, a query from any user (a consumer) specifies the owner o and label l. The querier expects a response that is cryptographically authenticated by module $\mathbf{T}$. Similarly, requests for updates by users (providers) are authenticated by providers for verification by the module. On submitting an update request the providers expected an authenticated acknowledgement from the module. On receipt of a response authenticated by the module, to the extent that the users trust the module, they are assured that all four assurances are met.

1) Index Ordered Merkle Tree: The main difference between the well known merkle tree and the IOMT is that in the latter (as the name suggests) the leaves are ordered by some index. Apart from their utility in providing provable responses to a wide variety of queries, as we shall see in the next section, ordering of leaves is also necessary to thwart some replay attacks.

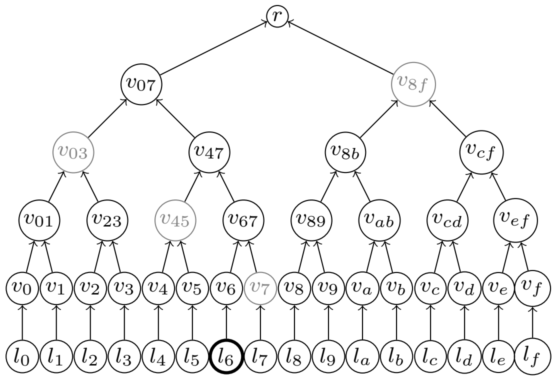
While merkle hash trees have received wide attention in the field of trustworthy computing, most ADS based do *not* employ the merkle hash tree due to some of the well recognized limitations of the binary merkle tree related to i) issues in inserting/deleting nodes and ii) inefficiencies in situations where the total number of leaves is not a power of 2. Consequently, most ADS based approaches have preferred the use of more complex tree structures. However, a simple extension to merkle trees - the IOMT - addresses such limitations of the plain merkle tree.

2) Query-Response Authentication: Modules have to verify authentication appended by providers for every update, and sign every response for verification by the querier. Obviously, for servers that may have to handle large volumes of queries and updates, the cost of authentication will be a significant bottle-neck - especially if asymmetric primitives are employed.

To amortize the overhead for authentication, asymmetric primitives are used only for setting up symmetric keys between the module and the users (providers and consumers). Message authentication codes (MAC) based on such symmetric keys are then used for authentication of exchanges between users and the module.

3) Opportunistic Shared Secrets Between Users: Often, a query is to locate some service/entity S with whom the querier Q expects to interacts soon after the query, and would thus desire a mechanism to secure the interaction. It is beneficial to use the trusted module to also opportunistically provide additional information required for the Q and S and to establish a secret K_{QS}. To cater for resource limited portable devices, it is desirable that mechanisms for establishing the shared secret be limited to symmetric primitives.

In the proposed approach, the shared secret established between the module and the users are also leveraged to



r
v_{07}
v_{8f}
v_{03}
v_{47}
v_{8b}
v_{cf}
v_{01}
v_{23}
v_{45}
v_{67}
v_{89}
v_{ab}
v_{cd}
v_{ef}
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_a v_b v_c v_d v_e v_f
l_0 l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8 l_9 l_a l_b l_c l_d l_e l_f

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

opportunistically establish shared secrets *between* users. Any user (say provider S) can submit a record to the look-up server to request the trusted module to serve as a mediator. In response to a query from some user Q for such a record from S, the querier will receive a non secret value p_{QS} that can be used to compute a symmetric secret K_{QS}. Both Q and S will need to perform only a single hash operation to compute the common secret K_{QS}.

4) Potential Applications: While almost any Internet/Intranet based service can be seen as falling under the category of a DDS, some specific examples are as follows:

1) Dissemination of dynamic DNS records by zone authorities. Response to negative queries will *not* reveal records that exist, and thus overcomes the DNS-walk issue that plagues the current DNSSEC [11] approach to secure DNS. In addition, unlike DNSSEC, the proposed approach can also be used to provide assurances for *dynamic* DNS (where DNS records may expire prematurely). Furthermore, server platforms may also convey records to facilitate any client (who performs a DNS look-up for the server) to opportunistically establish a shared secret with the server, which could be used as an IPSec security association.

