8-17-2013

Ordered Merkle Tree a Versatile Data-Structure for Security Kernels

Somya Darsan Mohanty

Ordered Merkle Tree - a versatile data-structure for security kernels

By

Somya D. Mohanty

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2013

Ordered Merkle Tree - a versatile data-structure for security kernels

By

Somya D. Mohanty

Approved:

Mahalingam Ramkumar
Associate Professor
Computer Science and Engineering
(Major Professor)

David A. Dampier
Professor
Computer Science and Engineering
(Committee Member)

Edward B. Allen
Associate Professor & Graduate Coordinator
Computer Science and Engineering
(Committee Member)

Yoginder S. Dandass
Associate Professor
Computer Science and Engineering
(Committee Member)

Nan Niu
Assistant Professor
Computer Science and Engineering
(Committee Member)

Jerome A. Gilbert
Interim Dean of the Bagley College of Engineering

Name: Somya D. Mohanty

Date of Degree: August 17, 2013

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Mahalingam Ramkumar

Title of Study: Ordered Merkle Tree - a versatile data-structure for security kernels

Pages of Study: 168

Candidate for Degree of Doctor of Philosophy

Hidden undesired functionality is an unavoidable reality in any complex hardware or software component. Undesired functionality — deliberately introduced Trojan horses or accidentally introduced bugs — in any component of a system can be exploited by attackers to exert control over the system. This poses a serious security risk to systems — especially in the ever growing number of systems based on networks of computers.

The approach adopted in this dissertation to secure systems seeks immunity from hidden functionality. Specifically, if a minimal trusted computing base (TCB) for any system can be identified, and if we can eliminate hidden functionality in the TCB, all desired assurances regarding the operation of the system can be guaranteed. More specifically, the desired assurances are guaranteed even if undesired functionality may exist in every component of the system *outside* the TCB.

A broad goal of this dissertation is to characterize the TCB for various systems as *a set of functions executed by a trusted security kernel*. Some constraints are deliberately

imposed on the security kernel functionality to reduce the risk of hidden functionality inside the security kernel.

In the security model adopted in this dissertation, any system is seen as an interconnection of subsystems, where each subsystem is associated with a security kernel. The security kernel for a subsystem performs only the bare minimal tasks required to assure the integrity of the tasks performed by the subsystem.

Even while the security kernel functionality may be different for each system/subsystem, it is essential to identify reusable components of the functionality that are suitable for a wide range of systems. The contribution of the research is a versatile data-structure — Ordered Merkle Tree (OMT), which can act as the reusable component of various security kernels. The utility of OMT is illustrated by designing security kernels for subsystems participating in, 1) a remote file storage system, 2) a generic content distribution system, 3) generic look-up servers, 4) mobile ad-hoc networks and 5) the Internet's routing infrastructure based on the border gateway protocol (BGP).

DEDICATION

I dedicate this dissertation to my parents Nalini Prava and Narasingh Ballav Mohanty, and to my wife Prashanti.

ACKNOWLEDGEMENTS

I want to take this opportunity to acknowledge and thank the people who facilitated and encouraged me during my graduate student career. Most importantly, I thank Dr. Mahalingam Ramkumar, my major professor and advisor for showing me through example what it means to be a passionate researcher and an effective mentor. He has always supported me and has been a source of much encouragement throughout my time at MSU. I greatly cherish the professional and personal relationship we have developed over the years.

I and grateful for my committee members Dr. David Dampier, Dr. Yogi Dandass, Dr. Edward Allen and Dr. Nan Niu and thank them for their valuable insights and suggestions regarding my research. Specifically, I would like to thank Dr. Dandass and Dr. Dampier on their feedback from a security perspective of the research, Dr. Allen and Dr. Niu for introducing me to concepts of software engineering and re-use.

My time as a graduate student at the Department of Computer Science and Engineering here at MSU has been a great experience for me. I extend my gratitude to the faculty and staff of CSE for helping me along the way and keeping me motivated. I acknowledge the funding agencies and the department of CSE for providing teaching and research assistantships and making my research possible.

On a personal note, I would like to thank my father for pushing me towards getting a doctoral degree, it would never have been possible without his encouragement. I will also forever be in debt of my wife Dr. Prashanti Manda, for introducing me to MSU and supporting me on every step towards my goals.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE

$\mathbf{U}$ **:** Untrusted user system

$\mathbf{T}$ **:** Trusted Module

χ **:** Module Secret

OMT : Ordered Merkle Tree

IOMT : Index Ordered Merkle Tree

ROMT : Range Ordered Merkle Tree

$\boldsymbol{\xi}$ **:** Primary Root of OMT

$(\mathbf{i}, \mathbf{i}', \omega_\mathbf{i})$ **:** A leaf in OMT with unique identifier i, next leaf identifier i' and a value ω_i

$\mathbf{v_i}$ **:** Instruction set to map a leaf with index i to root of its OMT

$\mathbf{cov}(\mathbf{c}, (\mathbf{a}, \mathbf{a}'))$ **:** Function to check if c is enclosed/covered by indexes a and a'

$\mathbf{H_L}(\mathbf{i}, \mathbf{i}', \omega_\mathbf{i})$ **:** Function to compute hash over individual elements of a leaf in a tree

$\mathbf{u_i}$ **:** User identity

$\mathbf{e}$ **:** User key expiry time

$\mathbf{K_u}$ **:** Pair-wise secret between module and user u

$\boldsymbol{\nu}$ **:** Nonce send in with the request

$\boldsymbol{\mu}$ **/** $\mathbf{MAC}$ **:** Message Authentication Code in response to a operation

$\mathbf{h}(\mathbf{ARGS})$ **:** Cryptographic hash function(SHA-1 or MD-5) which computes a 160-bit value over the concatenation of $ARGS$

CHAPTER 1

INTRODUCTION

Any application/system can be seen as a network of subsystems, each with a specific role in the operation of the system, interacting with each other according to system-specific and/or role-specific rules. For an ever increasing range of systems, some or all subsystems take the form of a computer, or a network of computers (for example, a server with one or more back-end servers). Undesired functionality — either deliberately introduced Trojan horses, or accidental bugs — are an unavoidable reality in complex subsystems. Such undesired functionality may be exploited by attackers to gain control of a subsystem for a wide range of nefarious purposes.

It is far from practical to assure the integrity of *every* hardware/software component in *every* component of *every* subsystem. One possible approach to secure systems is to mandate that all important subsystems should be associated with an appropriate *security kernel* that vouches for the integrity of (system-specific and role-specific) tasks performed by the subsystem. Specifically, all components of the subsystem are assumed to be untrustworthy; only the security kernel is trusted.

The security kernel for a system/subsystem is also commonly referred to as the *trusted computing base* (TCB) for the system/subsystem. More formally, the TCB for any system is "a small amount of software and hardware that the security depends on, and that we distinguish from a much larger amount that can misbehave without affecting security" [32]. In practice, the TCB for a subsystem could take the form of a dedicated hardware security module, or a software module executed on a general purpose platform, with some special protections [62] to guarantee that the security kernel will run unmolested.

A security kernel for a subsystem can also be seen as a set of functions executed inside a read-proof and write-proof [60, 64, 25, 51, 19] module $\mathbf{T}$. That the module is read-proof implies that secrets protected by the module (for authentication of the module) cannot be exposed; that the module is write-proof implies that the designed security kernel functionality of the module cannot be modified. It is however essential that the security kernel functionality is *deliberately* constrained to be simple — to permit consummate verification of the functionality, and thereby, rule out the presence of undesired functionality *within* the security kernel.

Some of the components of the security kernel for a subsystem will necessarily be specific to the subsystem whose operation is assured by the module $\mathbf{T}$. For example, the security kernel functionality for a domain name system server will be different from that of the security kernel for an IP registry, or a Border Gateway Protocol (BGP) router, or a device taking part in a mobile ad hoc network. Nevertheless, to simplify testing of the security kernel functionality, it is advantageous to possess efficient *reusable* components of the security kernels, with potential to be useful in a wide range of subsystems. Specifically, some of the important requirements of such reusable components are broad utility, and low (storage and computational) overhead.

1.1 Goals and Outcomes

The contribution of this research is an authenticated data-structure (ADS), the Ordered Merkle Tree (OMT) as an efficient and highly reusable component, suitable for security kernels for a wide range of systems. The utility of OMT is demonstrated by designing security kernel functionality for multiple systems like a) systems for providing generic look-up and content-distribution services; b) a sophisticated remote file storage system; c) mobile ad hoc networks (MANET) adhering to distance vector based routing protocols; and d) inter-autonomous-system routing infrastructure of the Internet, based on the border gateway protocol (BGP).

The specific contributions of this dissertation are

1. two variants of the Ordered Merkle Tree (OMT): the Index Ordered Merkle Tree (IOMT) and the Range Ordered Merkle Tree (ROMT);
2. design of simple algorithms to maintain OMTs, which can be executed even inside severely resource limited boundaries (inside a storage and computationally constrained hardware module $\mathbf{T}$);
3. design of generic security kernel functionality for mutual authentication of modules associated with different subsystems; and
4. design of subsystem specific functionality for subsystems associated with each of the investigated systems.

1.2 Research Strategy

Central to the proposed approach is the notion that any system can be modeled as interactions between subsystems. Each subsystem maintains a *database of records* [1] essential to its operation in the system. As subsystems are not trusted, we cannot expect

[1]For example, in a file storage system a records can correspond to a file, and specify parameters like file name, author, file hash, file encryption secret, access control policy, etc; in the case of a mobile ad hoc network (MANET) a record can contain information (hop count, next hop, sequence number, etc.) necessary to reach a specific destination .

them to guarantee the integrity of such records and/or to transact such records (with other subsystems) strictly according to the system specific rules.

In the approach presented in this dissertation, a subsystem is considered to have a *consequential role* if violation of rules by the subsystem is seen as a breach of security. Every subsystem with a consequential role is associated with a trusted module $\mathbf{T}$ that executes the security kernel for the subsystem. The security kernel is the enforcer for the subsystem specific rules. Unless the untrusted subsystem can demonstrate it's integrity to the security kernel module $\mathbf{T}$, the subsystem cannot take part in the operation of the system.

The security kernel functions can be broadly classified into three categories:

1. generic operations to maintain OMTs; this is used to track the dynamic database of (subsystem-specific) records stored by the untrusted subsystem;
2. generic operations for mutual authentication of interactions between modules belonging to different subsystems; and
3. subsystem specific security-kernel functionality that enforces subsystem specific rules for *updating* records.

Applications/systems can also be broadly classified (based on communication hierarchy) as client-server or distributed models. In client-server systems, interactions can occur only between a client subsystem and the server subsystem. In a distributed application model, any subsystem may directly interact with any other subsystem in the network. To illustrate the utility of OMTs in security kernels we illustrate our approach to secure two systems in each category (four systems in total). Applications in the client-server category include a remote file storage system and a generic look-up server. The distributed applications include mobile ad hoc networks (MANET) and the inter-AS routing infrastructure for the Internet (BGP).

In most systems based on the client-server model, misbehavior of a particular client subsystem does not affect other clients. In such a scenario only the server is associated with a security kernel. Client subsystems are not. The first application investigated under this model was a remote file storage system, which led to a conference publication [42] outlining a minimal TCB for a remote file-storage system. The approach utilized an Index Ordered Merkle Tree (IOMT) (described in Section 3.2) to represent stored records and access control privileges for files uploaded by users on to the centralized remote storage server. An extended version of the approach (with support for tracking multiple versions for each file) was submitted as a journal article [40]. The minimal TCB for remote file storage system was generalized to a minimal TCB for a generic content distribution infrastructure in a conference article [43]. The TCB for the content distribution system (CDS) ensured the integrity, authenticity and access control rules for distributing content (created by publishers) to the consumers of content (subscribers).

For distributed applications security kernels corresponding to different subsystem roles will need to be identified. In MANETs, each mobile device is seen as a subsystem with the same role. For BGP, subsystems may correspond to different roles like Registries, autonomous system (AS) owners, and BGP speakers/routers.

Our work in [39], illustrated the utilization of OMTs in designing a security kernel/TCB for various BGP subsystems like Registries, AS owners and BGP speakers/routers. A complex system like BGP serves as a good illustration of the scope of the proposed approach and the versatility of OMTs. Several types of ROMTs and IOMTs were utilized by various BGP subsystems. A journal article outlining this approach has been submitted [39]. Unlike BGP where the system included different types of subsystems, in a MANET, every subsystem (a MANET device) has the same role. In [41] we outlined simple security kernel functionality for MANET devices that can be adapted easily to support various protocol-specific rules.

1.3 Overview

Chapter 2 discusses various types of application models and the current approaches to address security issues. The core features of the proposed approach are explained in Chapter 3. Chapter 4 and 5 outlines the approach for securing a remote file storage system and a more generalized version of content distribution respectively. Chapter 6 discusses the work done towards establishing a security kernel for BGP. The TCB functionality for securing MANET protocols is discussed in Chapter 7. Chapter 8 discusses future work in the area and provides conclusions of the research.

CHAPTER 2

LITERATURE REVIEW

In a client-server model, subsystems (clients) have the ability to either to create new content or access content created by other subsystems. The central authority acts as a hub for storage of such content and controls access based on the permissions set by the creator of the content. Thus, subsystems always interact via the central subsystem (for example, a file storage server). In a file storage server model, the server acts as the central authority where users upload their files and set access privileges for other users to be able to access their content. Thus the participants is this model are primarily of two types (based on the role they play), 1) Clients (user machine) and 2) Host (server).

In comparison, a distributed application model suggests subsystems to be peers who create and distribute content. There is no central authority to regulate the distribution. However, subsystems who distribute the content may not be the actual creators but just act as relay nodes. An example of such a model is the border gateway protocol (BGP) [53, 8], where subsystems (routers) distribute routing information to their neighbors. The neighbors then redistribute the newly acquired information (routes) to their neighbors, and so on, to propagate content (routing information) over the network. However, in this model the subsystems can possess multiple roles, like router, system administrator, AS owner, AS Registry and Registrars, IP Registry and Registrars, etc.

A similar example of a distributed model is an ad-hoc network [48]. However in comparison to BGP, where there are multiple roles a subsystem can take on, each subsystem in a Ad-Hoc network has the same role in the network. Each subsystem has the responsibility of content creation and distribution along with assuming the role of routers in a dynamic network. As ad-hoc networks are dynamic in nature (participants/subsystems can join or sign out of the network on the fly), they have to accommodate frequent changes in routing information and changes in network topology. The current routing information possessed by neighboring nodes needs to be relayed to each new participant. Every time a subsystem leaves the network, the routing tables have to be modified and relayed to other subsystems so that they have the updated topology data.

From a security perspective the system may fail to realize its full potential if any participant fails to adhere to rules. Current approaches that attempt to secure systems include the use of authenticated data structures (ADS) and the use of trusted platform modules (TPM) to realize trusted platforms.

2.1 Authenticated Data Structures

In strategies that employ authenticated data structures [14, 4, 36, 26, 27] the provider A of a set of records $\mathbf{D}$ employs a special data structure along with a cryptographic hash function $h()$ to obtain a summary $d = f_c(\mathbf{D})$ (where the construction algorithm $f_c()$ is a specific sequence of hash operations). The summary is typically signed by the provider A (for example, using a public key scheme). The signed summary and data $\mathbf{D}$ can now be hosted by an untrusted server.

Along with a response R (typically one or more records from $\mathbf{D}$) to a query, the untrusted server is expected to provide a set of values in the form of a verification object $\mathbf{v}$ such that $d = f_v(\mathbf{v}, R)$, where $f_v()$ is also an algorithm composed of a specific sequence of hash operations. The server is effectively security-transparent to the provider A and the client, as successful verification of the ADS implies that the server has answered the query in exactly the same manner as the provider would have (if the client had directly queried the provider).

Most commonly used hash tree data structures for ADS applications include skip-lists [26], red-black trees [4], and B-trees [14, 27], all of which (similar to the Index Ordered Merkle Tree) 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.

ADS based approaches get around the limitations of the TCG approach (described in the Section 2.3) as no hardware/software is trusted. Specifically, all guarantees provided by the ADS approach are under limited and more reasonable assumptions like the existence of secure hash functions, and a trusted infrastructure for certifying public keys of users.

Unfortunately, some limitations of the ADS based approach render it unsuitable for several practical services with any of the following characteristics:

a) In scenarios with *multiple independent providers* a record for an index X may be provided by an 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 ADS.

b) 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 with *dynamic data* where the originator desires to be involved *only* for purposes of providing updates, existing ADS schemes are unsuitable.

c) In many application scenarios it is desirable that servers should *not provide unsolicited information* to the querier. Revealing unsolicited informations is the cause of the well known “DNS walk” or “zone enumeration” issue associated DNSSEC [67],[33].

d) 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 service providers need to be *entrusted with secrets*, conventional ADS schemes cannot be used.

2.2 Merkle Tree

A binary Merkle hash tree [38] is constructed using a cryptographic (SHA-1, Tiger, etc) hash function h(). A tree of height L has $N = 2^L$ leaves and $2N1$ nodes at l levels. Figure 2.1 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.

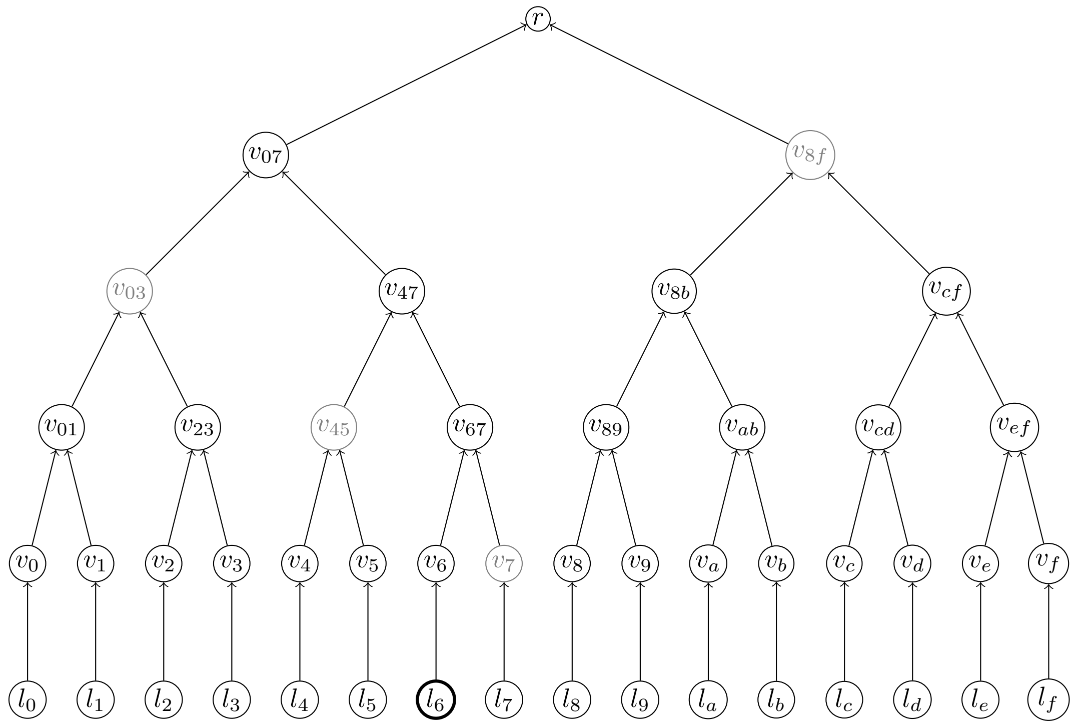

r
v_{07}
v_{8f}
v_{03}
v_{47}
v_{8b}
v_{cf}
v_{01}
v_{23}
v_{45}
v_{67}
v_{89}
v_{ab}
v_{cd}
v_{ef}
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_a v_b v_c v_d v_e v_f
l_0 l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8 l_9 l_a l_b l_c l_d l_e l_f

Figure 2.1

A Binary Merkle tree with 16 leaves. [1]

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

[1]Figure Notes: The complementary nodes for l_6 are v_7, v_{45}, v_{03} and v_{8f}.

$\{(v_7, 0), (v_{45}, 1), (v_{03}, 1), (v_{8f}, 0)\}$ 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 $v_{67} = h(v_6 \parallel v_7), v_{47} = h(v_{45} \parallel v_{67}), v_{07} = h(v_{03} \parallel v_{47})$, and $r = h(v_{07} \parallel v_{8f})$. Note that the bit specifies the ordering of two nodes before hashing them together to compute the parent node.

The Merkle tree also makes it possible to update each leaf independently. For example, if l_i is updated to l_i' the root can be updated to $r' = l2r(l_i', v_i)$ to reflect the new leaf l_i', where v_i represents a set of internal nodes. After the root has been updated, the old leaf l_i can no longer be demonstrated to be a part of the tree.