2) Mobile servers with highly dynamic addresses could disseminate their reach-ability information (and enable clients to establish a secure channel with such servers);

3) Dissemination of dynamic revocation lists by certificate authorities, without the bandwidth overhead associated with certificate revocation lists (CRL).

4) Publication of dynamic quotes by any "exchange";

5) Dissemination of encryption secrets for Email messages; a user desiring to send an encrypted Email to some address will merely make a query to a look-up server to obtain a symmetric secret for encrypting the message.

III. Index Ordered Merkle Tree

A binary merkle hash tree is a data structure constructed using a cryptographic hash function $h()$ (for example, SHA-1). A tree of height L has $N = 2^L$ leaves. Figure 1 displays a merkle tree with $N = 16$ leaves $l_0 \cdots l_f$ (with height $L = 4$). The N leaf-nodes are obtained as $v_0 = h(l_0) \cdots v_f = h(l_f)$. Two adjacent nodes in each level (a left node v_l and a right node v_r) are hashed together to yield the parent node $h(v_l, v_r)$ one level above. The lone node at the top of the tree is the root r of the tree, and is a commitment to all leaves.

Corresponding to a leaf node v_i is a set of L complementary nodes $\mathbf{v}_i$ (consisting of one node in each level) such that $r = f(v_i, \mathbf{v}_i)$ where $f()$ represents a sequence of hash operations. 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 $\tilde{\mathbf{v}}_i \neq \mathbf{v}_i$ that will satisfy $f(v_i, \tilde{\mathbf{v}}_i) = r$.

In practical applications each leaf is a record of some large dynamic database. A trusted module capable of executing $f()$ stores only the root of the tree. The leaves and all other internal nodes can be stored by an untrusted entity $\mathbf{U}$. To modify a leaf l_i to l_i' the old leaf l_i, complementary nodes $\mathbf{v}_i$ and the new leaf l_i' (duly authenticated by the entity authorized to modify the leaf) are provided as inputs to the module. The module verifies that $r = f(h(l_i), \mathbf{v}_i)$ and computes the new root as $r' := f(h(l_i'), \mathbf{v}_i)$. Once the root has been modified to replace leaf l_i with l_i', the old leaf l_i cannot be demonstrated to be a part of the tree as it is infeasible to determine values $\mathbf{v}'$ such that the old leaf is consistent with the current root.

A. Merkle Tree Limitations

Two well known limitations of the merkle hash tree are i) the power of two requirements for the total number of leaves; and ii) the ability to readily recognize non existence of a record. The implication of the first limitation is inefficiency in scenarios where the total number of leaves N is not a power of 2. Specifically, if 513 records need to be stored, then a tree of length 1024 need to be maintained.

The second limitation is however more serious. For the module (which maintains only the root) to be convinced that no record regarding some index a exists, the module should verify the integrity of every leaf and in this process, deduce that no leaf for a exists. Obviously, this is far from practical. That the module can *not* verify non existence can be abused to perform replay attacks.

As a specific example, consider a scenario where some new information is available regarding a record for some index a (and thus needs to be updated). However, the untrusted server (which stores all records) can incorrectly claim that no information exists currently for a, and request the new information for a to be added as a new record. After this, as both the old and new records are part of the tree, the server has the ability to advertise either record.

B. Index Ordered Merkle Tree

The index ordered merkle tree is a simple modification to the merkle tree which addresses both limitations of the merkle tree. In an IOMT a leaf $\mathbf{L}_i$ (the i^{th}leaf in the IOMT) associated with data index a is of the form $\mathbf{L}_i = (a, v_a, a')$ where the middle value v_a is the data associated with index a, and a' is the *next* data-index.

That a leaf $\mathbf{L}_i = (a, v_a, a')$ can be verified by the module to be consistent with the root implies that i) a leaf exists for index a, and ii) no leaf exists for any index that falls between a and 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 (address) which wraps around and points to the least index.

A value x is *covered* by (a, a') if $(a < x < a')$, or (for the wrapped around pair) if $(x < a' \leq a)$ or $(a' \leq a < x)$. If $a = a'$ *all* values except a are covered, and implies that a is the *only* index.