The primary advantage of Merkle trees is a single cumulative verification point for all its data records, which makes it easier to store that one value in a secure location. The leaves and other internal nodes can be stored in an untrusted location. A resource limited module (**T**) capable of i) storing the root, ii) performing simple sequences of hash functions to evaluate $l2r()$, and iii) verifying simple pre-conditions necessary for updating a leaf, can now provide integrity assurance to a large database of indexed records stored in an insecure location. More specifically, an untrusted device **U** may store such records, and request its **T** to verify the integrity of records and authenticate them to other **T**s. For the **T** to verify a leaf l_j containing a record corresponding to some index i, the untrusted device will have to provide a set of hashes v_j for the **T** to verify that $r = l2r(l_j, v_j)$ is indeed the root of the tree.

Even after this, it does not prevent a malicious entity to replay old records. The reason for this is that there is no explicit way to ensure that the device cannot associate the record for index a *with more than one leaf.* In such a scenario, the untrusted device can update one record for a and leave the other copy intact - which can then be replayed.

2.3 TPM Model

The Trusted Computing Group (TCG) [1] model using a Trusted Platform Module (TPM), was created to provide some assurances about the integrity of a machine's state of operation and its trustworthiness. In the TCG-TPM approach to realize a trusted platform, all software that can take control of the platform on which the machine runs is thoroughly verified and certified.

The specific goal of the TCG approach is to ensure that only pre-verified and authorized software can take control of the platform. The TCG-TPM approach assumes that some ”essential hardware” required for running software are trustworthy. This is required to employ the core root of trust measurement (CRTM), to build a chain of trust [5] and to prove the integrity of the system from startup. Starting from the time a computer is booted-up (from the time the CPU receives the first instruction stored at a fixed address), every piece of code is measured before control is passed to the code (typically, the measure of a code is the hash of the file). The first layer of code measures itself, measures the second layer of code, reports measurements to a trusted module (TPM chip), and passes control to the second layer. The second layer, loads and measures the third layer and provides the measurement to the TPM before control is passed to the third layer, and so on. Thus the root of trust realized is TPM $\rightarrow$ BIOS $\rightarrow$ Bootloader $\rightarrow$ OS $\rightarrow$ Application.

The TCG model relies on three roots of trust to achieve its security goals:

- **Root of trust for storage (RTS)**: This is provided by the hardware TPM module of the TCG-TPM approach. The TPM (v1.2) has 24 internal memory storage spaces called as Platform Configuration Registers (PCR). These PCRs have a secure storage space of 160 bits to represent measured values of software. Out of these 24 PCRs, 17 [0-16] are static registers whose values cannot be changed during runtime and the other [17-23] are dynamic registers which can be reset to zeros for storage of dynamic measurements. Every PCR register is initialized to NULL at the time of reboot of a machine. This secure storage provided by the PCR for keeping measurements is the core of RTS in the TCG model.

- **Root of trust for reporting (RTR)**: The TPM in the TCG model is assigned secrets (public-private keys) which are certified by a verification authority. Such secrets are unique to each TPM and is certified to the identity of the module. The module also contains non resettable counters. Both of these (secrets, counters) are used by the TPM to report measurements stored in the PCRs to verifying authorities. The authorities can then correctly associate the reported values to the identity of the TPM it came from. Thus, measurements along with certification of owning secrets and counters for a particular TPM identity provides the base for RTR.

- **Root of trust for measurement (RTM)**: The RTM is constituted by components *outside* the TPM. It relies on the entities who verify the measurements of software to deem them trustworthy. For this purpose, the TCG model employs the infrastructure (I_s) (of verifying authorities) to verify functionality of all software that can be loaded and executed by platforms, and disseminate validated measurements of verified software to entities that need to interact with the trusted platforms. This infrastructure I_s caters to providing correct measurements for valid code that an entity requests. This set of measurements is the RTM for TCG-TPM architecture.

Operation: An entity operating under the TCG-TPM model, uses these three roots of trust (RTS, RTR, RTM) to gain trust with other interacting entities. Each and every software module (data) is measured ($m = h(data)$) and stored to RTS using the PCRs on a TPM. These stored values represent the *state* the entity's machine is operating in. Whenever the entity wishes to report the state of its system, certified public keys and counters are used to generate RTR.

This RTR is given to other interacting entities, who then get the verified measurements of the same software module from the verification authorities infrastructure I_s. They then may choose to abandon the interaction if the reported measurements differ from expected values (which can happen if any code with uncertified measure was part of the chain). If the measurements match, the entities may be assured of the trustworthiness of the state of the machine they wish to interact with.

2.3.1 Issues with TCG-TPM model

Un-warranted trust in hardware components: The "essential hardware" components that are trusted in the TCG approach unfortunately include 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. Other than the TPM itself, none of the other components are verified and certified to be used on a trusted platform. Our inability to provide reasonable assurances to such components is the fundamental reason behind the well-known time-of-use-time-of-check (TOCTOU) problem [10] that plagues the TCG approach. More specifically, the TOCTOU problem is a result of 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. The attack describes ways to modify addresses or permission bits in Page Table Entries (PTE) to point to a attack frame for a trusted application in RAM of the machine. PTE's are used to store mapping between virtual memory addresses and its corresponding physical address on a machine. Consequently, the state reported by the TPM may not correspond to the actual state of the platform.

Verification Infrastructure (I_s) issues: The RTM component of the TCG-TPM approach depends on the verification of software and to generate measurements for such. The infrastructure I_s needs to formally prove every component of such software. For huge softwares such as OS and large applications it is nearly infeasible to able to prove its validity. Even the generic OS available on user machines in the current scenario have millions of lines of code. They themselves release security updates to plug security holes in their operation. Keeping such a measurement in the TCB can only result in unreliable trust in its operation. The I_s also needs to keep track of every application that is run on the system which can gain access to the hardware components (DMA, RAM) of the machine. Measuring each and every application along with its versions of updates is far from being practical. Then again is the question of what mandates the trust in I_s? Did the verification authorities correctly verify the security of software and according to what specification. Which authority verifies that? This leaves us with a large security question for the verification infrastructure I_s.

Failure of RTS: The root of trust for storage would fail in a case where the contents of the hardware protected PCRs is compromised by an malicious entity, i.e. a static PCR value can be changed dynamically by the attacker. In such a scenario the attacker would be able to present any state of the machine to be a valid state for the verifier. The measurements for the untrustworthy machine which the TPM has stored in PCRs would be replaced during verification for a trusted machine and the verifier would blindly trust the machine as its measurement reported is for a stable state. This in turn would compromise the security aspect of the TCG approach completely at its root.

Failure of RTR: A TPM reports measurements to a verifier using some internal secrets (master secret, counter). These secrets are stored persistently in the TPM protected memory and should not be revealed to any outside entity (even the machine it is attached to). If an attacker is able to break the protection of such secrets and reveal to itself, they can be used to report any value (measurement) certified with the TPM secrets. Even older measurements of a stable state can be replayed by spoofing the counter value of TPM and signing it with the compromised master secret. A verifier would have no way to detect such an attack unless the TPM reports that it has been tampered with.

Failure of RTM: The verification infrastructure I_v is responsible for measuring valid applications and reporting its corresponding values to other entities. In a case where a certifier of I_v is malicious i.e. the TPM has been compromised, any measurement can be certified to be a valid measurement. Other entities of the model can use such corrupt certification to prove themselves. A verifying entity trusts the TPM of the certifier and its invalid certification, thus has to extend the same trust to un-secure platforms of other entities.

2.3.2 Dynamic Root of Trust

Dynamic Root of Trust Measurement (DRTM) uses the dynamic resettable PCR [17-24] along with some particular functionality support by the newer generation CPU such as Intel Trusted eXecution Technology (TXT) [3] and AMD Secure Virtual Machine [SVM] [15]. These allow for a TPM to be able to *Late Launch* into measuring software module effectively removing the BIOS and legacy OS from the root of trust as with the SRTM.

At the core of DRTM is the mechanism of providing isolation for a piece of code to be able to execute using the TXT or the SVM features. These features enable security sensitive code to be protected from interference from legacy OS or interrupts from other hardware components of the machine. DRTM for such a system is the measurement of code that is executed in the environment thus, reducing the TCB of the machine to include just the secure code.

2.3.2.1 Realization of DRTM

To enter into a late launch environment in a system using AMD processors a special new instruction *SKINIT* [15] is called (*SENTER* [3] on Intel systems). The input to SKINIT is a physical address of Secure Loader Block (SLB) [15, 3]. When the CPU receives the instruction, it disables (changes DEV) all memory interrupts (DMA) and interrupts from previously executing code. Debugging mode is also disabled. After all this is done the isolation needed for the code is provided and the CPU jumps to the memory address provided by the SKINIT input.

During the execution of the SKINIT instruction the processor also resets dynamic PCR (17-24) to all zeros. The SLB is then measured ($m = h(SLB)$) and extended to PCR17. Only after this is done the SLB is loaded. The SLB contains an entry point address for the code to be executed and its length.

Examples using the DRTM method are:

OSLO: Open Secure LOader: OSLO [30] is a small (just over 1000 lines) secure kernel which provides an isolated environment for the applications to execute in. During initialization, OSLO resets the TPM PCR [17-24] and enables it for extend operation. It then stops all other processor cores that are present in a multi core CPU system. After this is done, SKINIT is executed for OLSO to take over control of the CPU using the SLB. The applications that are supposed to be executed in the environment are measured and loaded. These measurements can then be reported for verification along with the output the application generated. Thus, OSLO removes the BIOS and bootloader completely from the TCB for all applications.

Flicker: Flicker [37] uses the same SKINIT instruction to provide isolation but is used in a different way. Instead of loading a completely new kernel, flicker saves the state of the legacy OS before jumping into the Late Launch (as shown in Figure 2.2). Then it measures and executes a Piece of Application Logic (PAL) [37] in the isolated state. PAL are parts of the application which need to be executed in a secure environment to establish trust in the application itself. After this is done the machine is brought back to the previous saved state of the legacy OS and the measurements of the PAL code are provided in the PCR registers. The entry point address and the length of execution of PAL are initialized in the SLB core by the application developers.

The advantage of Flicker is the elimination of measurement of the OS itself along with BIOS and bootloader. The system also helps in restoration of normal OS operation after execution of PAL, hence the output and the measurements can be used by the applications designed for legacy OS for providing trustworthiness.

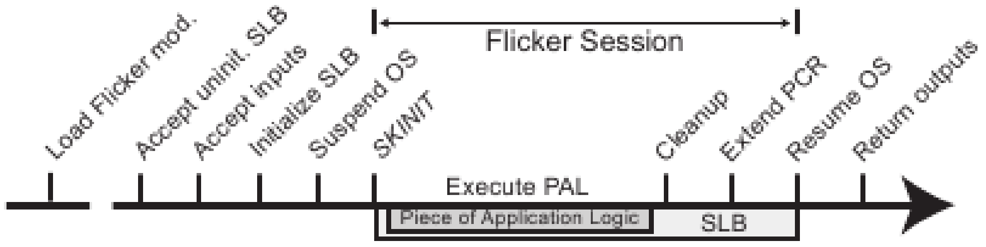


Figure 2.2

Flicker sequence of operation

2.3.2.2 SRTM (TCG) Vs DRTM comparison

In Static Root of Trust Measurement (SRTM) model the TCB comprises of TPM, CPU, Memory and its controllers, peripheral devices(DMA), BIOS, bootloader, OS, applications. In other words every software or hardware module that can take control over the system needs to be verified. The DRTM approach results in removal of large components such as BIOS, bootloader, OS and other applications (shown in Figure 2.3 [37]) as a result of the late launch environment.

Owing to large TCB in the SRTM approach the verification authorities need to keep track of measurements for every component in the SRTM TCB. Different versions of BIOS, OS and other applications along with their updates need to be measured in advance to be certified. Each application module loaded to the machine which can take control of the system also needs to be measured. Verification of large code bases such as for legacy OS and applications is highly complex and expensive. In the DRTM model, the verification authorities need to deal with a much smaller code base owing to the minimal TCB. The secure application (in question here for its trustworthiness) is the only reported value to the verification authorities. So they need to keep track of just the application base and its versions. The size of the reported measurements for verification is a lot less for DRTM in comparison to SRTM.

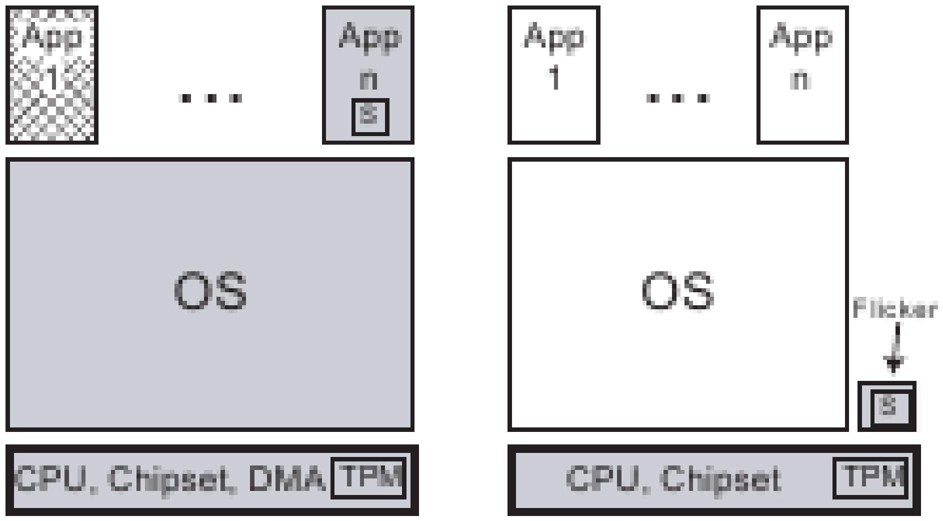


Figure 2.3

Comparision of trusted components (shaded) in SRTM and DRTM

However, the primary advantage of the DRTM approach (in comparison to the SRTM approach) is also its key drawback. More specifically, in order to *late launch* to a secure environment (for the secure application to execute), the device needs to stop every on going operation in its operating system and give up its resources/privileges to the late launch environment. Forcing the device operation into a standby state every time the secure application has to execute, is an impractical approach in the current computing environment.

2.4 Other Related Work

There have been many approaches [58, 56, 57, 2, 44, 34] towards improving trustworthiness of systems by leveraging a trusted module such as a TPM [1]. 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 [5] 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) [10] 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 [47] and Terra [24] 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 [7], Disco [11].

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 [24] 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 [59] or a record in a database [35, 45] 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 [35]. Refs. [59, 17, 18] 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 schemes and Merkle trees are used to provide integrity verification of the data objects.

CHAPTER 3

SALIENT FEATURES OF THE PROPOSED APPROACH

In the approach adopted in this dissertation, any system is seen as a set of subsystems collaborating (communicating) over a network to achieve the overall purpose/goal of the application/system. Each subsystem has a specific role in the operation of the system and interacts with other subsystems according to system-specific and/or role-specific rules. For example,

1) subsystems in the domain name system (DNS) have roles like *zone authorities*, who create DNS resource records (RR) pertaining to the zone; *authoritative name servers*, that are chosen by the zone authority to disseminate DNS RRs for the zone; and *local (or preferred) name servers*, that iteratively query authoritative name servers to resolve queries from clients.

2) subsystems in the inter-domain routing infrastructure for the Internet — the Border Gateway Protocol (BGP) — have different roles like *autonomous system (AS) owner*; *AS registry*, that assigns AS numbers to AS owners; *IP registry* that issues (through IP registrars and ISPs) chunks of IP addresses, or IP prefixes[1] to AS owners; and *BGP speakers* for an AS, authorized by the AS owner to originate routes for IP prefixes owned by AS.

[1]An IP prefix is a chunk of 2^n consecutive addresses, where $32 - n$ MSBs of all addresses in the chunk are the preserved.

3) subsystems in a remote file storage system include clients who create files and specify access control policies for the files, and servers that store files, and make them available for easy access by users.

4) subsystems in a on-line market place system include buyers, sellers, financial institutions, manufactures/wholesaler (who typically performs drop-shipping), etc.

Each subsystem is associated with a trusted security kernel which vouches for the tasks performed by the subsystem. In order to distinguish functionality in each, we represent an untrusted subsystem as $\mathbf{U}$ and the trusted security kernel module (associated with $\mathbf{U}$) as $\mathbf{T}$. Module $\mathbf{T}$ is the TCB for the subsystem.

In this model of systems, each subsystem is associated with one or more dynamic databases of subsystem-specific records. While the dynamic databases are stored by the untrusted subsystem $\mathbf{U}$, the integrity of the records are tracked by the trusted module $\mathbf{T}$. To enable a resource limited module to track dynamic data of any size, special authenticated data structures (ADS) are used to represent databases of records. Only the *root* of the ADS (a single cryptographic hash) needs to be stored inside the module $\mathbf{T}$ to track the entire database of records. The Ordered Merkle Tree (OMT) is the ADS used in this dissertation.

"Vouching" for a subsystem implies that only records recognized by $\mathbf{T}$ as consistent with the OMT root (stored inside $\mathbf{T}$) will be authenticated by $\mathbf{T}$. Only records authenticated by the trusted module of a subsystem will be entertained by trusted modules of *other* subsystems.

Security kernel functionality for tracking databases of records using OMTs, and for transacting authenticated records between modules associated with different subsystems can be broadly classified into three categories:

1. OMT functionality,
2. functionality for cryptographic authentication, and
3. subsystem-specific functionality.

In the rest of this chapter we discuss generic (system-independent) functionality. System specific functionality for various systems are discussed in chapters 4 to 7.

3.1 Cryptographic Authentication

While there are a wealth of options to choose from for cryptographic authentication one over-arching goal of the proposed approach is to reduce overhead — especially overhead for trusted modules, to the extent feasible. For this purpose we minimize the use of asymmetric cryptographic primitives in favor of light-weight symmetric primitives.

Ultimately, in our approach all exchanges between two entities are authenticated using time stamp message authentication codes (MAC). In general, a MAC μ for a value v exchanged between two entities is computed as

$$\mu = h(v \parallel t \parallel K) \tag{3.1}$$

where t is the current time, and K is a symmetric key shared between the sender and the receiver.

Two broad approaches for cryptographic authentication are used in this dissertation: one for client-server systems, and the second for distributed systems. In client-server systems one entity is a client, and the other is the security kernel module $\mathbf{T}$ associated with the server. The time t is the time according to the module $\mathbf{T}$.

In distributed systems both the entities are security kernel modules of different subsystems. In general, as every module will consider its clock as the current time, additional strategies are required for either synchronizing clocks between modules, or estimating the clock-offsets. If the integrity of the clocks cannot be assured when the modules are powered off, then the modules will require to maintain a non volatile counter which is incremented every time the module is powered on. For such scenarios the MAC is computed as

$$\mu = h(v, t, c, c', K) \tag{3.2}$$

where t is the time according to the sender of the message, c and c' are the respective session counters of the two entities, and K is the shared secret.

3.1.1 Shared Secret in Client Server Model

In the client-server model, module $\mathbf{T}$ spontaneously generates a private key R_T of some asymmetric cryptographic scheme. The corresponding public key U_T is assumed to be made known[2] to all participants of the system. Module $\mathbf{T}$ possesses a clock-tick counter. At any instant of time, the tick count t is interpreted by the module as the current time.

The system may possess any number of clients. Clients may join at any time. Every client generates a key pair. The identity of a client with public key U_i is simply $u_i = h(U_i)$. Every client can now establish a long-lived secret with the module $\mathbf{T}$. Specifically, the

[2]For example, the public key may be certified by one or more trusted entities responsible for verifying and certifying the integrity of the module $\mathbf{T}$.

long-lived secret shared between user u_i (with key-pair (R_i, U_i)) and the module $\mathbf{T}$ with key-pair (R_T, U_T) is

$$K = \mathcal{K}(R_i, U_T) = \mathcal{K}(R_T, U_i), \tag{3.3}$$

where the specifics of the function $\mathcal{K}()$ (which depends on the specific nature of the asymmetric scheme [16, 54, 20]) are not important for our purposes. Such shared secrets are computed infrequently, as they are used only to securely convey short-lived secrets (which are used for computing MACs).

3.1.2 Shared Secret in Distributed Model

In the distributed model we assume the existence of one or more trusted key distribution centers (KDC). Every module has a certified public key pair. This enables a KDC to convey a secret securely to each module. The secrets conveyed by KDCs enable modules to compute pairwise secrets. Let κ_x be the secret issued to module X.

While several strategies exist for establishing shared secrets using secrets issues by KDCs, in this dissertation we adopt the modified Leighton-Micali scheme [50]. Any two TMMs X and Y can using their respective KDC secrets κ_x and κ_y to compute a common secret

$$K_{xy} = h(\kappa_x, Y) \oplus \pi_{xy} = h(\kappa_y, X) \oplus \pi_{yx} \tag{3.4}$$

where π_{xy} and π_{yx} are pairwise *public* values made available to the untrusted subsystems corresponding to X and Y respectively. When the modified Leighton Micali scheme (MLS) [50] is used to compute the pairwise secret K_{xy} only one of the two subsystems requires access to a public value; the other simply employs the public value of 0 (if $\pi_{xy} \neq 0$, then $\pi_{yx} = 0$; if $\pi_{xy} = 0$ then $\pi_{yx} \neq 0$).

3.2 A Reusable Authenticated Data Structure

An ADS [14, 12, 4, 36, 26] is a strategy for obtaining a concise cryptographic commitment for a set of records. Often, the commitment is the root of a hash tree. Any record can be verified against the commitment by performing a small number of hash operations.

An Ordered Merkle Tree (OMT) is an ADS that is derived as an extension of the better known Merkle hash tree. Similar to a plain Merkle tree, an OMT permits a resource (computation and storage) limited module to track the records in a dynamic database of *any* size, maintained by untrusted components of the associated subsystem. Using an OMT (instead of a plain Merkle tree) permits the resource limited module to additionally infer a few other "useful holistic properties" regarding the database.

3.2.1 Ordered Merkle Tree

The Merkle hash tree [38] is a data structure constructed using repeated applications of a a pre-image resistant hash function $h()$ (for example, SHA-1). Figure 3.1 depicts a tree with $N = 16$ leaves. In practical Merkle tree applications each leaf can be seen as a record belonging to some database.

A tree with N leaves has a height of $L = \log_2 N$. At level 0 of the tree are N leaf-nodes, one corresponding to each leaf, typically derived by hashing the leaf. At the next level (level 1) are $N/2 = N/2^1$ nodes, each computed by hashing together a pair of "sibling" nodes in level 0. Level i has $N/2^i$ nodes computed by hashing a pair of siblings in level

$i - 1$, and so on, till we end up with a lone node ξ at level L — the root of the binary tree. A tree with $N = 2^L$ nodes has $2N - 1$ nodes distributed over $L + 1$ levels, where $L = \lceil \log_2 N \rceil$.

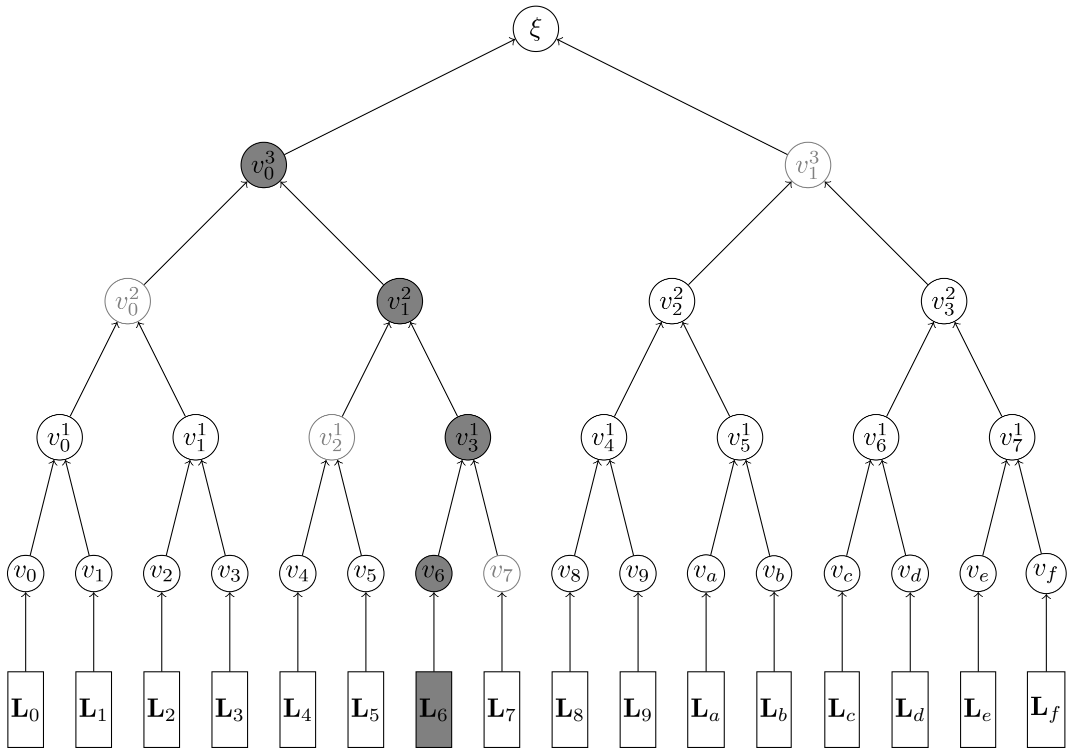

ξ
v_0^3
v_1^3
v_0^2
v_1^2
v_2^2
v_3^2
v_0^1
v_1^1
v_2^1
v_3^1
v_4^1
v_5^1
v_6^1
v_7^1
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_a v_b v_c v_d v_e v_f
$\mathbf{L}_0$ $\mathbf{L}_1$ $\mathbf{L}_2$ $\mathbf{L}_3$ $\mathbf{L}_4$ $\mathbf{L}_5$ $\mathbf{L}_6$ $\mathbf{L}_7$ $\mathbf{L}_8$ $\mathbf{L}_9$ $\mathbf{L}_a$ $\mathbf{L}_b$ $\mathbf{L}_c$ $\mathbf{L}_d$ $\mathbf{L}_e$ $\mathbf{L}_f$

Figure 3.1

A binary hash tree with 16 leaves. [3]

Two nodes node v_i^j and v_{i+i}^j at level j are siblings if i is even (else v_{i-1}^j and v_i^j are siblings). Two siblings — the left sibling u and the right sibling v are hashed together to obtain the parent node as $p = h(u, v)$. Any node in a binary tree can be seen as a root of a sub-tree, or a leaf-node of a subtree. Corresponding to a leaf node v in a sub-tree of

[3]Nodes v_6, v_3^1, v_1^2, v_0^3 (filled gray) and root ξ are ancestors of leaf $\mathbf{L}_6$. The set of "siblings of ancestors," viz., $\mathbf{v}_6 = \{v_7, v_2^1, v_0^2, v_1^3\}$ are "complementary" to v_6. The root is a commitment to all leaves. The complementary nodes $\mathbf{v}_6 = \{v_7, v_2^1, v_0^2, v_1^3\}$ are (together) commitments to all leaves *except* $\mathbf{L}_6$.

height k with root y is a set k *complementary* nodes $\mathbf{v}$ that map the leaf node v to sub-tree root y. The complementary nodes of v include a) the sibling of v, and b) the siblings of all ancestors of v. The complementary nodes of a node v can collectively be seen as the set of commitments to all *other* nodes (except v) in the sub-tree.

In a sub-tree of height k, let $u_0^0 \cdots u_{2^k-1}^0$ represent the 2^k nodes at the lowest level (level 0). The immediate parent of a node u_i^0 is $u_{i_1}^1$ at level one, where $i_1 = i/2$ (if i is even) or $(i-1)/2$ (if i is odd). In this fashion, one can readily determine the indexes $u_{i_1}^1 \cdots u_{i_{k-1}}^{k-1}$ of all ancestors of u_i^0, and thereby, the siblings of the ancestors (the sibling of v_m^n is v_{m+1}^n if m is even, or v_{m-1}^n is m is odd).

Given the value v_i^0, the index i of the leaf node, and the set of k complementary nodes, it is trivial to identify the sequence of k hash operations necessary to map a leaf node to the root. We shall represent by

$$y = f_m(v_i, i, \mathbf{v}_i), \tag{3.5}$$

a sequence of k hash operations to obtain the sub-tree root y from a leaf-node with value v and position index i.

3.2.2 Verification and Update Protocols

Protocols that use a binary hash tree can be seen as a two-party protocol involving a prover and a verifier. In the proposed model, the prover is an untrusted subsystem ($\mathbf{U}$), and the verifier is a trusted module $\mathbf{T}$. The prover stores all N records *and* $2N-1$ nodes (cryptographic hashes) of the tree. The verifier $\mathbf{T}$ stores only the root of the tree.

Corresponding to any leaf $\mathbf{L}$, the prover (who maintains the entire tree) can readily identify the set of complementary nodes $\mathbf{v}$. If the prover supplies the leaf $\mathbf{L}$ along with the complementary hashes, the verifier can readily compute $f_m(v = h(\mathbf{L}), i, \mathbf{v})$. If the root stored by the verifier r is the same as $f_m(v, i, \mathbf{v})$, the verifier is simultaneously convinced of a) the integrity of the leaf $\mathbf{L}$, *and* b) the integrity of the complementary hashes $\mathbf{v}$.

Note that by construction of the tree it is guaranteed that $r = f_m(h(\mathbf{L}), i, \mathbf{v})$. That the hash function is pre-image resistant guarantees that the prover cannot determine $\mathbf{L}' \neq \mathbf{L}$ or $\mathbf{v}' \neq \mathbf{v}$ that satisfies $r = f_m(h(\mathbf{L}'), i, \mathbf{v})$ or $r = f_m(h(\mathbf{L}), i, \mathbf{v}')$ (for any i).

Having demonstrated the integrity of the stored record to the module $\mathbf{T}$, if the prover can also demonstrate a legitimate need to modify the record $\mathbf{L}$ to $\mathbf{L}'$, the verifier simply computes $v' = h(\mathbf{L}')$ and updates the root to $r' = f_m(v', i, \mathbf{v})$. Once the leaf has been modified, the old leaf can no longer be proved to be consistent with the new root. However, all other leaves will still remain consistent with the new root (as the complementary nodes $\mathbf{v}$ are the commitments for *all other* leaves).

Note that verifying the integrity of any record in the database, or updating a record will require the security kernel to perform merely $\log_2 N$ hash function evaluations, where N is the number of leaves in the tree.

3.2.3 OMT Leaves and Nodes

An Ordered Merkle Tree (OMT) is an extension of the Merkle tree with the imposition of a special structure for the leaves of the tree. Every leaf is of the form.

$$\mathbf{L} = (A, A', \omega_A) \tag{3.6}$$

where the first field A is unique for all leaves in the tree.

Corresponding to a leaf (A, A', ω_A) is a leaf node v_A, computed as

$$\begin{aligned} v_A &= H_L(A, A', \omega_A) \\ &= \begin{cases} 0 & A = 0, \\ h(A, A', \omega_A) & A \neq 0. \end{cases} \end{aligned} \tag{3.7}$$

Unlike a plain Merkle tree which is intended primarily for dynamic databases with a static number of records (leaves), OMTs are intended to be used for scenarios where leaves may need to be inserted/deleted. For this purpose, it is advantageous to redefine the operation of mapping two siblings u and v to their parent p as

$$p = H_V(u, v) = \begin{cases} u & \text{if } v = 0 \\ v & \text{if } u = 0 \\ h(u, v) & \text{if } u \neq 0, v \neq 0 \end{cases} \tag{3.8}$$

In other words, the parent of two nodes is the hash of the two child nodes *only if both* children are non-zero. If any child is zero, the parent is the same as the other child. The parent of $u = v = 0$ is $p = 0$.

An OMT leaf with the first field set to zero is an *empty* leaf, represented as Φ. The leaf hash corresponding to an empty leaf is 0. As introducing an empty leaf node (corresponding to an empty leaf) does not affect any other node of the tree, any number of empty leaves may be seen part of the tree.

Existence of an OMT leaf (A, A', ω_A) in a tree with root ξ can be verified by computing $\xi' = f_m(H_L(A, A', \omega_A), i, \mathbf{v})$, where $\mathbf{v}$ is a set of complementary hashes, and comparing the values of ξ and ξ'. If the leaf exists in the tree (ie., if $\xi = \xi'$), a verifier can conclude that

1. the value ω_A is associated with a leaf with a unique first-field A.
2. no leaf exists in the tree with a first field value that is *circularly enclosed* by A and A'.

A value C is circularly enclosed by (A, A'), if $cov((A, A'), C)$ is true, where

$$
\begin{aligned}
&cov((A, A'), C)\{ \\
&\text{RETURN } (A < C < A') \vee (C < A' < A) \vee (A' < A < C); \\
&\}
\end{aligned}
\tag{3.9}
$$

For example, $(1, 442)$ encloses all values greater than 1 and less than 442 (or $cov((A, A'), x)$ is TRUE for $1 < x < 442$. The pair $(5, 1)$ (with first value *greater* than the second) circularly encloses all values greater than 5 *and* all values less than 1. In other words, $cov((5, 1), x)$ is TRUE for $x < 1$ *and* $x > 5$.

The existence of a leaf $(A, A' = A, \omega_A)$ indicates that it is the *sole* leaf of the tree.

3.2.4 OMT Types

OMTs can be seen as falling under two broad categories depending on the interpretation of the first two values. In the first category are index ordered MTs (IOMT), where the first value is interpreted as an index, the second value is the next higher index in the tree. For the leaf corresponding to the highest index the next index is the least index.

The third value ω_A in a leaf (A, A', ω_A) provides some information regarding index A. For example, ω_A could be the hash of the contents of a database record with index A. It is also possible that ω_A is a root of another OMT, in which case A is an index of a database (which may consist of any number of indexed records).

For range ordered MT (ROMT) the values A and A' represent the range $[A, A')$ of some quantity associated with the third value ω_A. For example, a leaf like $(432, 562, \omega)$ indicates that the quantity ω is associated with a range $[432, 562)$ (or $432 \leq x < 562$). For example, an ROMT may be used to represent a *look up table* (LUT) for some function $y = f(x)$. In such an ROMT each leaf indicates a range of the independent variable x, corresponding to which the function evaluates to the dependent variable $y = \omega$ (the third value in the leaf).

3.3 OMT Properties

Some of the important properties of OMTs are as follows:

1. The leaf hash corresponding to an empty leaf Φ is zero.

2. A tree with root 0 can be seen as a tree with *any* number of empty leaves.

3. For a tree with a single leaf, the leaf hash is the same as the root of the tree.

4. The existence of a leaf (A, A, ω_A) in an OMT indicates that the leaf is the only leaf in the tree (in which case the root will be the same as the leaf hash $H_L(A, A, \omega_A)$).

5. Existence of a leaf like $(1, 3, \omega_1)$ (Figure 3.2) is proof that no leaf exists with first field in-between 1 and 3. Existence of a leaf like $(7, 1, \omega_7)$ is proof that no leaf exists with first field less than 1 *and* that no leaf exists with first field greater than 7.

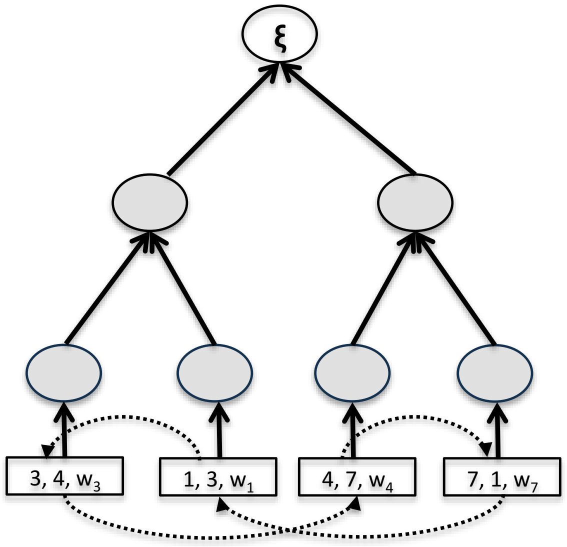


Figure 3.2

OMT with 4 leaves as $(1, 3, \omega_1), (3, 4, \omega_3), (4, 7, \omega_4), (7, 1, \omega_7)$

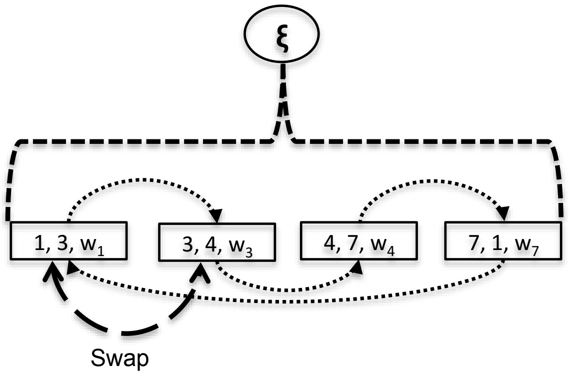


Figure 3.3

Equivalent OMT by swapping of leaves $(3, 4, w_3)$ and $(1, 3, w_1)$

6. As leaves are ordered virtually, the actual physical ordering of leaves has no inherent meaning. Thus, swapping leaves of an OMT (Figure 3.3) does not affect the integrity of the database represented by the OMT. For example, both

$$\begin{aligned} &(1, 3, \omega_1), (3, 4, \omega_3), (4, 7, \omega_4), (7, 1, \omega_7) \text{ and} \\ &(3, 4, \omega_3), (1, 3, \omega_1), (4, 7, \omega_4), (7, 1, \omega_7) \end{aligned} \tag{3.10}$$

represent an identical database with four records — either an IOMT corresponding to four indexes 1,3,4, and 7, or an ROMT for four ranges $[1, 3)$, $[3, 4)$, $[4, 7)$ and $\{7, 1\}$ representing all values greater than or equal to seven, or less than 1).

7. For both IOMT and ROMT, a leaf with a first field A can be inserted only if a leaf that circularly encloses A exists. (B, B') is a circular enclosure for A only if

$$(B < A < B') \text{ OR } (B' < B < A) \text{ OR } (A < B' < B) \tag{3.11}$$

For inserting a leaf the contents of two leaves in the tree will need to be modified; and empty leaf Φ will be modified to become the newly inserted leaf with first values as (A, B'), and the second value of the enclosing leaf will be modified from (B, B') to (B, A).

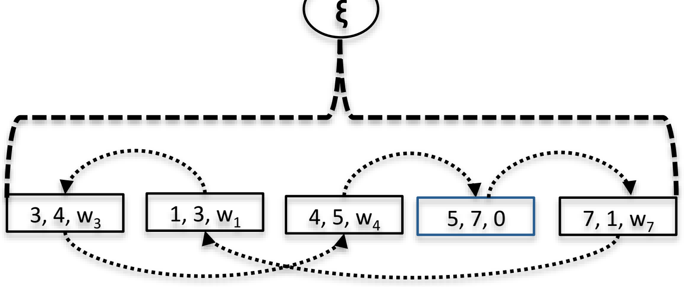


Figure 3.4

Equivalent IOMT with inserted placeholder $(5, 7, 0)$

8. A place-holder is a non-empty leaf whose insertion does not change the interpretation of the database. For an IOMT (Figure 3.4), a leaf of the form $(A, A', 0)$ (third value zero) is a place holder. Introduction of a place holder for an index A does not change the database in any way, as both existence of place holder for index A and non-existence of a leaf for index A implies that "no record exists for index A." Thus,

$$\begin{aligned}&(3, 4, \omega_3), (1, 3, \omega_1), (4, 7, \omega_4), (7, 1, \omega_7) \text{ and}\\&(3, 4, \omega_3), (1, 3, \omega_1), (4, 5, \omega_4), (5, 7, 0), (7, 1, \omega_7)\end{aligned} \tag{3.12}$$

which correspond to before and after insertion of a place holder for an index 5, represent identical databases.

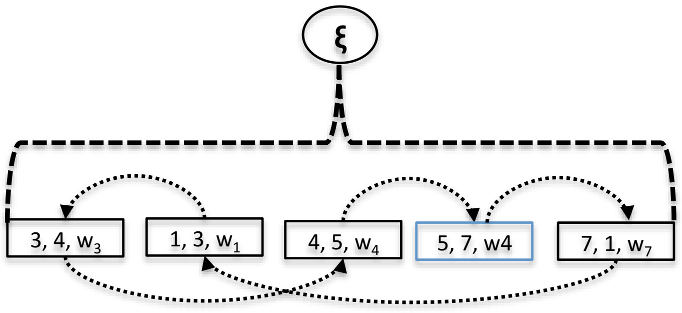


Figure 3.5

Equivalent ROMT with inserted placeholder $(5, 7, w_4)$

9. For an ROMT (Figure 3.5), a place holder is a leaf with third value the same as the third value of the enclosing leaf. Specifically, inserting a leaf can be seen as a process

of splitting a leaf (for example), $(4, 7, \omega_4)$ into two leaves (for example) $(4, 5, \omega_4)$ and $(5, 7, \omega_4)$. Specifically, both

$$\begin{aligned} &(1,3,a),(3,4,b),(4,7,c),(7,1,d) \text{ and} \\ &(1,3,a),(3,4,b),(4,5,c),(5,7,c),(7,1,d) \end{aligned} \tag{3.13}$$

represent identical databases. Before insertion, the leaf $(4, 7, c)$ indicated that values $4 \leq x < 7$ are associated with c. Nothing has changed after the range is split into two, as values $(4 \leq x < 5)$ *and* values $(5 \leq x < 7)$ are associated with the same quantity c.

10. While operations like swapping leaves in any OMT or insertion/deletion of a place holder do not change the contents of the database, they will result in a change in the root of the tree — say from r to r'. Such roots are considered as *equivalent* roots.

3.4 OMT Functions for Security Kernels

The module $\mathbf{T}$ is assumed to possesses limited protected storage, and exposes well defined interfaces to the associated untrusted subsystem. Such interfaces can be used by a untrusted subsystem (say) $\mathbf{U}$ to demonstrate the integrity of databases stored by the subsystem, and request $\mathbf{T}$ associated with subsystem $\mathbf{U}$ to attest verified records.