The issues associated with the power of 2 requirement is addressed by simple modifications to hash functions $H_L()$ used for deriving a leaf node from an IOMT leaf and $H_V()$ used for combining two nodes to obtain a parent node. In an IOMT empty leaf is of the form $\mathbf{L}_i = (0, 0, 0)$. The function $v_i = H_L(\mathbf{L}_i)$ which maps a leaf to a leaf node is defined as

$$v_i = H_L(i, v_i, i) = \begin{cases} h(i, v_i, i') & \text{if } i \neq 0 \\ 0 & \text{if } i = 0 \end{cases} \quad (1)$$

The function $H_V(u, v)$ which maps two internal nodes to a common parent is defined as

$$p = H_V(u, v) = \begin{cases} u & \text{if } v = 0 \\ v & \text{if } u = 0 \\ h(u, v) & \text{if } u \neq 0, v \neq 0 \end{cases} \quad (2)$$

Consequently, an IOMT with a root 0 can be seen as a tree with *any* number of zero leaves of the form $(0, 0, 0)$.

By performing some simple checks the IOMT ensures that only one leaf can exist for an index. Specifically, a leaf with an index c can be inserted only if can be demonstrated that no leaf with index c exists currently by providing a leaf that covers c. Specifically, to insert a leaf for an index c two leaves need to be provided: i) an empty leaf $(0, 0, 0)$ and ii) a leaf for some other index a - say (a, v_a, a') such that (a, a') covers c. After insertion the two leaves will be modified to (c, v_c, a') and (a, v_a, c) respectively. To insert a leaf with index a when the root is zero, the root is simply set to $H_L(a, v_a, a)$. Similarly, when a leaf (x, v_x, x') needs to be deleted a leaf (b, v_b, b') should be provided such that $x' = b$. After deletion the first leaf becomes $(0, 0, 0)$ and the second becomes (b, v_b, x'). To delete a sole leaf (a, v_a, a) the current root $H_L(a, v_a, a)$ is set to 0.

Except for the case of insertion of the first leaf or deletion of a sole leaf, to insert or delete a leaf two leaves will need to be modified simultaneously. Two leaf hashes v_l and v_r can be simultaneously mapped to the root r by mapping the leaf hashes to the common parent, and then mapping the common parent to the root. Let v_p be lowest common parent of two leaf nodes v_l and v_r. More specifically, let $v_p = H_V(v_p^l, v_p^r)$ where v_p^l and v_p^r are the left and right child of v_p. Let $\mathbf{v}_l$, $\mathbf{v}_r$ and $\mathbf{v_p}$ be a set of hashes satisfying $v_p^l = f(v_l, \mathbf{v}_l)$, $v_p^r = f(v_r, \mathbf{v}_r)$, and $r = f(v_p, \mathbf{v}_p)$. Thus,

$$r = f(H_V(f(v_l, \mathbf{v}_l), f(v_r, \mathbf{v}_r)), \mathbf{v}_p) \quad (3)$$

IV. A Simple TCB for a DDS

In the proposed security architecture look up servers (LU) are associated with a module $\mathbf{T}$ with modest capabilities. Specifically, module $\mathbf{T}$ has the following limited abilities:

1) Generate an asymmetric key pair (R, U); the public key U doubles as the identity of the module;

2) Generate a random secret S; this secret will be used for generating symmetric secrets conveyed to users of the system (for authentication/verification of queries/responses);

3) Perform asymmetric encryption and decryption $X' = f_{enc}(U_o, X)$ (encryption of a value X using a public key U_o) and decryption $P = f_{dec}(C)$ using private secret R; such operations will be used infrequently - for conveying (secrets derived from S) to users.

4) perform simple sequences of logical and hash operations.

5) possess constant small memory for storing secrets S,R, IOMT root ξ, and temporary storage of inputs, outputs, and intermediate values.

6) possess a clock, which is however *not* synchronized with any external clock; the frequency of the clock is assumed to be fixed, and known to all users of the system.

Trust in the module implies that the secrets R and S are known only to the module with identity U (the public key of the module) and that one or more trustworthy entities/organizations have certified that the immutable functionality of the module has been verified (by certifying the public key).

A. TCB Functions

The module $\mathbf{T}$ exposes four functions $F_{uk}()$, $F_{idl}()$, $F_{upd}()$, and F_{qry} to the look-up server housing the module. In general inputs to the functions include values sent by a user to the look up server, and values stored by the server that are demonstrably consistent with the IOMT root stored inside the module. The outputs of the module include values like current time (according to the module), a message authentication code (MAC) for verification by a user encrypted secrets that can be decrypted by the user.

1) Conveying User Secret: Interface $F_{uk}()$ is employed to securely convey a secret K_o to a user with public key U_o (who is assigned an identity $o = h(U_o)$). The secret K_o may be used by the user to send authenticated update requests regarding objects owned by the user or send authenticated queries for objects provided by any user.

User :	Generate Key pair (R_o, U_o)
User :	Choose random challenge c, Compute $C = F_{enc}(U, c)$
User $\rightarrow$ LU :	$U_o, C, (\tau)$,
LU $\rightarrow$ $\mathbf{T}$:	$F_{uk}(U_o, C, \tau)$
$\mathbf{T}$:	$o := h(U_o); t_{ek} = t + \tau; K_o = h(S, o, t_{ek});$
$\mathbf{T}$:	$K_o^c := f_{dec}(C) \oplus K_o;$
$\mathbf{T}$ $\rightarrow$ LU :	$t, C' = f_{enc}(U_o, K_o^c), \mu = h(t, C', \tau, K_o)$
LU $\rightarrow$ User :	$t, C', \mu, (\tau)$
User :	$K_o = f_{dec}(R_o, C') \oplus c;$ Verify $\mu = h(t, C', \tau, K_o)$
User :	$t_{ek} = t + \tau$

Any user can generate a key pair (R_o, U_o) (using the specific asymmetric scheme supported by the module) and send a challenge to the module $\mathbf{T}$, encrypted using the module's public key, along with the user's public key U_o. The user or the server can specify the validity duration τ for the MAC key K_o issued to the user. The key K_o can be used by the user for computing MACs to authenticate update requests or queries sent by user till time $t_{ek} = t + \tau$ (time according to the module).

2) Inserting and Deleting IOMT leaves: The module maintains the root of an IOMT with any number of leaves of the form (a, v_a, a'). It is the responsibility of the server to store the leaves and intermediate hashes. The module considers a leaf as valid only if provided a set of hashes $\mathbf{v}_a$ satisfying $f(l_a, \mathbf{v}_a) = \xi$ where the leaf hash is computed as $l_a = H_L(a, v_a, a')$.

If $v_a \neq 0$, the leaf is interpreted as a record provided by owner o with label l such that $a = h(o, l)$, and $v_a = h(v, t_e, t_o)$, where v is a value associated with the record, t_e is the expiry time of the record, and $t_o \in \{0, t_{ek}\}$ (t_o can be zero, or the expiry time t_{ek} of the key K_o of the owner o). If $t_o \neq 0$ the module interprets this as a request to enable a private channel between any querier of the record and the owner o.

If $v_a = 0$ (if the middle value in the leaf is zero) the leaf is a "place-holder" and implies that no information is available regarding index a. The server can request insertion of any place holder (if no leaf or place holder currently exists for the index to be inserted) or delete any place holder. TMM function $F_{idl}()$ verifies the simple conditions that need to be satisfied for a place-holder to be deleted, and can be used to delete or insert a place-holder can be described algorithmically as shown below.

```
LU → $\mathbf{T}$ : $(l, v_l, l'), (r, v_r, r'), i, \mathbf{v}_l, \mathbf{v}_r, \mathbf{v}_p$
$\mathbf{T}$ → LU  : $\xi = F_{idl}((l, v_l, l'), (r, v_r, r'), i, \mathbf{v}_l, \mathbf{v}_r, \mathbf{v}_p)\{$
IF $(l = 0) \wedge (r = 0)$ RETURN; //At least one leaf should be non zero
IF $(l = 0) \vee (r = 0)$//If one leaf is zero it is the only leaf
  $\xi_1 := H_L(i, 0, i); \xi_2 := 0;$ //$i$ is the sole index
ELSE
  IF $(l = i)$//$(l, v_l = 0, l') \rightarrow (0, 0, 0)$
    IF $(v_l \neq 0) \vee (r' \neq i)$ RETURN; //Prereqs not satisfied
    $l'_l := 0; l'_r = H_L(r, v_r, l');$
  ELSE IF $(r = i)$//$(r, v_r = 0, r') \rightarrow (0, 0, 0)$
    IF $(v_r \neq 0) \vee (l' \neq i)$ RETURN; //Prereqs not satisfied
    $l'_r = 0; l'_l := H_L(l, v_l, r');$
  ELSE RETURN;
  $l_l := H_L(l, v_l, l'); l_r := H_L(r, v_r, v');$
  $\xi_1 = f(H_V(f(l_l, \mathbf{v}_l), f(l_r, \mathbf{v}_r)), \mathbf{v}_p);$ //Root before deletion
  $\xi_2 = f(H_V(f(l'_l, \mathbf{v}_l), f(l'_r, \mathbf{v}_r)), \mathbf{v}_p);;$ //Root after deletion
IF $(\xi = \xi_1)$ $\xi := \xi_2;$// delete index $i$
IF $(\xi = \xi_2)$ $\xi := \xi_1;$//insert index $i$
RETURN $\xi$;
}
```