For attesting records or contents of records (for verification by other subsystems, or security kernels in other subsystems) every module is assumed to possess a unique identity, and secrets used for authenticating messages. For example, the secret could be a private component of an asymmetric key pair, which is used for signing messages. In this case, the public key of the module is certified by a trusted key distribution center, attesting the integrity of the module. Alternately one or more secrets could be provided by a trusted key distribution centers to each module. Only modules that have been verified for integrity and issued such secrets by the trusted key distribution centers will be able to use their secrets to compute a pairwise secret with other modules attested by the KDCs. Such pairwise secrets may be used to compute message authentication codes for attesting the integrity of the contents of a record.

Apart from secrets provided by trusted KDCs or certified by trusted certificate authorities, every module is assumed to spontaneously generate a random self-secret χ which is used for authenticating memoranda to itself. For example, after executing (say) $z = f_m(x, i, \mathbf{v})$ a module may issue a memoranda to itself to remind itself that it has already verified that "z is an ancestor of x." As we shall see very soon, the self-memoranda in this scenario is a value $\rho = h(V1, x, z, \chi)$ computed as a function of the type $V1$ of the memoranda, the values x and z, and the secret χ. No entity other than the module can fake such a memorandum. Thus, if values x, z, and ρ are provided as inputs to the module, the module can safely conclude that "z is an ancestor of x."

In the rest of this section we provide an algorithmic description of generic OMT functions suitable for security kernels for a wide range of systems/ subsystems. OMT functions issue different types of self-memoranda. Such self-memoranda may then be used by other system-specific (or role-specific) security kernel components of the same module. As an illustration of how such memoranda can be used by other system-specific security kernel components of the same module, in a later chapters we outline the use of such memoranda in security kernels for various systems (file storage server, content distribution infrastructure, BGP subsystems, MANET subsystems).

3.4.1 OMT Memoranda

Five different types of memoranda are issued by OMT functions.

A certificate of type $U1$ is issued by functions $F_{bt}()$ and $F_{cat1}()$. The inputs to $F_{bt}()$ include a leaf node x in a subtree, the index i of the leaf node (in the sub-tree), and complementary nodes $\mathbf{v}$. The root of the subtree can now be computed as $y = f_m(x, i, \mathbf{v})$. The function also accepts another value x' and computes $y' = f_m(x, i, \mathbf{v})$ (using the same complementary nodes). The certificate of type $U1$ issued by this function, viz,

$$\rho = h(U1, x, y, x', y', \chi) \tag{3.14}$$

states that "(it has been verified by me that) y is the root of a sub-tree with leaf node x, and if $x \to x'$ then $y \to y'$." More generally, such a certificate implies that y is an *ancestor* of x, and that if $x \to x'$, then $y \to y'$.

Functions $F_{cat1}()$ and $F_{cat2}()$ combine self memoranda to issue (in general) more complex self-memoranda. $F_{cat1}()$ accepts inputs necessary to verify the integrity of two type $U1$ certificates. If the second certificate is 0, and in the first certificate binding x_1, y, x_1', y, if $x_1 = x_1'$ (implying merely that y is an ancestor of x_1), a certificate of type $V1$, viz., $\rho = h(V1, x_1, y, \chi)$ is issued.

If the child in the second certificate x_2 is the same as the parent y in the first certificate, the two certificates are combined to issue a single certificate of type $U1$ binding the child x_1 in the first certificate to the parent z in the second certificate. Else, $F_{cat1}()$ computes $p = H_V(y, z)$ and $p' = H_V(y', z')$ to issue a certificate of type $U2$

$$\rho = h(U2, x_1, x_2, p, x_1', x_2', p', \chi) \tag{3.15}$$

```
$F_{bt}(x, x', i, \mathbf{v}_x)$ {
    $y \leftarrow f_m(x, i, \mathbf{v}_x); y' \leftarrow (x = x')?\, y : f_m(x', i, \mathbf{v}_x);$
    RETURN $h(U1, x, y, x', y', \chi);$
}
    $F_{cat1}(x_1, y, x'_1, y', \rho_1, x_2, z, x'_2, z', \rho_2)$ {
    IF $(\rho_1 \neq h(U1, x_1, y, x'_1, y', \chi))$ RETURN;
    IF $((\rho_2 = 0) \wedge (x_1 = x'_1))$ RETURN $h(V1, x_1, y, \chi);$
    IF $(\rho_2 \neq h(U1, x_2, z, x'_2, z', \chi))$ RETURN;
    IF $((y = x_2) \wedge (y' = x'_2))$ RETURN $h(U1, x_1, z, x'_1, z', \chi);$
    $p \leftarrow H_V(y, z); p' \leftarrow H_V(y', z');$
    RETURN $h(U2, x_1, x_2, p, x'_1, x'_2, p', \chi);$
}
    $F_{cat2}(x_1, x_2, y, x'_1, x'_2, y', \rho_1, z, z', \rho_2)$ {
    IF $(\rho_1 \neq h(U2, x_1, x_2, y, x'_1, x'_2, y', \chi))$ RETURN;
    IF $((\rho_2 = 0) \wedge (x_1 = x'_1) \wedge (x_2 = x'_2))$ RETURN $h(V2, x_1, x_2, y, \chi);$
    IF $(\rho_2 \neq h(U1, y, z, y', z', \chi))$ RETURN;
    RETURN $h(U2, x_1, x_2, z, x'_1, x'_2, z', \chi);$
}
```

Figure 3.6

Verification and Update Memoranda.

to the effect that "x_1 and x_2 are leaf nodes of a sub-tree with root p, and if $x_1 \to x_1'$ and $x_2 \to x_2'$ then $p \to p'$." Note that if y is an ancestor of x_1 and z is an ancestor of x_2, then $p = H_V(y, z)$ is simultaneously an ancestor of x_1 and x_2.

Function $F_{cat2}()$ extends the common ancestor y of two nodes to an ancestor z of y. In other words, $F_{cat2}()$ combines a $U2$ certificate with a $U1$ certificate to produce a $U2$ certificate. If only a certificate of type $U2$ is provided as input to $F_{cat2}()$ with $x_1 = x_1'$ and $x_2 = x_2'$, bound to $y = y'$, $F_{cat2}()$ issues a certificate of type $V2$ binding two nodes x_1 and x_2 to a common ancestor y.

Certificates of type $U1$ and $U2$ are useful for simultaneously verifying and updating the root of the tree. Certificates of type $V1$ and $V2$ are useful in scenarios where only verification is required.

Functions $F_{ph}()$, $F_{sw}()$ and $F_{ce}()$ create certificates that bind equivalent roots. A certificate of $\rho = h(EI, y, y', \chi)$ attests to the equivalence of IOMT roots y and y'. A certificate $\rho = h(ER, y, y', \chi)$ attests to the equivalence of ROMT roots y and y'.

Through a certificate of type $U2$, $F_{sw}()$ recognizes the relationship between two roots resulting from swapping two leaves. As swapping leaves does not affect the integrity of an IOMT or an ROMT, the roots are equivalent for both IOMT and ROMT. Thus, depending on the value o which identifies the type of request ($o = 1$ for ROMT certificate), $F_{sw}()$ outputs a EI or ER certificate.

Function $F_{ph}()$ issues equivalence certificates binding roots before and after deletion of a place holder. The input $o = 1$ is a request to issue an ER certificate (else, the request is for an EI certificate). If no certificate is provided as input to $F_{ph}()$ (or $\rho = 0$), one root

```
  $F_{sw}(x_1, x_2, y, y', \rho, o)$   {
  IF $(\rho = h(U2, x_1, x_2, y, x_2, x_1, y', \chi))$
    IF $(o = 1)$ RETURN $h(ER, y, y', \chi)$;
    ELSE RETURN $h(EI, y, y', \chi)$;
}
  $F_{ph}(A, A', \omega_A, B', y, y', \rho, o)$   {
  IF $(\rho = 0) \wedge (o = 1)$ RETURN $h(ER, 0, H_L(A, A, 0), \chi)$
  ELSE IF $(\rho = 0)$ RETURN $h(EI, 0, H_L(A, A, 0), \chi)$
  $x_2 \leftarrow (o = 1)?\, H_L(A', B', \omega_A) \,:\, H_L(A', B', 0)$;
  $x_1 \leftarrow H_L(A, A', \omega_A); x'_1 \leftarrow H_L(A, B', \omega_A); x'_2 \leftarrow 0$;
  IF $((\rho = h(U2, x_1, x_2, y, x'_1, x'_2, y', \chi)) \vee$
    $(\rho = h(U2, x_2, x_1, y, x'_2, x'_1, y', \chi)))$
    RETURN $(o = 1)?\, h(ER, y, y', \chi) \,:\, h(EI, y, y', \chi)$;
}
  $F_{ce}(i, x, y, \rho_1, z, \rho_2)$   {
  IF $i \notin \{EI, ER\}$ RETURN;
  IF $((\rho_2 = 0) \wedge (\rho_1 = h(i, x, y, \chi)))$ RETURN $h(i, y, x, \chi)$;
  IF $((\rho_1 = h(i, x, y, \chi)) \wedge (\rho_2 = h(i, y, z, \chi)))$
    RETURN $h(i, x, z, \chi)$;
}
```

Figure 3.7

OMT Functions for Issuing Equivalent-Root Memoranda.

is assumed to be the root of an empty tree, and the equivalent root is after insertion of the first place-holder for an index A. For both IOMT and ROMT the first place holder will be $(A, A, 0)$, and the root after insertion will be $H_L(A, A, 0)$. Specifically, 0 is equivalent to $v_x = H_L(x, x, 0)$ for *all* x.

If $\rho \neq 0$ function $F_{ph}()$ interprets (A, A', ω_A) (with leaf hash x_1) and a place-holder (A', B', ω) (with leaf hash x_2) as two leaves in a tree with root y. If $o = 1$ (ROMT) the place holder has $\omega = \omega_A$, else (for an IOMT), $\omega = 0$. After deletion of the place holder the two leaves will need to be modified to (A, B', ω_A) (leaf-hash x_1') and an empty leaf (leaf hash 0) respectively. The certificate ρ attests that modifying two leaves x_1 and x_2 to x_1' and x_2' is equivalent to changing the root from y to y'. Hence, y and y' are equivalent roots. [4]

The equivalent root certificate generated by $F_{ph}()$ for deleting a placeholder can also be utilized for inserting a placeholder. More specifically, if the current root of the OMT is y then changing it to y', denotes deletion of the placeholder $(A', B', 0)$ from the OMT. However, if the current root of the OMT is y' then the same equivalence certificate can be used to update it to y, implying insertion of placeholder leaf $(A', B', 0)$ into the tree. In other words, y' represents a state of OMT where the placeholder is absent, and y represents existence of the placeholder, thus the equivalence certificate can be used for both purposes of inserting or deleting a placeholder.

Functions $F_{ce}()$ combines two equivalence certificates. If $x \leftrightarrow y$ and $y \leftrightarrow z$, the module can conclude that $x \leftrightarrow z$.

[4]In the provided $U2$ certificate, the ordering of the two leaf hashes in the certificate (which can be $x1, x2$ or $x2, x1$) depends on their respective physical positions in the tree. Hence, the leaves are compared for both the conditions 1) placeholder (x_2) is to the left of x_1, for which ($\rho = h(U2, x_2, x_1, y, x_2', x_1', y', \chi)$) or 2) placeholder is to the right of x_1, for which ($\rho = h(U2, x_1, x_2, y, x_1', x_2', y', \chi)$).

3.4.2 Sub-Trees and OMT Memoranda

Any internal node in an OMT can be seen as a root of a sub-tree. More specifically, the common ancestor of a set of leaves in an OMT is the root of the sub-tree the leaves are a part of. The scenario is illustrated in Figure 3.8, where two sub-trees (Sub-tree 1 — root v_1^3, Sub-tree 2 — root v_2^3) which are a part of primary tree with root ξ (Figure 3.1) are shown.

Taking an example leaf L_c, in sub-tree 1. A leaf verification certificate of the form $\rho_1 = h(U1, x = v_c, y = v_1^3, x' = v_c, y' = v_1^3, \chi)$, can be generated by $f_{bt}()$ by computing the root $v_1^3 = f_m(v_c, i = 4, \mathbf{v} = \{v_d, v_7^1, v_2^2\})$, where index i of leaf L_c in the tree is 5. In comparison, for sub-tree 2 with root v_3^2, the index of the leaf L_c is $i = 0$ and $v_3^2 = f_m(v_c, i = 0, \mathbf{v} = \{v_d, v_7^1\})$. The certificate generated is $\rho_2 = h(U1, x = v_c, y = v_3^2, x' = v_c, y' = v_3^2, \chi)$, suggesting L_c represented by leaf node v_c is a part of tree with root v_3^2.

The leaf L_c can also be mapped to the primary tree root ξ by using two $U1$ certificates of the form $\rho_1 = h(U1, x = v_c, y = v_1^3, x' = v_c, y' = v_1^3, \chi)$ (sub-tree 1) and $\rho_\xi = h(U1, x = v_1^3, y = \xi, x' = v_1^3, y' = \xi, \chi)$ (primary tree). As the certificates imply v_1^3 is an ancestor of v_c and ξ is an ancestor of v_1^3, a $V1$ certificate can be generated of the form $\rho = h(V1, x_1 = v_6, y = \xi, \chi)$, mapping ξ as an ancestor of v_6 (leaf L_c).

In order to update a single leaf ($L_c \to L_c'$) in sub-tree 2, a $U1$ certificate can be used of the form $\rho = h(U1, x = v_c, y = v_2^3, x' = v_c', y' = v_2^{3'}, \chi)$, suggesting is $L_c \to L_c'$ ($v_c \to v_c'$), then $v_2^3 \to v_2^{3'}$. To update two leafs at once, for example $L_c \to L_c'$ and $L_e \to L_e'$, a $U2$ certificate of the form $\rho = h(U2, x_1 = v_c, x_2 = v_e, p = v_2^3, x_1' = v_c', x_2' = v_e', p = v_2^{3'}, \chi)$ can be used it update the root $v_2^3 \to v_2^{3'}$.

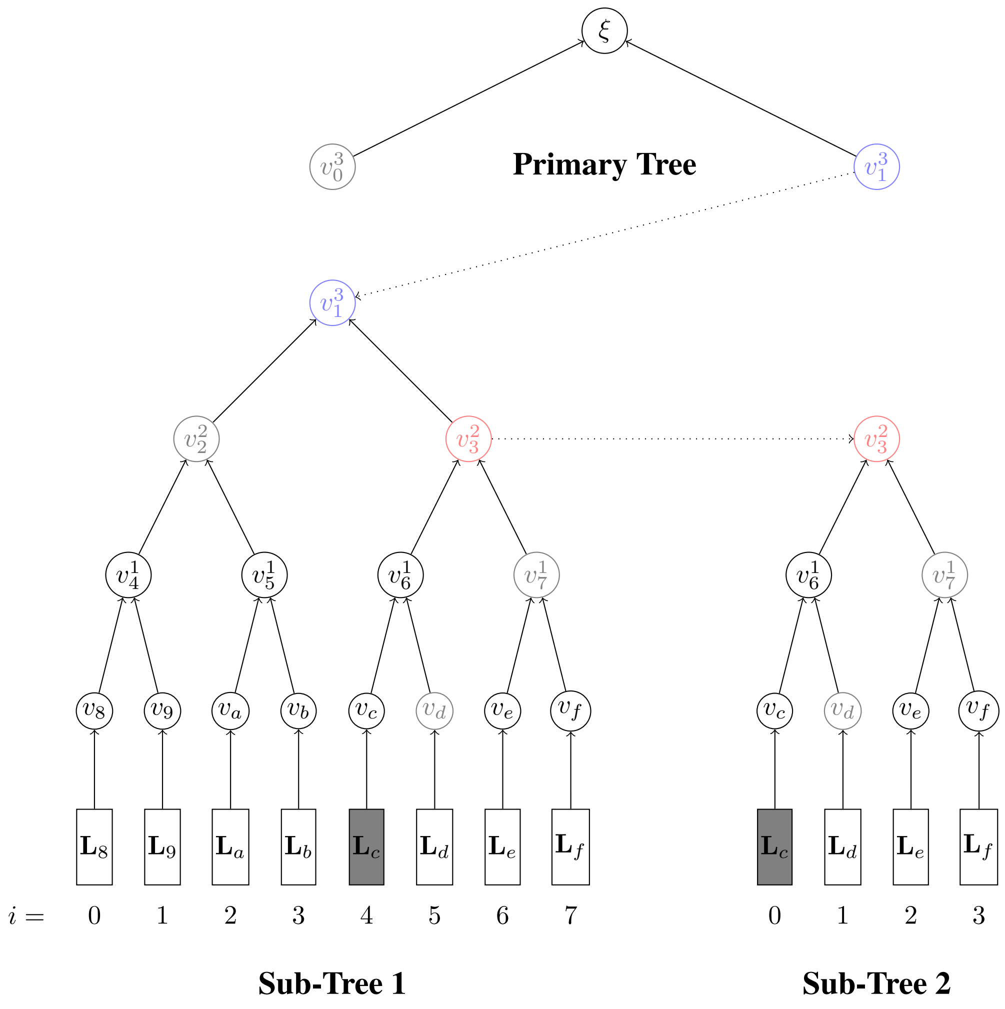


Figure 3.8

Sub-Trees in OMT

Any two leaves in a sub-tree can be swapped (physical positioning in the tree) by invoking the $F_{sw}()$ with a $U2$ certificate . For example, in our case if leaf L_b (v_b) and L_e (v_e) are to be swapped in sub-tree 1 (root v_1^3), the $U2$ certificate provided as input to $F_{sw}()$ is $\rho = h(U2, v_b, v_e, v_1^3, v_e, v_b, v_1^{3\prime}, \chi)$. The function outputs an equivalence certificate of the form $h(o_t, v_1^3, v_1^{3\prime}, \chi)$ ($o_t = EI/ER$ depending on the type of OMT - IOMT/ROMT), where v_1^3 and $v_1^{3\prime}$ are equivalent roots. Equivalent root certificates can also be generated by swapping of internal nodes at the same level (for example swapping of nodes v_2^2 and v_3^2).

The ability to swap leaves/nodes in an OMT is particularly advantageous in certain scenarios. For example, if L_c and L_a are two frequently updated records of the tree, it would be more computationally more efficient if L_a was swapped with any of the leaves of sub-tree 2 (for example, L_d). This feature can be useful, for example, in a remote file storage system where each leaf corresponds to a file. In such an application the files that are actively being edited at any time is typically substantially smaller than the total number of files.

The ability to swap leaves also lends to efficient delegations of credentials represented by the leaves. For example, a subsystem may store various credentials as leaves of an OMT. Subsets of credentials that need to be delegated can be physically ordered together, and the root of the sub-tree can now efficiently represent all delegated credentials. As we shall see later in this dissertation, this feature is utilized by autonomous system (AS) owners to delegate sub-sets of IP addresses they own to different BGP speakers of the AS.

CHAPTER 4

REALIZING A SECURE FILE STORAGE SYSTEM

In an increasingly interconnected world with anywhere-anytime-access-to-anything, remote file storage services[1] are a popular approach for storing files for easy access, and collaboration between users. Users of such a service may create files for access from numerous platforms owned by the same user, and/or for access by other users explicitly permitted by the creator of the file.

From a security perspective, users (client subsystems) of such a service desire some assurances regarding the *integrity*, *privacy*, and *availability* of files stored at a remote location. Specifically, users desire that

1. files cannot be modified at the server (central subsystem), except by users explicitly authorized to do so by the owner;
2. only the latest version of the file should be provided (if a file has been modified by a legitimate user, the service will not *replay* older versions of the file — unless a older version is explicitly solicited);
3. contents of the files should not be revealed to unauthorized parties; and
4. authorized users will *not* be denied access to the file by the service.

A practical file storage service may be composed of perhaps several computers (servers), running complex and buggy application software, on top of bug-ridden operating systems,

[1]Well known real-world services include Dropbox, Google Drive, iCloud, Skydrive, etc.

which run on inexpensive general purpose computers, constructed using *untrustworthy* off-the shelf hardware. It is far from practical to rule out undesired functionality in every component of the service. Such undesired functionality could be a result of accidental bugs or malicious trojan-horses, and may be exploited to gain control of one or more components of the service. Personnel who may exploit such functionality can be malicious employees of the service, or an external attacker — who is either able to trick/bribe an employee, or utilize a remote exploit. Ultimately, undesired functionality in *any* component may lead to violation of the assurances desired by users.

4.1 File Storage Service: Desired Features and Assurances

In the proposed generic model for file storage service, the participants/subsystems include an untrusted service provider $\mathbf{U}$, a trusted module $\mathbf{T}$, and any number of users.

The provider-side components may include one or more public servers with access to large back-end storage. The specific components of the file storage service $\mathbf{U}$ are irrelevant from a security perspective, as such components are assumed to be untrusted.

Module $\mathbf{T}$ spontaneously generates a master secret χ, and a private key R_T corresponding to some asymmetric cryptographic scheme. The module's public key U_T is assumed to be made known[2] to all participants of the system. Module $\mathbf{T}$ possesses a clock-tick counter. At any instant of time, the tick count t is interpreted by the module as the "current time." It is assumed that the module is read-proof and write-proof. In other words, a) secrets χ and

[2]For example, the public key may be certified by one or more trusted entities responsible for verifying and certifying the integrity of the module $\mathbf{T}$.