To delete a place holder the LU server provides two current leaves - a place-holder $(i, 0, i)$ corresponding to the index i to be deleted, and a leaf $(j, v_j, j' = i)$ which points to the place-holder to be deleted. An exception is for deletion of a sole leaf $(i, 0, i' = i))$ in the tree (in which case the root $\xi = H_L(i, 0, i)$ should be set to 0).

To compute the root from two leaves three sets of complementary hashes are provided to the module. The module computes the roots ξ_1 and ξ_2 - the roots before and after deletion respectively. If the current root ξ is either ξ_1 or ξ_2 it is reset to ξ_2 or ξ_1 respectively. Specifically, if the current root is ξ_1, and if the leaves provided satisfy the condition for deleting index i, by setting the root to ξ_2 a leaf with index i is deleted. On the other hand, for the same inputs, if the current root is ξ_2 then by setting the root to ξ_1 a leaf corresponding to index i is *inserted*.

3) Updating Records: Typically, to provide a record or update a record the owner needs to send the values corresponding to the new record authenticated using a MAC μ. An exception for updating a stored record is when the stored record has expired, in which case the server can request the module to convert the record to a place holder (which can then be deleted if required using $F_{idl}()$).

A request for update from user o for a record with label l, includes a value v', a period of validity τ, and a flag f, a nonce n, and a MAC computed over values o, l, v, τ, f, n and secret K_o. After completion of the update the user expects an acknowledgement authenticated by the module.

To enable the module to compute $K_o = h(S, o, t_{ek})$ the inputs include the time of expiry t_{ek} of the key K_o. If no leaf exists for index $a = h(o, l)$ a place holder is inserted by the server using $F_{idl}()$. If the current leaf is a place holder (as will be the case when the record is provided for the first time) this fact is indicated to the module by setting $t_e = 0$. In the updated record the value v is set as requested to v'. The expiry time t_e of the record is set as $t_e = t + \tau$. If the flag f is set in the request the value t_o is set to be the same as the time of expiry t_{ek} of the key K_o of the owner. If $f = 0$ the value t_o is set to 0 instead.

```
User → LU : $o, l, v', n, \tau, f, t_{ek}, \mu = h(v', l, \tau, f, n, K_o)$
LU : If no leaf with index $a = h(o, l)$ insert index $a$ using $F_{idl}()$
LU : If leaf for index $a$ is a place holder, set $t_e = 0$;
LU : If no request from user $\mu = 0; t > t_e$
LU → $\mathbf{T}$ : $(o, l, v, t_e, t_o, a', \mathbf{v}, \mu, t_{ek}, n, v', \tau, f)$
$\mathbf{T}$ → LU  : $F_{upd}(o, l, v, t_e, t_o, a', \mathbf{v}, \mu, t_{ek}, n, v', \tau, f)\{$
$a := h(o, l); v_a := (t_e = 0)\,?\,0 \;:\; h(v, t_e, t_o);$
IF $(\xi \neq f(H_l(a, v_a, a'), \mathbf{v}))$ RETURN;
IF $(\mu = 0) \wedge (t > t_e)$//Expired record
  RETURN $\xi := f(H_l(a, 0, a'), \mathbf{v});$
IF $(t > t_{ek})$ RETURN; //Expired user key
$K_o := h(S, o, t_{ek}); t'_e = t + \tau;$
IF $(\mu \neq h(v', l, \tau, f', n, K_o))$ RETURN;
$t'_o := (f = 1)?\, t_{ek} \;:\; 0;$
$v'_a := h(v', t'_e, t'_o); \xi := f(H_l(a, v'_a, a'), \mathbf{v});$
RETURN $\xi, t, \mu' := h(a, v', t'_e, f, n, K_o);$
}
LU → User (only on successful execution of $F_{upd}()$): $\mu', t$
User : $t'_e = t + \tau; a = h(o, l);$ Check $\mu = h(a, v', t'_e, f, n, K_o);$
```

Only if $F_{upd}()$ executes successfully will the server receive a MAC μ' that can be conveyed to the user.

4) Querying Records: A user q may send a query for an object by specifying the owner o and label l and a nonce. No information may exist regarding the queried index a due to one of the following reasons

1) no leaf with index a exists; or
2) the leaf with index a is a mere place holder; or
3) the record has expired;

As any such reason can be verified by the module, the module can send an acknowledgement to the effect that no data is available. On the other hand, if the queried index exists, the module prepares an authenticated response which conveys the value v, and the remaining duration of validity (which is $t_e - t$). In addition, if the value t_o is not zero the module includes an additional value p_{qo} in the response which will enable q to compute a shared secret with the owner o of the queried record.

Specifically, if $t_{ek} > t$ and $t_o > t$ (both are current) the module computes $K_q = h(S, q, t_{ek})$, $K_o = h(S, o, t_o)$, and

$$p_{qo} = h(K_q, o, t_o) \oplus h(K_o, q, t_{ek}). \tag{4}$$

Using secret K_q the user q can compute $K_{qo} = h(K_q, o, t_o) \oplus p_{qo} = h(K_o, q, t_{ek})$ which can be readily computed by o using its secret K_o. Thus, both q and o can compute a secret by performing a single hash. The query response process can be algorithmically described as follows:

```
User $q \rightarrow$ LU : $a_q, n, \mu = h(a_q, n, K_q)$;
LU : either $a = a_q$ or $a$ covers $a_q$
LU : If leaf for index $a$ is a place holder, set $t_e = 0$;
LU $\rightarrow \mathbf{T}$ : $(a, o, l, v, t_e, t_o, a', \mathbf{v}, q, \mu, t_{ek}, n)$
$\mathbf{T} \rightarrow$ LU : $F_{qry}(a, o, l, v, t_e, t_o, a', \mathbf{v}, q, \mu, t_{ek}, n)\{$
IF $(t > t_{ek})$RETURN ; //Expired user key
$K_q := h(S, o_q, t_{ek}); a_q := h(o, l); p_{qo} := 0;$
IF $(\mu \neq h(a_q, n, K_q)$ RETURN;
$v_a := (v = 0)\,?\,0\,:\,h(v, t_e, t_o);$
IF $(\xi \neq f(H_l(a, v_a, a'), \mathbf{v}))$ RETURN;
IF $(t_e < t)$ RETURN;
IF $(a = a_q) \wedge (v_a \neq 0) \wedge (t < t_e)$//Unexpired Queried Record
  IF $(t_o > t)$//Compute pairwise public value
    $K_o := h(S, o, t_o); p_{qo} = h(K_o, h(q, t_{ek})) \oplus h(K_q, h(o, t_o));$
  $\mu' = h(a, v, t_e - t, p_{qo}, t_o, n, K_q);$
ELSE IF $(a = a_q) \vee ((a < a_q < a') \vee (a_q < a' < a) \vee (a' < a < a_q))$
  $\mu' = h(a_q, 0, 0, 0, 0, n, c);$ //Expired Record or Record NA
ELSE RETURN; //incorrect proof of non existence by server
RETURN $\xi, t, \mu', p_{qo}$;
}
LU $\rightarrow$ User (only on successful execution of $F_{qry}()$): $t, \mu', p_{qo}, v, t_e, t_o$;
User : if $v = 0$ Verify $\mu' = h(a_q, 0, 0, 0, 0, n, K_q)$
User : if $v \neq 0$ Verify $\mu' = h(a_q, v, t_e - t, p_{qo}, t_o, K_q)$
User : if $t_o > 0$ compute $K_{qo} = h(K_q, h(o, t_o)) \oplus p_{qo}$
```