R_T cannot be exposed, and b) the designed functionality of the module (which is described in later sections) cannot be modified.

The system may possess any number of users/subscribers. Users may join at any time. Every user has a key pair. Within the system, the unique identity assigned to a user with public key U_i is simply $u_i = h(U_i)$. Every user shares a long-lived secret with the module $\mathbf{T}$. Specifically, the long-lived secret shared between user u_i (with key-pair (R_i, U_i)) and the module $\mathbf{T}$ with key-pair (R_T, U_T) is

$$K_i^{ll} = \mathcal{K}(R_i, U_T) = \mathcal{K}(R_T, U_i), \tag{4.1}$$

where the specifics of the function $\mathcal{K}()$ are not important for our purposes. Such shared secrets are used to secure succinct messages exchanged between users and the module.

An (untrusted) component of $\mathbf{U}$ possesses an open channel for communicating with the module $\mathbf{T}$. The module can be physically located in *any* convenient location. As only $\mathbf{U}$ has access to the module, $\mathbf{U}$ serves as an untrusted middle-man in the interactions between users and module $\mathbf{T}$.

4.1.1 Files and ACLs

Any user can create a file, assign a label, choose a secret to encrypt the file, and upload the encrypted file to service $\mathbf{U}$. The file encryption secret is also conveyed to $\mathbf{U}$, after encrypting it using a secret shared between the user and the module.

A file with label l_j created by user u_i is associated with an index

$$f_{ij} = h(u_i, l_j). \tag{4.2}$$

The owner u_i of the file is also required to specify an ACL for the file, which is also uploaded to $\mathbf{U}$. The ACL for a file f_{ij} is of the form

$$\{(x_1, a_1), (x_2, a_2), \ldots, (x_l, a_l)\} \tag{4.3}$$

where $x_1 \cdots x_l$ represent user identities and $a_1 \cdots a_l$ their respective access permissions. Associated with the ACL for f_{ij} is a succinct summary α_{ij} of the ACL. The ACL uploaded to $\mathbf{U}$ is accompanied by the summary α_{ij}, duly authenticated by u_i for verification by the module $\mathbf{T}$.

We assume three types of permissions: $a_i = 1$ implies read-only; $a_i = 2$ implies read-write (such users may create newer versions of the file); and $a_i = 3$ implies read-write access for the file *and* the ACL (such users are allowed to modify the ACL). Once the ACL for a file is modified, the older version of the ACL is expunged. An empty ACL implies that the file should no longer be made available to anyone (all versions of the file f_{ij} may then be expunged by the service $\mathbf{U}$).

In the authenticated message sent from users to the module

1. users authorized to provide new version (say version q) of a file f_{ij} convey a value v_q, which is function of several values like the file hash and the secret used to encrypt version q of file f_{ij};
2. users authorized to modify the ACL convey a succinct summary α' of the new ACL.

4.1.2 Desired Assurances

The desired assurances regarding the operation of the untrusted file storage service $\mathbf{U}$ are as follows:

[A1.] $\mathbf{U}$ will not alter files; in other words, only users explicitly granted the permission (by the owner) to modify the file can do so.

[A2.] $\mathbf{U}$ will not gain access to file-encryption secrets; only the trusted module $\mathbf{T}$ and users explicitly specified in the ACL (for a file f_{ij}) can gain access to the file encryption secret (for any version of f_{ij});

[A3.] $\mathbf{U}$ cannot modify the ACL; only users explicitly granted the permission to modify the ACL can do so;

[A4.] A user u authorized to access file f_{ij} can easily verify if the file is the latest version.

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

[A6.] When a user u in the ACL requests access to file f_{ij}, $\mathbf{U}$ will *not* refuse to provide access to the file. Furthermore, no unsolicited information should need to be revealed by the server to unauthorized users to ensure this requirement.

Assurances A1 is towards *authenticity and integrity* of files; assurances A2 and A3 are required to guarantee *privacy* of file contents; assurances A4 and A5 address *replay* attacks (A4 address replay attacks on files, and A6 addresses replay attacks on ACLs).

Assurance A6 is towards *authenticated denial* — to prevent improper denial of service to a subscriber for a legitimate request. When authenticated denial is mandated, the server is not allowed to simply respond that "the file does not exist," or "you do not have permission." The server is expected to provide a verifiable response to *every* query — either by providing the requested data, or by convincing the querier that the query *cannot be answered*.

As proving existence of an object is easier than proving non-existence, typical strategies for convincing the querier of the non-existence of an object is performed by ordering all objects that *do* exist, and demonstrating the presence of two adjacent objects — say A and C that "cover" (or span) the requested (missing) object B.

An unfortunate side effect of such a strategy is that makes it possible for attackers to query for random objects simply to gain knowledge of the existence of two other objects (that enclose the randomly queried object). This ability to gain unsolicited information (regarding files that actually exist, or users who are actually granted access to a file) can motivate attackers to mine data for nefarious purposes, and thereby a) unduly burden the service provider, and b) compromise the expectations of privacy of user interactions (interactions through a shared file). To prevent such attacks it should be ensured that *no unsolicited information will need to be revealed* by the server for providing authenticated denial.

4.2 Overview of Proposed Approach

In the proposed approach requests from users to the service to create a new file, a new version, or modify the ACL, are accompanied by a authenticated message μ. The response to such requests is an authenticated acknowledgement from the module that "the request has been carried out." In addition, the server **U** can request the module to send information regarding any version of any file to any user. However, the module will authenticate such information, and convey the file-decryption secret to the user, only if the module is able to verify that the user is authorized to receive such information (as per the current ACL for the file).

As long as the user receives an authenticated acknowledgement, or information regarding a file (authenticated by the module $\mathbf{T}$), the user is convinced of the integrity of the service. More specifically, *if* the integrity of the service is violated in any way, the server $\mathbf{U}$ will not be able to deliver specific module-authenticated messages expected by users.

4.2.1 User-Module Secrets

The long lived secret K_i^{ll} is utilized to securely convey short lived secrets. A short lived secret K_i conveyed to user u_i by the module $\mathbf{T}$ is computed as

$$K_i = h(\chi, h(u_i, e_i)) \tag{4.4}$$

where χ is the master secret known only to the module $\mathbf{T}$, and e_i is the time at which the secret K_i expires. The short-lived secret K_i can be used for securing all exchanges between the user u_i and the module $\mathbf{T}$ till time e_i. The choice of the time of expiry e_i (for example, a week, or a month or a year from the time of issue) is left to the discretion of the service provider $\mathbf{U}$.

To convey a short-lived secret to a user with public key U_i the server utilizes a function $F_{sl}()$ exposed by the module:

```
    $F_{sl}(U_i, e_i)\{$
    IF $((e_i < t) \vee (e_i - t > MAX))$ RETURN;
    $K_i^{ll} = \mathbf{K}(R_T, U_i);$
    $u_i = h(U_i); w_i = h(u_i, e_i);$
    $K_i := h(\chi, w_i);$
    RETURN $t, \bar{K}_i := h(K_i^{ll}, e_i, t) \oplus K_i, \mu_t = h(h(t, \bar{K}_i), K_i);$
$\}$
```

Given the public key U_i of a user, along with the desired time of expiry e_i, the module verifies that the time of expiry is in the future (or $e_i > t$), but not further than some duration MAX (or $e_i - t < MAX$). The short lived secret $K_i = h(\chi, h(u_i, e_i))$ is encrypted using the long lived secret K_i^{ll} and can be decrypted only by the user with the secret counterpart R_i of public key U_i.

The server $\mathbf{U}$ sends values $t, e_i, \bar{K}_i$ and μ_t to the user u_i who can then compute

$$K_i := h(K_i^{ll}, e_i, t) \oplus \bar{K}_i \tag{4.5}$$

and verify that $\mu_t = h(h(t, \bar{K}_i), K_i)$.

4.2.2 Module State ξ

Apart from protecting the privacy of secrets R_T and χ, the module $\mathbf{T}$ assures the integrity of a single dynamic value ξ — a cryptographic hash stored inside the module. This dynamic value ξ is a concise summary of values associated with *all* versions of *all* files, and the ACL corresponding to every file. Specifically, the cryptographic hash ξ is the root of a special data structure — an *Index Ordered Merkle Tree* (IOMT) — and can be considered as *the state of the module* $\mathbf{T}$.

Similar to the better known Merkle hash tree [38], an IOMT is a binary tree. A tree with N *leaves* has $2N-1$ internal nodes (cryptographic hashes) distributed over $L = \log_2 N$ levels ($\frac{N}{2^l}$ nodes in levels $l = 0$ to $l = L$). The lone node at level L is the *root* of the tree. The primary difference between a plain Merkle tree and an IOMT is that leaves of an IOMT form a *virtual circular linked list* (more details in Section 3.2).

The module state ξ is the root of an IOMT with one leaf corresponding to every f_{ij} in the system. The number of such leaves are dynamic as new files may be added, existing files may be deleted. The leaf corresponding to a file f_{ij} conveys two values (as shown in Figure 4.2.2) — θ_{ij} and α_{ij} — which are themselves roots of two IOMTs. Specifically,

1. α_{ij} is the root of an "ACL IOMT" for file f_{ij}, and includes one leaf corresponding to each entry (u_i, a_i) in the ACL;
2. θ_{ij} is the root of the "file version IOMT," and includes one leaf of the form (q, v_q) corresponding to every version of the file f_{ij};

The untrusted server $\mathbf{U}$ is expected to maintain

1. the main database and the corresponding main IOMT with N leaves, with root ξ.
2. a version database and its IOMT, and an ACL database and corresponding IOMT for every f_{ij} ($2N$ databases and $2N$ IOMTs).

The module $\mathbf{T}$ stores and tracks only the root ξ. It is in the server's interest to ensure that the state ξ of the module always remains consistent with the root ξ of the main IOMT maintained by the server. Else, the server will not be able to send authenticated acknowledgements to users.

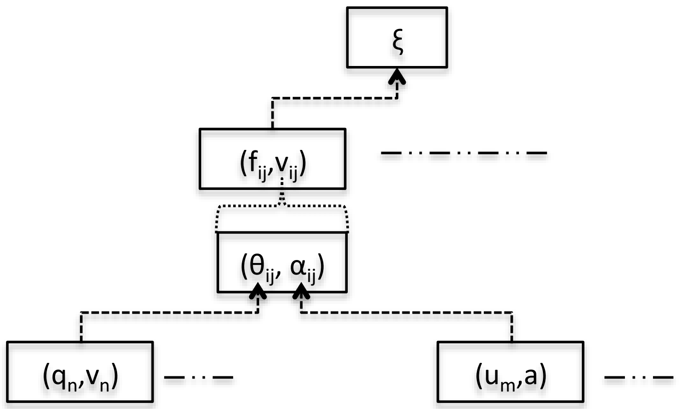


Figure 4.1

OMT roots for Trusted File Storage. [3]

The module executes simple immutable algorithms which accept user authenticated values as inputs and modify the state ξ. Specifically, on receipt of a valid request from a user (to create a new file, update a file, or modify the ACL)

1) the server is expected to make the necessary changes to its database, due to which the root of the IOMT maintained by the server changes from ξ to ξ';

2) the server is then required to submit a user authenticated request μ, to modify the state of the module $\mathbf{T}$ to ξ'.

3) the module executes immutable deterministic algorithms of the form $\mathcal{F}(\mu) : \xi \rightarrow \xi'$ to change it's state, and outputs an authenticated acknowledgement verifiable by the user.

[3]Figure Notes: The static descriptor ξ can be seen as the super root of the tree. θ_{ij} is the root of the left sub-tree keeping track of the file versions of f_{ij}. α_{ij} is the root of the right sub-tree which maintains the ACL for authorized users for f_{ij}.

Specifically, as the server is *required* to invoke $\mathcal{F}$ to obtain an authenticated response to the user, and as execution of the algorithm will result in the change of module state to ξ', the server is forced to modify its database to remain in sync with the module.

In the rest of this chapter we describe such simple deterministic algorithms $\mathcal{F}()$ executed by the module, as a set of discrete functions.

4.2.3 Module T Functions

The functions exposed by the module can be broadly classified into three categories

1. functions used to change the state ξ of the module (by providing user authenticated inputs), and
2. functions that depend on the state ξ, but do *not* modify the state, and
3. state independent functions

The functions in the first category include

1. $F_{in}()$ which is invoked when a new file is created by a user;
2. $F_{upd}()$ which is invoked when a file is updated (a new version of a file is created);
3. $F_{acl}()$ which is invoked when an ACL is updated;

A function $F_{rep}()$ belongs to the second category. This function enables the server to *report* values corresponding to a version of file to a user (only if the values to be reported are consistent with the state ξ of the module). The function is invoked to send parameters of associated with a specific version of a file (file hash, encryption secret, version number etc.) to an authorized user, or to report to the user that the user has no permission to access the file.

The state-independent functions include

1. function $F_{sl}()$ (discussed earlier in Section 4.2.1) which is invoked to convey short-lived secrets to users; and
2. several "certificate generation" functions.

The certificate generation functions (described in Section 3.2) are highly reusable, and independent of the specific application domain (file storage system). Such functions verify state-independent facts like "x is a child node of y" (in a binary hash tree), or "x is a child of y, *and* if $x \rightarrow x'$ then $y \rightarrow y'$,", etc., and output *self-certificates*.

A self certificate is a "memorandum issued to one-self" [13]; a self certificate created by $\mathbf{T}$ is intended for verification by itself at a later time. For our purposes, a self-certificate is a MAC computed using a symmetric secret χ known only to $\mathbf{T}$. For example, a function may verify that x is a child of y and output a self-certificate of type $V1$ (ancestor certificate) computed as $\rho_{vu} = h(V1, x, y, \chi)$, to the effect that "(it has been verified by me that) x is a child of y." Note that such certificates cannot be impersonated without the knowledge of the secret χ. Certificate generation functions perform several OMT-specific operations required to infer relationships between internal nodes of the tree, and relationships between leaves and nodes of a tree.

Apart from the OMT self-memoranda certificates ($U1, U2, V1, V2$) (described in Section 3.2), two additional certificates are introduced for the purposes of this model, 1) leaf verification certificate (LV) and 2) leaf update certificate (LU). A certificate of type LV (leaf verification) states that an OMT with root y possesses a leaf with index a bound to a value v_a. If $v_a = 0$ in the certificate, the implication is that either a) no leaf exists with an index a, or b) a place-holder (a leaf with $v_a = 0$) exists in the tree with root y (the distinction is not important as the two cases are equivalent).

Given a certificate $\rho_{v1} = h(V1, x, y, \chi)$ and values (a, a', v_a) satisfying $x = H_L(a, a', v_a)$, it can be inferred (using F_{lv}) that a leaf for index a is associated with a value v_a in a sub-tree (or full tree) with root y. On verification, a certificate of type LV of the form

$$\rho_{lv} = h(LV, a, v_a, y, \chi) \tag{4.6}$$

is issued. Given that (a, a', v'_a) is a leaf of an IOMT with root y, and a value b that is covered by (a, a') it can be concluded that no leaf for index b exists. In such a scenario a LV certificate

$$\rho_{lv} = h(LV, b, 0, y, \chi) \tag{4.7}$$

is issued.

```
$F_{lv}(a, a', v_a, b, y, \rho)\{$
    $x = H_L(a, a', v_a);$
    IF $(\rho \neq h(V1, x, y, \chi))$ RETURN;
    IF $(b = 0)$
        RETURN $\rho_{lv} = h(LV, a, v_a, y, \chi);$
    ELSE IF $(cov((a, a'), b))$
        RETURN $\rho_{lv} = h(LV, b, 0, y, \chi);$
$\}$
```

Similarly, a certificate of type LU (leaf update) states that a leaf for index a with value v_a exists in an IOMT with root y, and that updating the value $v_a \to v'_a$ will require the root to be modified to y'.

Given a (node update) certificate $\rho_{u1} = h(U1, x, y, x', y', \chi)$, values a, a', v_a, v'_a satisfying $x = H_L(a, a', v_a)$ and $x' = H_L(a, a', v'_a)$ the function $F_{lu}()$ issues a certificate

$$\rho_{lu} = h(LU, a, v_a, y, v'_a, y', \chi). \tag{4.8}$$

```
    $F_{lu}(a, a', v_a, y, v'_a, y', \rho_{u1})\{$
    $x = H_L(a, a', v_a); x' = H_L(a, a', v'_a);$
    IF $(\rho_{u1} \neq h(U1, x, x', y, y', \chi))$ RETURN;
    RETURN $\rho_{lu} = h(LU, a, v_a, y, v'_a, y', \chi);$
$\}$
```

4.3 Application Specific Module $\mathbf{T}$ Functionality

Thus far we have investigated several generic functions exposed by module $\mathbf{T}$. Specifically, function $F_{sl}()$ for conveying short-lived secrets to users was discussed in Section 4.2.1.

In this section we illustrate module functions specific to remote file storage systems. Such functions enable the module to assure the integrity of dynamic databases stored by an untrusted entity $\mathbf{U}$ — by tracking a single cryptographic hash ξ (the root of an IOMT). Specifically, the IOMT root ξ stored inside the module $\mathbf{T}$ (or the module *state*) is a succinct representation of all information regarding *every* version of *every* file, *and* the ACL for every file.

4.3.1 IOMTs for File Storage System

Each leaf of the IOMT with root ξ corresponds to a file with index f_{ij} (as shown in Figure 4.2). For a file owned by user u_i with label l_j, the unique file index f_{ij} is computed as

$$f_{ij} = h(u_i, l_j). \tag{4.9}$$

By obtaining a certificate of the form

$$\rho_{lv} = h(LV, f_{ij}, v_{ij}, \xi, \chi) \tag{4.10}$$

the server $\mathbf{U}$ can demonstrate to the module $\mathbf{T}$ (which trusts only the state ξ) that "the current state is consistent with the existence of a file f_{ij} associated with a value v_{ij}."

The value v_{ij} is of the form

$$v_{ij} = h(\theta_{ij}, \alpha_{ij}) \tag{4.11}$$

where

1. α_{ij} is the root of an "access control IOMT" for file f_{ij}.
2. θ_{ij} is the root of another IOMT a "file version IOMT"

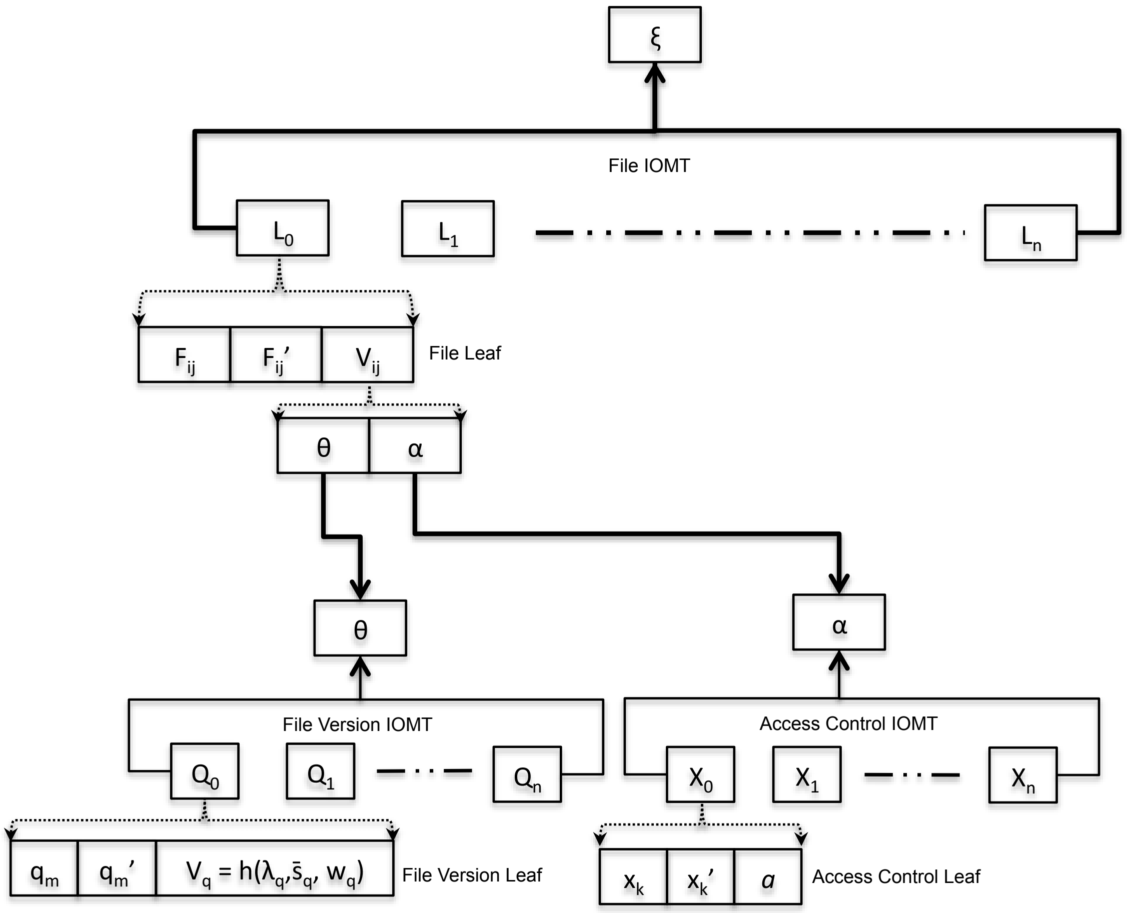


Figure 4.2

Data-Structure for File Storage. [4]

The server can invoke a $F_{cr}()$ at any time to modify the root ξ to any equivalent root ξ' by producing a certificate of type EI. Similarly, function $F^{v}_{cr}()$ can be used to change the version tree root from θ_{ij} to an equivalent root θ'_{ij} (corresponding to which the root of the main tree will need to be changed from ξ to ξ').