V. Discussions and Conclusions

A simple trusted module with fixed functionality defined by functions $F_{uk}()$, $F_{idl}()$, $F_{upd}()$ and $F_{qry}()$ can be utilized to assure the operation of any look-up server, and thereby secure a wide range of applications under the DDS model. Specifically, the server maintains all records and internal nodes of the IOMT, and is forced to ensure that the values stored by the server remains consistent at all times with the root stored inside the module. Any record that cannot be demonstrated to be consistent cannot be updated, or conveyed to users. Specifically, only if the updates are applied in a consistent manner can the server send an authenticated acknowledgement to the user requesting the update; only if a record is consistent with the root can the server send the record to the querier (along with a MAC generated by the module).

In the proposed approach asymmetric cryptographic primitives are used sparingly - only for establishing shared secrets between users and the module. Unlike conventional ADS systems where for verification of any record the client requires a verification object in the form of a set of $\log_2 n$ hashes (where n is the total number of records stored by the server), in the proposed approach the VO is provided by the middlemen to the module, and only a single MAC is sent to the user to attest the accompanying record.

No component of the server, the user in control of the server, or the numerous components necessary for the functioning of any wide area network, need to be relied upon to realize the desired assurances. As long as the cryptographic hash function $h()$ is pre-image resistant, and the functions executed by the module cannot be modified, and the secrets protected by module cannot be exposed, all desired assurances are guaranteed.

Our motivation to reduce the complexity of operation performed by the module $\mathbf{T}$ is to improve the trustworthiness of the module. The simpler the functionality of the module, the better is the ability to verify the integrity of such functionality. Furthermore, due to generic nature (fixed functionality irrespective of the type of the look-up service) such modules can be easily mass produced; the process for verifying and certifying fixed functionality modules can also be easily automated.

Acknowledgement

This research was partly funded by the Department of Homeland Security (DHS)-sponsored Southeast Region Research Initiative (SERRI) at the Department of Energy's Oak Ridge National Laboratory.

References

[1] B. Lampson, M. Abadi, M. Burrows, E. Wobber, "Authentication in Distributed Systems: Theory and Practice," ACM Transactions on Computer Systems, 1992.

[2] R.C. Merkle "Protocols for Public Key Cryptosystems," In Proceedings of the 1980 IEEE Symposium on Security and Privacy, 1980.

[3] P. Devanbu, M. Gertz, C. Martel, S. Stubblebine, "Authentic third-party data publication," In Fourteenth IFIP 11.3 Conference on Database Security, 2000.

[4] A. Buldas, P. Laud, and H. Lipmaa, "Accountable certificate management using undeniable attestations," In ACM Conference on Computer and Communications Security, pages 918. ACM Press, 2000.

[5] A. Anagnostopoulos, M. T. Goodrich, R. Tamassia, "Persistent authenticated dictionaries and their applications," In Proc. Information Security Conference (ISC 2001), volume 2200 of LNCS, pages 379393. Springer-Verlag, 2001.

[6] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, S. Stubblebine, "A general model for authentic data publication," VC Davis Department of Computer Science Technical Report, 2001.

[7] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, S. Stubblebine, "Flexible authentication of XML documents," In Proc. ACM Conference on Computer and Communications Security, 2001.

[8] M. T. Goodrich, R. Tamassia, A. Schwerin, "Implementation of an authenti- cated dictionary with skip lists and commutative hashing," In Proc. 2001 DARPA Information Survivability Conference and Exposition, volume 2, pages 6882, 2001.

[9] M. T. Goodrich, R. Tamassia, N. Triandopoulos, R. Cohen, "Authenticated data structures for graph and geometric searching," In Proc. RSA Conference Cryptographer's Track, pages 295313. Springer, LNCS 2612, 2003.

[10] RFC 1034, Domain Names - Concepts and Facilities.

[11] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose "RFC 4033: DNS Security: Introduction and Requirements," March 2005.

[12] S. Weiler, J. Ihren, "RFC 4470: Minimally Covering NSEC Records and DNSSEC On-line Signing," April 2006.

[13] B. Laurie, G. Sisson, R. Arends, Nominet, D. Blacka, "DNS Security (DNSSEC) Hashed Authenticated Denial of Existence," RFC 5155, March 2008.