[4]Figure Notes: F_{ij} is the file index, θ and α are the roots of version tree and access control tree respectively.q_m is the sequence number of a particular version, V_q is the hash of file hash λ_q, encrypted secret s'_q and w_q. w_q is a hash of provider identity u_l and their key expiry time e_l. X_k is the user identity having access control a.

```
    $F_{cr}(\xi', \rho_{eq})\{$
    IF $(\rho_{eq} = h(EI, \xi, \xi', \chi))$
        $\xi := \xi'$; RETURN $\xi$;
}
    $F_{cr}^{v}(f_{ij}, \alpha_{ij}, \xi', \rho_{lu}, \theta_{ij}, \theta'_{ij}, \rho_{eq})\{$
    $v_{ij} = h(\alpha_{ij}, \theta_{ij}); v'_{ij} = h(\alpha_{ij}, \theta'_{ij})$;
    IF $(\rho_{lu} \neq h(LU, f_{ij}, v_{ij}, \xi, v'_{ij}, \xi', \chi))$ RETURN;
    IF $(\rho_{eq} = h(EI, \theta_{ij}, \theta'_{ij}, \chi))$ $\xi := \xi'$;
}
```

A leaf (u_l, u'_l, a_l) of the access control IOMT (with root α_{ij}) corresponds to an entry in the ACL for file f_{ij}. Specifically, the implications of the presence of such a leaf are

1. user u_l possesses access permission a_l, and
2. any user u enclosed by (u_l, u'_l) has access permission 0 (or, does *not* have access rights)

If a leaf for u_l exists in an ACL tree with root α_{ij}, the server can obtain a certificate

$$\rho_{lv}^{ac} = h(LV, u_l, a_l, \alpha_{ij}, \chi) \tag{4.12}$$

indicating the access privilege associated with a user u_l in an IOMT with root α_{ij}. If *no* leaf for u_l exists the server will be able to obtain a certificate of the form

$$\rho_{lv}^{ac} = h(LV, u_l, 0, \alpha_{ij}, \chi) \tag{4.13}$$

The file version IOMT has leaves of the form (q, q', v_q). A $U1$ certificate

$$\rho_{u1} = h(U1, x, \theta_{ij}, x, \theta_{ij}, \chi) \tag{4.14}$$

where $x = H_L(q, q', v_q)$ can be used to demonstrate to the module that

1. version q of the file f_{ij} is associated with value v_q; and
2. if $q' = q + 1$, fresher versions of the file f_{ij} exist; and
3. if $q' = 1$, then q is the freshest version.

4.3.2 Reporting File Parameters

The value v_q associated with version q of file f_{ij} is a function of three values:

$$v_q = h(\lambda_q, \bar{s}_q, w_q), \tag{4.15}$$

where λ_q is the file hash, $\bar{s}_q$ is the *encrypted* file encryption secret; The value w_q relates $\bar{s}_q$ to the encryption secret s_q, and the file hash λ_q as

$$s_q = h(h(\chi, w_q), \lambda_q) \oplus \bar{s}_q. \tag{4.16}$$

A user (say u_l) may query the server for *any* version of *any* file f_{ij}. The user may explicitly specify the desired version q, or indicate that he/she simply requires the freshest version.

If the file does not exist, or the user u_l does not have the necessary permission, the user still expects an authenticated response from the module (as the user does not trust the server **U**). Under such scenarios function $F_{rep}()$ outputs a time-stamped a MAC μ_t computed as

$$\mu_t = h(NACK, f_{ij}, t, K_l); \tag{4.17}$$

On receipt of the MAC the user is convinced that either a) the file does not exist, or b) that the user does not have necessary access rights (the user does not need to know to know which of the two possibilities is actually true).

If the file exists, and the user is authorized access to the file, the values returned to the user include

1. the file hash λ_q and encryption secret s_q,
2. the version number q along with the next version number q' (which can be $q+1$ or 1), and
3. the user's access rights a_l.

The user expects such values to be authenticated by the module $\mathbf{T}$. Furthermore, the secret s_q should also be encrypted using the secret K_l shared between the user and the module $\mathbf{T}$. In this case $F_{rep}()$ outputs a time-stamped MAC μ_t, and encrypted file encryption secret s'_q where

$$\begin{aligned} s'_q &= h(K_l, \lambda_q, t) \oplus s_q \\ \mu_t &= h(INFO, f_{ij}, v_{ij}, q, q', \lambda_q, s'_q, a_l, t, K_l). \end{aligned}$$

The inputs $f_{ij}, \theta_{ij}, \alpha_{ij}$ and ρ_{lv} to $F_{rep}()$ enable the module to verify the consistency of the leaf for file f_{ij} against the root ξ.

The inputs u_l and e_l enable the module to compute the short-lived secret it shares with user u_l as $K_l = h(\chi, h(u_l, e_l))$.

The inputs u_l, a_l and ρ^{ac}_{lv} enable the module to verify the access privilege of the user u_l against the ACL root α_{ij}.

The inputs $\{q, q', (\lambda_q, \bar{s}_q, w_q)\}$ and ρ_{nv} enable the module to verify the integrity of the values corresponding to version q of the file f_{ij} and compute the file encryption secret. The encrypted file encryption secret is first decrypted to obtain s_q and re-encrypted as s'_q.

It should be noted that if any place-holder with index q exists in the version tree (a place holder $(q, q', v_q = 0)$ the contents of such a leaf can never be reported as the server cannot provide the pre-images of $v_q = 0$ (values satisfying $h(\lambda_q, \bar{s}_q, w_q) = 0$).

The function $F_{rep}()$ can be algorithmically described as follows:

```
$F_{rep}(f_{ij}, \theta_{ij}, \alpha_{ij}, \rho_{lv}, u_l, a_l, \rho_{lv}^{ac}, q, q', \{\lambda_q, \bar{s}_q, w_q\}, \rho_{u1})\{$
    IF $(e_l < t)$ RETURN;
    $K_l = h(\chi, h(u_l, e_l));$
    IF $((\theta_{ij} = 0) \wedge (\rho_{lv} = h(LV, f_{ij}, 0, \xi, \chi)))$
        RETURN $t, \mu_r := h(NACK, f_{ij}, t);$
    $v_{ij} := h(\theta_{ij}, \alpha_{ij});$
    IF $(\rho_{lv} \neq h(LV, f_{ij}, v_{ij}, \xi, \chi))$ RETURN;
    IF $(\rho_{lv}^{ac} \neq h(LV, u_l, a_l, \alpha_{ij}, \chi))$ RETURN;
    IF $(a_l < 1)$ RETURN $t, \mu_r := h(NACK, f_{ij}, t);$
    $v_q = h(\lambda_q, \bar{s}_q, w_q); x := H_L(q, q', v_q);$
    IF $(\rho_{u1} \neq h(U1, x, \theta_{ij}, x, \theta_{ij}, \chi))$ RETURN;
    $s'_q = h(h(\chi, w_q), \lambda_q) \oplus \bar{s}_q \oplus h(K_l, \lambda_q, t);$
    RETURN $t, s'_q, \mu_t = h(INFO, f_{ij}, v_{ij}, q, q', \lambda_q, s'_q, a_l, t, K_l);$
}
```

4.3.3 User Request Driven State Changes

Function $F_{in}()$ expects an authenticated request from a user u_i to provide the first version of file f_{ij}. In other words, user u_i authenticates values l_j and the value v_{ij} to be bound to f_{ij}. Only if $f_{ij} = h(u_i, l_j)$ is the user u_i recognized as the owner of the file (and therefore permitted to initialize file f_{ij}). To compute the shared key K_i (to verify the MAC μ provided by the user) the expiry time e_i is also provided as input.

Before the file is initialized, the module expects it's current state ξ to be consistent with the existence of a place-holder for f_{ij}. After the file is initialized, as the value 0 in the place holder should be updated to v_{ij}, the module expects as inputs the new root ξ' and an LU certificate

$$\rho_{lu} = h(LU, f_{ij}, 0, \xi, v_{ij}, \xi', \chi). \tag{4.18}$$

This function changes module state to ξ' and outputs an ACK message for verification by the user.

```
$F_{in}(u_i, l_j, v'_{ij}, \xi', \rho_{lu}, e_i, \mu)\{$
    IF $(e_i < t)$ RETURN; // Expired key
    $f_{ij} := h(u_i, l_j); K_i := h(\chi, h(u_i, e_i));$
    IF $(\mu \neq h(f_{ij}, v_{ij}, K_i))$ RETURN; //Incorrect MAC
    IF $(\rho_{lu} \neq h(LU, f_{ij}, 0, \xi, v_{ij}, \xi', \chi))$ RETURN; //Bad Input
    $\xi := \xi'$; //Change state
    RETURN $t, \mu_t = h(ACK, f_{ij}, v_{ij}, t, K_i);$
$\}$
```

Function $F_{acl}()$ is used to modify the ACL for a file. Inputs to this function are

1. f_{ij}, θ_{ij}, the current ACL root α_{ij},
2. u_l, a_l and an LV certificate $\rho_{lv} = h(LV, u_l, a_l, \alpha_{ij}, \chi)$ to demonstrate that a user u_l has access permission $a_l > 2$, and
3. an authenticated message from the user to convey the new ACL root α'_{ij}.

As incorporating the requested change will require the IOMT leaf for index f_{ij} (currently associated with a value $v_{ij} = h(\theta_{ij}, \alpha_{ij})$ to be modified to $v'_{ij} = h(\theta_{ij}, \alpha'_{ij})$ the module expects additional inputs ξ' and ρ_{lu} satisfying

$$\rho_{lu} = h(LU, f_{ij}, v_{ij}, \xi, v'_{ij}, \xi', \chi). \tag{4.19}$$

```
  $F_{acl}(f_{ij}, \alpha_{ij}, u_l, e_l, a_l, \rho_{lv}, \mu, \alpha'_{ij}, \xi', \rho_{lu})\{$
  IF $(e_l < t)$ RETURN; // Expired key
  IF $(a_l < 3)$ RETURN; // No authority
  IF $(\rho_{uac} \neq h(LV, u_l, a_l, \alpha_{ij}, \chi))$ RETURN; //Bad Input
  $K_l := h(\chi, h(u_l, e_l)); v_{ij} = h(\theta_{ij}, \alpha_{ij});$
  $v'_{ij} = (\alpha'_{ij} = 0)?0 : h(\theta_{ij}, \alpha'_{ij});$
//if $\alpha'_{ij} = 0$ user is requesting the file to be deleted
//Set $v'_{ij} = 0$ converting the leaf to a place-holder
  IF $(\mu \neq h(f_{ij}, v_{ij}, \alpha'_{ij}, K_l))$ RETURN; //Incorrect MAC
  IF $(\rho_{lu} \neq h(LU, f_{ij}, v_{ij}, \xi, v'_{ij}, \xi', \chi))$ RETURN; //Bad Input
  $\xi := \xi'$; //Change state
  RETURN $t, \mu_t = h(ACK, f_{ij}, \alpha'_{ij}, t, K_l);$
}
```

If the new ACL root is 0, this is interpreted by the module as a request to *delete* the file. In this case, the value v_{ij} associated with the file is set to 0 — thus changing the leaf to a place-holder.

Function $F_{upd}()$ is invoked to modify state ξ for purposes of inserting a new version of file f_{ij}. Inputs to this function are

1. $f_{ij}, \alpha_{ij}, \theta_{ij}$,
2. u_l, a_l, an LV certificate $\rho_{lv} = h(LV, u_l, a_l, \alpha_{ij}, \chi)$ to demonstrate that a user u_l has access permission $a_l > 2$, and
3. an authenticated message from the user to convey the version q and value v_q to be associated with f_{ij}.

```
$F_{upd}(f_{ij}, \theta_{ij}, \alpha_{ij}, u_l, e_l, a_l, \rho_{lv}, \mu, q, v_q, \theta'_{ij}, \rho_{u1}, \xi', \rho_{lu})\{$
    IF $(e_l < t)$ RETURN; // Expired key
    IF $(a_l < 2)$ RETURN; // No authority
    IF $(\rho_{lv} \neq h(LV, u_l, a_l, \alpha_{ij}, \chi))$ RETURN; //Bad Input
    $K_l := h(\chi, h(u_l, e_l))$;
    $v_{ij} = h(\theta_{ij}, \alpha_{ij}); v'_{ij} = h(\theta'_{ij}, \alpha_{ij})$;
    IF $(\mu \neq h(f_{ij}, v_{ij}, v_q, K_l))$ RETURN; //Incorrect MAC
    IF $(\rho_{u1} \neq h(U1, H_L(q,1,0), \theta_{ij}, H_L(q,1,v_q), \theta'_{ij}, \chi))$ RETURN;
    IF $(\rho_{lu} \neq h(LU, f_{ij}, v_{ij}, \xi, v'_{ij}, \xi', \chi))$ RETURN//Bad Input
    $\xi := \xi'$; //Change state
    RETURN $t, \mu_t = h(ACK, f_{ij}, q, v_q, t, K_l)$;
}
```

The module expects θ_{ij} to be consistent with the existence of a place-holder $(q, 1, 0)$ to accommodate the new version. Specifically, as the version tree leaf should be modified from $(q, 1, 0)$ to $(q, 1, v_q)$ the module expects inputs θ'_{ij} and ρ_{u1} satisfying

$$\rho_{u1} = h(U1, H_L(q,1,0), \theta_{ij}, H_L(q,1,v_q), \theta'_{ij}, \chi). \tag{4.20}$$

Furthermore, as modifying θ_{ij} to θ'_{ij} necessitates changing the value $v_{ij} = h(\theta_{ij}, \alpha_{ij})$ associated with f_{ij} to $v'_{ij} = h(\theta'_{ij}, \alpha_{ij})$, the module expects inputs ξ' and ρ_{lu} satisfying

$\rho_{lu} = h(LU, f_{ij}, v_{ij}, \xi, v'_{ij}, \xi', \chi)$.. After updating the state to ξ' the module outputs an authenticated ACK for the user u_l.

4.4 Security Protocol

The security protocol for the remote file storage system can be seen as the sequence of steps to be taken by users and the file storage server **U** for modifying the state for purposes of creating a new file, updating a file or updating the ACL.

4.4.1 Protocol for Creating a New File

The creator (say u_i) of the first version of a file

1. assigns a unique label l_j;
2. creates an ACL as a list of two tuples $(x_1, a_1) \cdots (x_l, a_l)$ arranged in the order of increasing user identities $x_1 < x_2 < \cdots < x_l$;
3. constructs an IOMT with l leaves to represent the ACL; let the root of the IOMT be α_{ij}.
4. chooses a file encryption secret s_1, and encrypts the file, and computes the hash λ_1 of the encrypted file;
5. computes

$$\begin{aligned} \bar{s}_1 &= h(K_i, \lambda_1) \oplus s_1 \\ v_1 &= h(\lambda_1, s'_1, w) \text{ where } w = h(u_i, e_i) \\ \theta_{ij} &= h(1, v_1, 1) \\ v &= h(\theta_{ij}, \alpha_{ij}) \end{aligned} \tag{4.21}$$

To instruct that a new file should created, the user u_i sends (to the service **U**) the following values:

$$\begin{aligned} u_i \rightarrow \mathbf{U} \quad &: \quad \{u_i, e_i, l_j, \lambda_1, \bar{s}_1, \mu = h(f_{ij}, v, K_i)\} \\ &+ \ \text{Encrypted file and ACL.} \end{aligned} \tag{4.22}$$

If no place-holder exists for f_{ij} the server creates a place-holder. Specifically, only if no leaf for index f_{ij} exists can the server obtain an equivalent certificate binding the current root ξ and the root ξ' (after insertion of place-holder).

The service provider $\mathbf{U}$ then invokes interface $F_{in}()$ to obtain an authenticated acknowledgement. and send values $\mathbf{r} = \{\mu_r, t\}$ as proof to the user u_i that the file has been initialized.

Assurances and Rationale: On receipt of the authenticated acknowledgement the user is assured that value v_{ij} has been bound to leaf f_{ij}. Specifically,

1. as K_i is privy only to the user u_i and the module, and $h()$ is pre-image resistant, only the module could have generated μ_r;
2. $\mathbf{U}$ can invoke $F_{in}()$ only if a place-holder exists for f_{ij}; this assures the user that even if the user had accidentally requested the server to create a file with the same label l_j (and hence the same index $f_{ij} = h(u_i, l_j)$ corresponding to an existing file) such a request can *not* be entertained by the server.

4.4.2 Creating a New Version

Before a user sends a new version of a file, or modifies the ACL for a file, the user u_l should request the server to provide information regarding the current state of the file (which the server can do by using function $F_{rep}()$).

If the user has access permission $a_l > 1$, to supply a newer version (say version q) of the file the user u_l creates the updated version, chooses an encryption secret s_q, encrypts the file, computes the hash λ_q of the encrypted file. The user the computes

$$\begin{aligned} \bar{s}_q &= h(K_l, \lambda_q) \oplus s_q \\ v_q &= h(\lambda_q, \bar{s}_q, w) \text{ where } w = h(u_l, e_l) \\ \mu &= h(f_{ij}, v_{ij}, q, v_q, K_l) \end{aligned} \tag{4.23}$$

The user sends the encrypted file, along with values $\bar{s}_q, u_l, e_l$ and μ to the service $\mathbf{U}$.

$\mathbf{U}$ is required to insert a place holder in the version tree with root θ_{ij} before it can invoke module function $F_{upd}()$. The output $\{\mu_r, t\}$ of $F_{upd}()$ is returned as proof to the user u_l that the new version of f_{ij} has been incorporated.

Assurances and Rationale: On receipt of the authenticated acknowledgement for a request to insert a new version the user is assured that a new version q is the highest version number, and is bound to value v_q. Specifically,

1. as K_l is privy only to the user u_l and the module, and $h()$ is pre-image resistant, only the module could have generated μ_r;
2. the value v_q can be bound only to a place holder with next index 1. Unless q is the highest version, such a place-holder $(q, 1, 0)$ can *not* exist in the file version IOMT.

The server is however free to insert any number of place-holders in the version tree. Consider a scenario where when version 6 of file is the highest version, the server inserts 2 place holders — say for versions 7 and 8. Now, when the server receives an update message, the new version may be incorporated as version 8.

However, if the server skips versions (6 to 8) in this manner, then there would remain a place holder $(7, 8, v_7 = 0)$ for version 7, which can never be bound to meaningful values. More specifically, the server would not be able to respond to a query for information regarding version 7 as the function F_{rep} requires inputs $\lambda_7, \bar{s}_7$ and w_7 satisfying $h(\lambda_7, \bar{s}_7, w_7) = v_7 = 0$. Thus, the only recourse for the server would be to remove the place holder for version 7 which will result in version 6 pointing to version 8 as the next version. Only then can the server create a report for version 6 indicating version 8 as the next version (thereby demonstrating that version 7 does not exist).

4.4.3 Modifying ACL

To supply a newer ACL for a file f_{ij} a user u_l (in the ACL with $a_l > 2$)

1. creates the new ACL, computes α'_{ij};
2. computes $\mu = h(f_{ij}, v_{ij}, \alpha'_{ij}, K_l)$;
3. submits the new ACL and MAC μ to the server.

The server employs an interface $F_{acl}()$ exposed by $\mathbf{T}$ to obtain an authenticated acknowledgement from the module to the effect that "the ACL for f_{ij} has been updated."

Assurances and Rationale: On receipt of the authenticated acknowledgement the user is assured that the ACL root has been modified to α' in the leaf with index f_{ij}. Specifically, as K_l is privy only to the user u_l and the module, and $h()$ is pre-image resistant, only the module could have generated μ_r

Binding the current v_{ij} to the user's request for changing the ACL or updating a file serves two purposes.

For purposes of updating the version, it prevents two users — say u_l and u_k from providing two different next updates to the current version. For example, without this, the user u_l may create a version $q+1$ as the next version of a file f_{ij}^q, and the user u_k will end up creating version $q+2$ as the next version of a file f_{ij}^q. Furthermore, if the requests from the two users come close together, the untrusted server will then have the ability to decide which of the two versions should be $q+1$ (and which should be version $q+2$).

For purposes of updating the ACL including the current state v_{ij} in the request ensures that the server cannot replay the user's MAC to the function F_{acl}. Consider a scenario where after the request from u_l resulting in an ACL root update to α_l, a user u_k updates the ACL to another value α_k. If v_{ij} is not included in the computation of the the MAC by the user u_l, then the MAC from u_l could be replayed to set the ACL back to α_l. Including value v_{ij} prevents such a possibility, as the replayed MAC will not be consistent with the current file state v_{ij}.

4.4.4 Swapping IOMT Leaves

In practical file storage services the total number of files may be or the order of several billions. However, out of the billions, only a small fraction (say, thousands) may be *currently* under frequent modification. From the perspective of reducing computational overhead it is advantageous to group such currently active files close together in the tree. Specifically, a small sub-tree, and values required to map the sub-tree to the main root ξ could then be stored in faster cache memory to speed up operation of the server.

The equivalence feature in IOMT (described in Section 3.2) — which allows the untrusted server to swap leaves — can be very useful in practice to reduce the overhead for the server. Using this feature the physical position of two leaves — a frequently used one and a one that has not been modified for a long time (and thus is not expected to be used in the near future) — can be swapped. From the perspective of the module, a swap can be performed by providing a root equivalence certificate.

4.5 Related Work

In the file storage model used in [59] users create files for access by themselves from other locations. The goal is to ensure that that only the latest version will be provided by the service (older versions should not be replayed). Specifically, a user may update a file using her computer at work. When she later tries to retrieve the file from her home, she expects the latest version of the file. The assumption here is that the user may not actually remember the previous updates she had made, and thus may not recognize the freshness of the version provided by the server. As the user does not trust the server, the user expects the trusted module to attest the freshness of the file. Specifically, the module will only attest the file hash of the freshest version of any file. No attempt is made in [59] to provide assurances of privacy, or enforce any type of access control policy.

In [59] the trusted module stores the root of a Merkle tree, where the leaves of the tree are *virtual counters* — one associated with every file. The virtual counter associated with a file will be incremented by the module only when the file is updated through an authenticated message from the owner of the file. The hash of the new version of the file is bound to the incremented virtual counter value.

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

There is however a simple attack against this approach. The hole in this mechanism [42] is that there is no way to prevent the untrusted server from creating *multiple virtual counters* for the same file. Thus, when an update is received, the server can request the module to update one such leaf. However, the older version — bound to another leaf — still remains legitimate from the perspective of the module, and can thus be replayed.

The use of IOMT instead of a plain Merkle tree ensures that only one leaf can exist for a file (as the IOMT is indexed by file identifier f_{ij}), and thus prevents such attacks. In addition, the enhanced capability of the IOMT to permit verification of non existence can be leveraged to enforce sophisticated access control paradigms, and cater for authenticated denial without revealing unsolicited information.

CHAPTER 5

A GENERIC CONTENT DISTRIBUTION SYSTEM

A content distribution system includes *publishers* who create content for consumption by *subscribers*, and a third party *distribution infrastructure* as publishers themselves may not possess the infrastructural capabilities required to distribute content. In any such content distribution system (CDS), publishers desire mechanisms to ensure that their content is made available only to a select set of subscribers, by prescribing an access control list (ACL) for the content. Subscribers desire mechanisms to ensure the integrity and authenticity of the content.

Irrespective of the nature of the specifics of the CDS, the users of the system — viz., publishers and subscribers — are expected to trust the infrastructural elements to preserve integrity of content, and adhere to the (content specific) ACL prescribed by the publisher. In practice, the "infrastructure" may be composed of possibly several agencies, computers and personnel. As malicious behavior by any infrastructural component may lead to violation of the desired assurances, and as it is impractical to rule out such behavior in complex systems, explicit mechanisms are required to assure the operation of such infrastructural elements.

5.1 A Model for a Generic Content Distribution System

A content distribution system consists of a dynamic set of user subsystems $\mathcal{U} = \{u_1 \cdots u_n\}$ (who may be publishers, or subscribers, or both) and a dynamic set of content $\mathcal{C} = \{c_1 \cdots c_m\}$, where u_i is a unique identity of a user, and c_i is a unique label assigned to a content. Both sets $\mathcal{U}$ and $\mathcal{C}$ may be dynamic, and posses practically unlimited cardinality (unlimited n and m).

Associated with a content with label c_j are

1) a user of the system u_i identified as the publisher/owner of the content;

2) a content encryption secret s_j, and

3) an access control list $\mathbf{A}_j$ (created by the owner);

4) a cryptographic hash γ_j of the encrypted content.

As long as a mechanism exists to securely deliver the content hash γ_j to a user, irrespective of the channel over which the actual content is delivered, the user receiving content c_j can verify the integrity of the content. For example, such encrypted content could be made available for download from a public repository or even distributed across several peer-to-peer clients. However, to gain clear access to the content, a user u requires the content encryption secret s_j. Only privileged users specified in the access control list (ACL) should be able to do so.

More generally, the ACL could assign various levels of privileges; for example, some users may be allowed only to access the content (more specifically, to receive secret s_j); some users with a higher privilege may be allowed to modify the content; some users with an even higher privilege may even be allowed to modify the ACL (in other words, the ACL may also be dynamic).

The infrastructural elements $\mathcal{I}$ in a CDS include mechanisms required to physically host content and the ACL associated with the content, accept requests from users, and deliver encrypted content and content decryption keys to privileged users.

5.1.1 Desired Assurances

Ideally, publishers (subsystems) should need to interact with the CDS infrastructure $\mathcal{I}$ (central subsystem) only for uploading their content, or for modifying the ACL. Some of the specific desired assurances regarding the operation of the CDS are as follows:

1. $\mathcal{I}$ will not alter the content; more specifically, $\mathcal{I}$ will ensure that only users explicitly granted the permission (by the owner) to modify the content can do so.
2. $\mathcal{I}$ will not reveal content encryption secrets to unauthorized parties;
3. $\mathcal{I}$ will not modify the ACL; more specifically, $\mathcal{I}$ will ensure that only users explicitly granted the permission (by the owner) to modify the ACL can do so.
4. a user u authorized (as per the most recent ACL) to receive content will receive only most up-to-date version of the content;
5. when a user u requires access to content c_j, and if u *is* authorized access to the content, $\mathcal{I}$ will *not* refuse to provide access to the content.
6. when a user u requests access to content c_j, and if the content c_j does not exist, the user will learn nothing about the existence of other contents that have not been explicitly queried by the user; similarly, if u is *not* authorized access to the content, the user will learn nothing about other users who have access to the content.

Broadly, the first assurance is towards *authentication and integrity* of content. The second and third assurances are necessary to guarantee *privacy* of the content as intended by the creator. The fourth assurance addresses replay attacks — after a content has been modified, the older version may not be replayed by the CDS infrastructure; similarly, after an ACL has been modified, the old ACL should not be used to distribute content.

The fifth assurance is towards *authenticated denial* to prevent improper denial of service to a subscriber with a legitimate request. For this purpose, the infrastructure is expected to respond to *every* query. The response should either provide the requested content, or should contain a justification to convince the user that the requested content cannot be provided.

The sixth assurance is required to address some of the *undesirable side effects of providing authenticated denial*. In providing the proof of denial no information that was not explicitly queried should be provided.[1]

5.2 Overview of Proposed Approach

In the proposed model, the infrastructure $\mathcal{I}$ is the untrusted participant/subsystem $\mathbf{U}$, and has access to a trusted module $\mathbf{T}$ which serves as the TCB for $\mathbf{U}$. As every component of $\mathbf{U}$ is untrusted the nature of the specific components of $\mathbf{U}$ (like personnel, computers and software) is irrelevant for our purposes. Some component of the $\mathbf{U}$ communicates with the module $\mathbf{T}$ using fixed interfaces exposed by the module. For example, the module $\mathbf{T}$ could be plugged into a computer in $\mathbf{U}$. Alternately, the module $\mathbf{T}$ could be housed in a secure location and connected over a (possibly untrusted) network to a computer in $\mathbf{U}$.

[1]Ignoring such assurances in practical applications have sometimes resulted in attacks that undermine the utility of the protected system. For example, in DNSSEC [6], the security protocol for assuring the operation of a domain name system (DNS) server the unsolicited DNS records obtained from querying non existent records result in the undesirable "DNS walk" problem [33].

The module $\mathbf{T}$ is assumed to be capable of computing a shared secret with any user. A detailed description of key establishment algorithms is given in Section 3.1. For the purposes of this model, we shall simply represent by K_i the secret common to a user u_i and the module $\mathbf{T}$. Such secrets are employed by users and the module $\mathbf{T}$ to authenticate requests and responses using message authentication codes (MAC), and for securely conveying content encryption secrets. The module is also assumed to be capable of executing some simple functions $F_{adl}(), F_{inc}(), F_{upd}()$, $F_{cac}()$, and $F_{snd}()$ which are described algorithmically later in this chapter.

In the proposed approach all desired assurances enumerated in Section 5.1.1 are guaranteed to the extent we can trust the integrity of the module. Specifically, that the module is read-proof implies that only a user u_j and the module have access to the secret K_j, and thus impersonation of messages is infeasible. That the module is write-proof implies that the functionality of the module is immutable.

5.2.1 TCB Functions

In the proposed model TCB function $F_{inc}()$ is used for making content available for distribution. The inputs to $F_{inc}()$ are various parameters (like content hash, encryption secret, ACL, etc.) associated with the content c_j, and are authenticated by the owner o_i of the content for verification by $\mathbf{T}$ using a MAC computed using the secret K_i. After executing $F_{inc}()$ the module outputs an acknowledgment message for verification by o_i.

The TCB function $F_{upd}()$ is used to modify the content or modify the ACL associated with the content. Inputs of the function $F_{upd}()$ are authenticated by a user u_a who is (authorized to modify the content/ACL) using the secret K_a shared with the module. The output of the module is an authenticated acknowledgment. If the user is not authorized, the module will respond with an acknowledgement for a message indicating failure to carry out the request.

The input to $F_{snd}()$ is an authenticated request from a user u_q for some content c_j. Only if the user is authorized to receive the content will the module convey the content encryption secret s_j and γ_j (to verify the integrity of the content) to user u_q. If the user is not authorized, the module will respond with an acknowledgement for a message indicating failure to carry out the request.

In all exchanges between users and the module, $\mathbf{U}$ is an untrusted middle man. The middle man $\mathbf{U}$ is expected to faithfully perform some tasks in order to provide additional inputs $\mathbf{v}$ required for the TCB functions, and receive an authenticated acknowledgement from the module, which can then be delivered to the user. More specifically, if $\mathbf{U}$ does not execute such tasks faithfully, then $\mathbf{U}$ will not be able to obtain an authenticated response from the module to satisfy the user.

Unlike the three functions above the inputs to $F_{adl}()$ and $F_{cac}()$ do not include authenticated requests from users. $F_{cac}()$ is employed by $\mathbf{U}$ to request the module to verify access control permission for a user, and generate a certificate vouching for the same. As this certificate is intended for verification by the module (which issues the certificate) at a later time, the self-certificate is simply a MAC computed using a secret known only to the module $\mathbf{T}$.

Interface $F_{adl}()$ is used by $\mathbf{U}$ to request the module to insert or delete a leaf in an Index Ordered Merkle Tree (IOMT). The IOMT, (which is explained in greater depth in Section 3.2) is a simple extension of the better known (plain) Merkle hash tree [38]. The main differences between a "plain" Merkle tree and an IOMT are a) some additional rules to be observed in the IOMT for *inserting* and *deleting* leaves - to ensure uniqueness of indexes, and b) the ability to efficiently deal with any number of leaves - even if the number of leaves is not a power of 2.

In the proposed approach each leaf of the IOMT corresponds to a content. Associated with a set of m leaves (where m is the total number of content identifiers) are $2m-1$ hashes which are the "internal nodes" of the tree. The number m is assumed to be dynamic — m grows as content with new labels are introduced into the system (a leaf is inserted into the IOMT) for distribution and may reduce (an IOMT leaf is deleted) as their circulation is cut off by the owner or an entity authorized by the owner. The IOMT is also used to efficiently represent the ACL associated with each content.

Untrusted $\mathbf{U}$ stores all m contents, $2m - 1$ hashes, and some values associated with each content (leaf). Recall that such values associated with a content c_j include a) content owner u_i, b) content hash γ_j, c) ACL $\mathbf{A}_j$ for the content, and d) an *encrypted* version of the content encryption secret s_j.

The module $\mathbf{T}$ stores only[2] i) a single hash - the root ξ of the IOMT; and ii) a secret χ (known only to the module) used for encrypting the content encryption key.

[2]In addition, the module may be required to store one or more secrets that enable the module to compute pairwise secrets.

Computationally, the module performs simple sequences of hash operations to execute functions $F_{adl}(), F_{inc}(), F_{upd}()$, $F_{cac}()$, and $F_{snd}()$, for which the module requires temporary scratch-pad memory for a mere $\mathbb{O}(\log_2 m)$ hashes (at most a few kB).

5.2.2 Access Control List

The access control list $\mathbf{A}_i$ can be seen as a list of two tuples of the form (o_i, a_i) where o_i is the identity of a user assigned privilege a_i. We shall assume that $a_i = 0$ implies that the user o_i is *not* granted access, and $a_i = 1$ to imply that the user o_i is granted access. We shall also assume an empty access control list to imply that the content cannot be distributed.

In such a list $\mathbf{A}_j = \{(o_1, a_1), (o_2, a_2), \ldots, (o_k, a_k)\}$ associated with some content c_j, and ordered by ascending order of o (or $o_1 < o_2 < \cdots < o_k$), that a tuple for o_{i+1} follows o_i (in the list $\ldots, (o_i, a_i), (o_{i+1}, a_{i+1}), \ldots$) implies that no tuple exists for an identity that falls between o_i and o_{i+1}. In such a case it is more meaningful to see the ACL as a three tuple of the form $\{(o_1, o_2, a_1), (o_2, o_3, a_2), \ldots (o_{n-1}, o_n, a_{n-1}), (o_n, o_1, a_n)\}$.

For example, in a ACL

$$\mathbf{A} = \{(o_1, o_2, 1), (o_2, o_3, 0), (o_3, o_4, 1), (o_4, o_1, 1)\} \tag{5.1}$$

1) the implication of the first tuple $(o_1, o_2, 1)$ is that o_1 is explicitly granted access, and that all users $o_1 < x < o_2$ are denied access;

2) from the second tuple, o_2 is explicitly denied access; all users $o_2 < x < o_3$ are granted access;

3) o_3 is explicitly granted access; all users $o_3 < x < o_4$ are denied access;

4) o_4 is explicitly granted access; as $(o_1 < o_4)$, users $x > o_4$, and users $x < a_1$ are denied access;

In some scenarios the privileged users may have different types of privileges. For example, we shall assume that $a_i = 2$ implies that the user o_i is allowed to modify the content, and $a_i = 3$ implies that o_i is permitted to modify the content, and the ACL for the content. For example, if $\mathbf{A}_j = \{(o_1, o_2, 0), (o_2, o_3, 3), (o_3, o_4, 1), (o_4, o_5, 0), (o_5, o_1, 2)\}$,

1) o_1 is denied access; all users $o_1 < x < o_2$ are provided regular access;

2) o_2 is granted enhanced privilege to modify the ACL; all users $o_2 < x < o_3$ are denied access;

3) o_3 is provided regular access; all users $o_3 < x < o_4$ are denied access;

4) o_4 is denied access; all users $o_4 < x < o_5$ are granted regular access;

5) o_5 is explicitly granted privilege to modify the content; users $x > o_5$ and $x < a_1$ are denied access;

In the proposed approach each tuple in an ACL is seen as a leaf of an IOMT. The root α of such an IOMT is a succinct representation of the ACL. Specifically, given the value α, a set of hashes $\mathbf{v}$ and an IOMT leaf (u_r, u'_r, a_r) the module can verify the leaf against the root α and infer the access control restrictions associated with user $u_q = u_r$ or any user u_q covered by (u_r, u'_r).

5.2.3 IOMT for Storing Content Leaves

An IOMT leaf (as shown in Figure 5.1) for storing values associated with a content c_j takes the form (c_j, c'_j, v_j) where c'_j is the next label and $v_j = h(u_i, \gamma_j, s^s_j, \alpha_j)$ is a one way function of the owner u_i, content hash γ_j, s_j^s (the encrypted version of content encryption secret s_j) and α_j (the succinct representation of the ACL associated with the content c_j). The root ξ of this IOMT changes whenever a content is introduced, or deleted, or if the ACL for a content is modified. The dynamic root ξ of this IOMT is maintained inside the module.

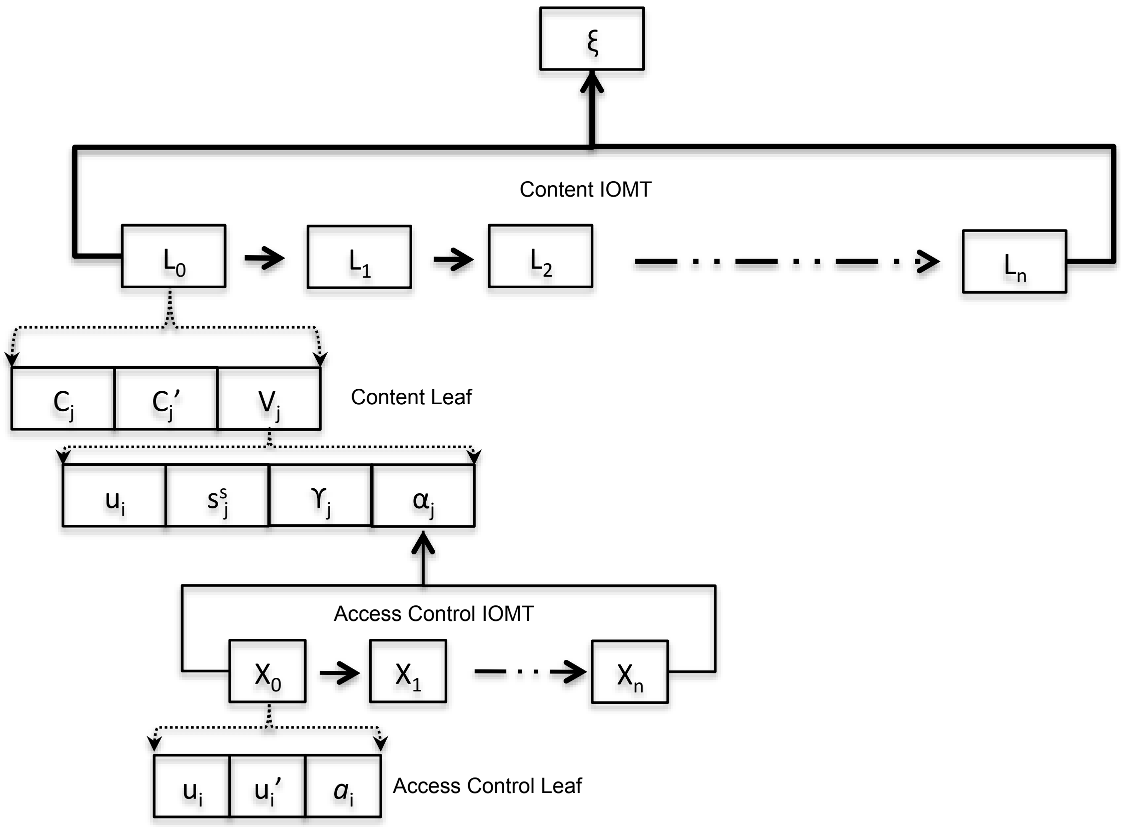


Figure 5.1

Data-Structure for Content Distribution. [3]

[3]Figure Notes: α_j is the root of ACL tree with leaves as (u_i - user, u_i' - next user, a_i - access type).

Given values $c_j, c'_j, u_i, \gamma_j, s^s_j, \alpha_j$ along with a set of hashes $\mathbf{v}$ the module $\mathbf{T}$ can verify the integrity of all values associated with the content c_j (by computing $v = H_L(c_j, c'_j, v_j = h(u_i, \gamma_j, s^s_j, \alpha_j))$, and then verifying that $f(v, \mathbf{v}) = \xi$.

A leaf with middle value zero, say $(c_j, c'_j, 0)$ is a "place-holder" for a content and can be used to reserve a content label. After content specific values are received for the content from the owner, such values are bound to the leaf by appropriately setting value v_j and updating the root ξ.

Unlike the IOMT for ACL which is prepared at one go by the owner or an authorized agent, various leaves of the content IOMT are provided by different content owners. Thus, for maintaining such an IOMT the module needs the ability to insert and delete IOMT leaves.

5.3 TCB Functions

The resource limited module $\mathbf{T}$ securely stores a secret χ, spontaneously generated inside the module, and the root ξ of an IOMT. It is assumed that the module can compute a secret K_i it shares with any user u_i.

5.3.1 Secret χ

This secret χ (spontaneously generated inside the module) is used by the module to encrypt content secrets entrusted to the module. Specifically, for a content c_j associated with a owner u_i, the content encryption secret s_j is conveyed by the owner u_i to the module as $s'_j = h(K_i, \mu_{ij}) \oplus s_j$, where $\mu_{ij} = h(c_j, \gamma_j, \alpha_j, s_j, K_i)$ is a message authentication code (MAC) used to securely convey the content related values to the module $\mathbf{T}$. The secret s_j

is then re-encrypted by the module as $s_j^s = h(\chi, c_j, \gamma_j) \oplus s_j$ and handed back to untrusted $\mathbf{U}$ for storage.

The secret χ is also used by the module to generate *self-certificates* for verification by itself at a later time. Specifically, using the function $F_{cac}()$ the module can be requested to issue a certificate of the form $\mu_s = h(u, a, \alpha, \chi)$ which states that "a user with identity u and access control permission a is *consistent* with access control digest α. The function $F_{cac}()$ can be described algorithmically as follows:

```
$F_{cac}((u_r, u'_r, a_r), \mathbf{v}, u_q)\{$
IF $(u_q = u_r)$ $a_q := a_r;$
ELSE IF$(cov(u_q, (u_r, u'_r)))$ $a_q := (a_r = 0)?1 : 0$
ELSE RETURN;
$\alpha = f(H_L(u_r, u'_r, a_r), \mathbf{v});$
RETURN $\mu_s := h(u_q, a_q, \alpha, \chi);$
$\}$
```

The inputs to $F_{cac}()$ include a leaf of a ACL IOMT with root α along with the hashes $\mathbf{v}$ necessary to verify the leaf against α. The function $F_{cac}()$ will output a certificate only if $u_q = u_r$ or if u_q is covered by (u_r, u'_r). The values u_q, a_q, α and μ_s satisfying $\mu_s = h(u_q, a_q, \alpha, \chi)$ can be provided to the module at any later time to convince the module that "for a content with ACL digest α, a user u_q has access restriction a_q."

5.3.2 Addition and Deletion of IOMT Leaves

In general, to add or delete an IOMT leaf two leaves need to be updated. 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_p^l and v_p^r, and let $v_p = H_V(v_p^l, v_p^r)$ (v_p^l and v_p^r are the left and right child of v_p), $v_p^l = f(v_l, \mathbf{v}_l)$, $v_p^r = f(v_r, \mathbf{v}_r)$, and $r = f(v_p, \mathbf{v}_p)$. Now,

$$\xi = f(H_V(f(v_l, \mathbf{v}_l), f(v_r, \mathbf{v}_r)), \mathbf{v}_p) \tag{5.2}$$

The interface $F_{adl}()$ can be used to insert a leaf for an index a (if a is covered by another leaf), or deleting a leaf with index a (if another leaf exists that points to a). Specifically, when a leaf is inserted the middle value is set to 0. Only leaves with middle value 0 can be deleted. Henceforth we shall refer to an IOMT leaf of the form $(a, a', 0)$ as a "place-holder."

The module functionality $F_{adl}()$ for inserting or deleting a place holder can be algorithmically described as follows:

```
$\xi = F_{adl}((l, l', v_l), (r, r', v_r), i, \mathbf{v}_l, \mathbf{v}_r, \mathbf{v}_p)\{$
IF $(l = 0) \wedge (r = 0)$//Insertion of first leaf
    $h_l := 0; h'_l := H_L(i, i, 0); h_r := 0; h'_r := 0;$//
ELSE IF $(l = 0) \wedge cov(i, (r, r'))$
    $h_l := 0; h_r := H_L(r, r', v_r); h'_l := H_L(i, r', 0); h'_r := H_L(r, i, v_r);$
ELSE IF $(r = 0) \wedge cov(i, (l, l'))$
    $h_r := 0; h_l := H_L(l, l', v_l); h'_r := H_L(i, l', 0); h'_l := H_L(l, i, v_l);$
ELSE RETURN;
    $\xi_1 = f(H_V(f(h_l, \mathbf{v}_l), f(h_r, \mathbf{v}_r)), \mathbf{v}_p);$//Root before insertion
    $\xi_2 = f(H_V(f(h'_l, \mathbf{v}_l), f(h'_r, \mathbf{v}_r)), \mathbf{v}_p);$//Root after insertion
IF $(\xi = \xi_1)$ $\xi := \xi_2;$// insert index $i$
IF $(\xi = \xi_2)$ $\xi := \xi_1;$//delete index $i$
RETURN $\xi$;
}
```

U invokes this function whenever a new content is created or when distribution of an existing content has to be stopped, or when queried for an non existent content. Specifically,

1. if a new content with unique label c_j is made available for distribution, a place holder for index c_j is inserted to reserve a leaf for c_j (after this function $F_{upd}()$ will be used to bind the content related values to the middle value of the leaf).
2. if a query for a content c_J is made and no content exists for index c_j, a place holder for index c_j is inserted; soon after the query is answered the place-holder may be deleted.
3. The function $F_{uac}()$ (to update ACL) is employed to convert a leaf to a place-holder (by setting the middle value to 0) for halting the distribution of the content. The place holder can be deleted if required using $F_{adl}()$

To insert a place holder, two leaves - an empty leaf, and a covering leaf (j, j', v_j) (or $cov((j, j'), i)$ is TRUE) are provided as inputs to $F_{adl}()$. 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 insertion 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 inserting index i, by setting the root to ξ_2 a leaf with index i is inserted. 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 *deleted*. Thus, even while $F_{adl}()$ only verifies the pre-requisites for inserting a place-holder, $F_{adl}()$ can be used for both insertion and deletion of place holders.

5.3.3 Distribution of Content

The owner u_i of the content c_j performs the following steps to make the content available to users:

1) assign a unique label c_j to the content;

2) choose a random content encryption secret s_j and encrypt content;

3) compute hash γ_j of the encrypted content;

4) compute root α_j of an ACL IOMT $\mathbf{A}_j$;

5) submit to $\mathbf{U}$, i) the encrypted content, ii) access control list $\mathbf{A}_j$, iii) γ_j and iv) values μ_{ij}, s'_j where

$$\mu_{ij} = h(c_j, \gamma_j, \alpha_j, s_j, K_i), s'_j = h(K_i, \mu_{ij}) \oplus s_j, \tag{5.3}$$

and K_i is a secret shared between user u_i and the module $\mathbf{T}$; $\gamma'_j = 0$ implies that this is the first version of the content.

On receipt of such a message from user u_i, $\mathbf{U}$ performs the following steps:

1) reserve label c_j for user u_i; for this purpose $\mathbf{U}$ employs the interface $F_{adl}()$ exposed by the module $\mathbf{T}$. At the end of this process a place holder $(c_j, c'_j, 0)$ consistent with the IOMT root ξ will be available satisfying $f(H_L(c_j, c'_j, 0), \mathbf{v}_j) = \xi$.

2) evaluate α_j using the ACL;

3) employ module interface $F_{inc}()$ to supply values that include u_i, α_j, γ_j, s'_j, μ_{ij}, provided by the user and values c_j, c'_j and $\mathbf{v}_j$ associated withe the corresponding IOMT leaf. On completion of the execution of $F_{inc}()$ the module will outputs an acknowledgement MAC

$$\mu'_{ij} = h(ACK, \mu_{ij}, K_i) \tag{5.4}$$

The function $F_{inc}()$ can be described algorithmically as follows:

```
$F_{inc}(c_j, c'_j, \mathbf{v}_j, u_i, \gamma_j, \alpha_j, s'_j, \mu_{ij})\{$
IF $f(H_L(c_j, c'_j, 0), \mathbf{v}_j) \neq \xi)$ RETURN;
$s_j = h(K_i, \mu_{ij}) \oplus s'_j;$
IF $(h(c_j, \gamma_j, \alpha_j, s_j, K_i) \neq \mu_{ij})$ RETURN;
$s^s_j := s_j \oplus h(\chi, c_j, \gamma_j); v_j := h(u_i, \gamma_j, s^s_j, \alpha_j);$
$\xi := f(H_L(c_j, c'_j, v_j), \mathbf{v});$
RETURN $\xi, s^s_j, \mu'_{ij} = h(ACK, \mu_{ij}, K_i);$
}
```

4) store s^s_j, and make appropriate modifications to the parent nodes of the updated IOMT leaf to be consistent with the new root ξ;

5) send the acknowledgment MAC μ'_{ij} to the user u_i.

On receipt of the acknowledgment the owner u_i is assured (to the extent the owner can trust the module $\mathbf{T}$) that all the expectations of the owner with regards to distribution of the content will be met by $\mathbf{U}$. More specifically, this trust is based on the premise that i) it is infeasible for any entity except the module $\mathbf{T}$ and user u_i to compute the MAC, and that ii) the module functionality cannot modified.

5.3.4 Updating Content and/or ACL

To update the content c_j, a user u_a authorized to do so encrypts the modified content with a new key s_{ju} and computes the hash γ_{ju} of the updated content. The user then submits

$$\mu_{aj} = h(c_j, \gamma_j, \alpha_j, \gamma_{ju}, \alpha_{ju}, s_{ju}, K_a), s'_{ju} = h(K_a, \mu_{aj}) \oplus s_{ju},$$

to $\mathbf{U}$.

On receipt of the request to update content $\mathbf{U}$ employs interface $F_{cac}()$ to receive a certificate attesting the access control permission for u_a. Then, $\mathbf{U}$ employs interface $F_{upd}()$ shown below to update the IOMT leaf for c_j.

```
$F_{upd}(c_j, c'_j, u_i, \gamma_j, s^s_j, \alpha_j, \mathbf{v}_j, u_a, a_a, \gamma_{ju}, \alpha_{ju}, s'_{ju}, \mu_{aj}, \mu_s)\{$
IF ($\mu_s \neq h(u_a, a_a, \alpha_j, \chi)$) RETURN; //Invalid certificate
$s_{ju} = s'_{ju} \oplus h(K_a, \mu_{aj})$;
IF ($\mu_{aj} \neq h(c_j, \gamma_j, \alpha_j, \gamma_{ju}, \alpha_{ju}, s_{ju}, K_a)$) RETURN; //Invalid Request
$v_j := h(u_i, \gamma_j, s^s_j, \alpha_j)$;
IF $f(H_L(c_j, c'_j, v_j), \mathbf{v}_j) \neq \xi$) RETURN;
IF ($a_a < 2) \vee ((a_a < 3) \wedge (\alpha_{ju} \neq \alpha_j)$); //user not authorized
  RETURN $\mu'_{aj} = h(ACK, \mu_{aj}, 0, K_a)$;
IF ($\alpha_{ju} = 0) v_{ju} := 0$;
ELSE    $s^s_{ju} := s_{ju} \oplus h(S, c_j, \gamma_{ju}); v_{ju} := h(u_i, \gamma_{ju}, s^s_{ju}, \alpha_{ju})$;
$\xi := f(H_L(c_j, c'_j, v_{ju}), \mathbf{v}_j)$;
RETURN $\xi, s^s_{ju}, \mu'_{aj} = h(ACK, \mu_{aj}, K_a)$;
}
```

If the user is not authorized, this function returns $\mu'_{aj} = h(ACK, \mu_{aj}, 0, K_a)$; if the user is authorized the function returns $\mu'_{aj} = h(ACK, \mu_{aj}, K_a)$ and the encrypted version s^s_{ju} of the new content encryption key s_{ju} for storage by $\mathbf{U}$.

When the user u_a receives the MAC μ'_{aj} the user u_a is convinced that the content has been modified and thus, from this point onwards, the old content hash γ_j cannot be replayed by $\mathbf{U}$.

5.3.5 Querying Content

Any user can send a query for any content. To query a content c_j a user u_q (who shares a a secret K_q with $\mathbf{T}$) computes a MAC

$$\mu_{qj} = h(c_j, \nu, K_q) \tag{5.5}$$

where ν is a random nonce selected by the user. The user sends values u_q, c_j, ν and μ_{qj} to the DC.

If the queried content does not exist, $\mathbf{U}$ inserts a place holder for c_j. If the content exists and the user u_q is authorized access, the ACL associated with the content will possess a tuple of the form $(u_q, u'_q, a_q > 0)$ or $(u_r, u'_r, a_r = 0)$ where (u_r, u'_r) covers u_q. On the other hand, if the user u_q is *not* authorized, the ACL will possess a tuple $(u_q, u'_q, a_q = 0)$ or $(u_r, u'_r, a_r > 0)$ where (u_r, u'_r) covers u_q. In either case, $\mathbf{U}$ can employ function $F_{cac}()$ to obtain a certificate $\mu_s = h(u_q, a_q, \alpha_j, \chi)$ from the module.

This certificate, along with values c_j, ν and μ_{qj} sent by the user are provided to the module using the interface $F_{snd}()$, which can be described algorithmically as follows:

```
$F_{snd}(c_j, c'_j, u_i, \gamma_j, s^s_j, \alpha_j, \mathbf{v}, u_q, a_q, \mu_s, \nu, \mu_{qj})\{$
IF $(\mu_{qj} \neq h(c_j, \nu, K_q))$ RETURN;
$v_j := (u_i = 0)?\, 0 :\ h(u_i, \gamma_j, s^s_j, \alpha_j);$
IF $f(H_L(c_j, c'_j, v_j), \mathbf{v}) \neq \xi)$ RETURN;
IF $(v_j = 0)$ RETURN $\mu'_{qj} := h(c_j, \nu, 0, 0, K_q);$ //Content does not exist
IF $(h(u_q, a_q, \alpha_j, \chi) \neq \mu_s)$ RETURN;
IF $(a_q > 0)$RETURN $\mu'_{qj} = h(c_j, \gamma_j, s_j, \nu, K_q), s'_j = h(K_q, \mu'_{qj}) \oplus s_j;$
ELSE RETURN $\mu'_{qj} = h(c_j, 0, 0, \nu, K_q), 0;$ //No access
}
```

If the user is authorized access, the module returns $\mu'_{qj} = h(c_j, \gamma_j, s_j, \nu, K_q)$ and $s'_j = h(K_q, \mu'_{qj}) \oplus s_j$. When such values are conveyed to the user by $\mathbf{U}$, the user gains access to the content encryption secret $s_j = s'_j \oplus h(K_q, \mu'_{qj})$ and can verify the MAC μ'_{qj}. The user may now fetch the encrypted content from untrusted components of the DC, verify its integrity, and decrypt the content using the secret s_j.

If the user is not authorized, or if the queried content does not exist, the module outputs a MAC $\mu'_{qj} = h(c_j, \nu, K_q)$. When $\mathbf{U}$ relays μ'_{qj} values to the querier, the querier is assured (to the extent the module is trusted) that the content does not exist, or the user does not have access to the content.

5.4 Related Work

Models for content distribution and security mechanisms for such models, have attracted substantial attention in the literature. Simple models that do not cater for dynamic content and dynamic ACL employ broadcast encryption [22, 46]. Specifically, in such systems the ACL associated with a content is a list of users/devices that are allowed access to the content, specified at the time the content is made available for distribution. The content is accompanied by a *message key block* consisting of various encryptions of the content encryption key such that at least one can be decrypted by a privileged device (and none by any revoked device).

More sophisticated models with dynamic content and ACL are generally considered under the umbrella of publish-subscribe systems [21, 66]. In such systems the infrastructure may consist of several servers, typically classified into servers at the provider end, consumer end, and core servers. While several security solutions have been proposed, they rely (to varying degrees) on the trustworthiness of servers in the infrastructure. Specifically in such systems the server is trusted to not replay old content, and not deny service to authorized users. One of the main motivations for the proposed approach stems from the lack of a suitable rationale for trusting complex servers employed by the CDS infrastructure.

CHAPTER 6

SECURITY KERNELS FOR BORDER GATEWAY PROTOCOL

The Internet is an interconnection of autonomous systems (AS) [53], [52]. Each AS owns one or more chunks of the IP address space, where the number of addresses in each chunk is a power of 2. IP chunks are represented using the CIDR (classless inter-domain routing) IP prefix notation. For example, the IP prefix 132.5.6.0/25 represents 2^{32-25} IP addresses for which the first 25 bits are the same as the address 132.5.6.0, viz., addresses 132.5.6.0 to 132.5.6.127. An AS registry assigns AS numbers to AS owners. AS owners may acquire ownership of IP prefixes from an IP registry (through IP registrars, or ISPs).

While each AS may follow any protocol for routing IP packets within their AS, all ASes need to follow a uniform protocol for inter-AS routing. The current inter-AS protocol is the border gateway protocol (BGP), where AS owners employ one or more BGP speakers to advertise reachability information for IP prefixes owned by the AS. Specifically, every BGP speaker recognizes a set of neighboring BGP speakers. Neighbors may belong to the same AS or a different AS. The main responsibility of BGP speakers are

1. originate BGP update messages for prefixes owned by the AS, and convey such originated messages to neighbors of other ASes
2. relay BGP update messages received from neighbors to other neighbors.
3. aggregate destination prefixes (that can be aggregated) for reducing the size of routing tables.

6.1 BGP Protocol

BGP is a path vector protocol. BGP update messages communicated between BGP speakers indicate an AS path vector for a prefix. Specifically, a BGP update message

$$[P_a, (A, B, C, D), W_d] \tag{6.1}$$

from a speaker S_d (belonging to AS D — the last AS in the path) indicates that prefix P_a owned by the first AS A in the path. W_d is the *weight* of the path.

A BGP speaker may receive multiple paths for the same prefix. All such paths are stored by the BGP speaker in the incoming routing information database (RIDB-IN). However only the *best* path for a prefix may be copied to the outgoing database (RIDB-OUT), and advertised to other BGP speakers. Most often a BGP speaker is a component of a router which uses entries in RIDB-OUT (best path for different prefixes) to forward IP packets.

The best path is the one with the maximum weight. Several parameters are used to compute the weight of a BGP path. For simplicity, we restrict ourselves to some of the more important weight parameters, i) pre-path weight; ii) local preference iii) AS path length; and iv) multi-exit descriptor (MED).

The pre-path weight is assigned at time of origination. If two paths for the same prefix have the same pre-path weight, then the the local preference is considered (higher the better). If both pre-path weight and local preference are the same, the AS path length (number of ASes in the path) is considered. The longer the path, the lower the weight. If the path lengths are also the same, then the MED weight is considered (higher the better).

6.1.1 Local Preference and MED

Every BGP speaker recognizes a set of other BGP speakers as neighbors. Every neighbor is associated with two weight parameters — a local preference, and an MED. From the perspective of a speaker S_a

1. L_b is the local preference of S_b implies that for all paths received *from* S_b the local preference component of the weight should be reset to L_b.
2. M_b is the MED of S_b implies that for all paths advertised *to* S_b, the MED component of the weight should be set to M_b.

Local preference and MED weights are assigned only to neighbors that are speakers of foreign ASes.

6.1.2 Processing Received BGP Updates

When a BGP update message is received from a foreign speaker S_b (of AS B) the steps to be taken by a speaker S_a (AS A) are as follows:

1. Increment hop-count;
2. Add own AS A to the path vector;
3. Change local preference to value L_b;
4. Set next hop to S_b
5. Store path in RIDB-IN.

When a path is received from a speaker S'_a belonging to the *same* AS, no component of the weight is changed, and the AS number is not inserted (as it was already inserted by the previous hop). The next hop is set to S'_a.

6.1.3 Relaying and Originating BGP Updates

For relaying a BGP message for a prefix P to a BGP speaker S_b in a foreign AS, the steps to be taken by speaker S_a are

1. Among all paths for the same prefix, choose the path with the highest weight
2. Change the MED component of weight to M_b;
3. Advertise the path with modified weight.

For originating a path (for owned prefixes), the pre-path weight is set, and the MED is set to that of the foreign neighbor. Such originated paths are *not* sent to speakers of the same AS (as paths to IP addresses within the AS are established using an intra-AS protocol). For relaying a BGP update message (for a prefix owned by a foreign AS) to a speaker S_a' of the same AS, simply choose the path with the highest weight and send it without changing the weight.

6.1.4 Policies and Preferences

The choice of BGP speakers for the AS, the prefixes for which a speaker may originate BGP update messages (along with their pre-path weights), neighbors of each speaker, along with their local preference and MED weights, etc., can be seen as policies and preferences specified by the AS owner to influence the weights assigned to BGP paths.

6.1.5 Aggregation of IP Prefixes

One of the major benefits of CIDR prefixes come from the fact that BGP speakers may aggregate prefixes. If two consecutive prefixes A and B (say 126.5.4.0/25 and 126.5.4.128/25) and can be aggregated into a single prefix C (126.5.4.0/24) if the next hop for prefixes A and B is the same. The speaker that performed the aggregation is the originator for the aggregated prefix.

6.2 Security Kernels for BGP subsystems

Section 3.2 outlines generic security kernel functionality for issuing OMT certificates. In this section we consider other subsystem specific security kernel functionality for various BGP subsystems like AS and IP registries, AS owners, and BGP speakers.

For simplicity, we shall assume a single registry for both AS numbers and IP addresses. All security kernel modules have a unique identity. Let $\mathbf{R}$ be the identity of the module associated with the registry. One module is associated with every AS owner. We shall assume the identities of an AS owner module to be the same as the AS number. Each BGP speaker is associated with a module. We shall assume that the identity of BGP speaker modules to be the IP address of the router/BGP speaker. We also assume the existence of module functionality for authentication/verification of messages exchanged between modules. Specifically, we shall represent such functionality as

$$\begin{aligned} \mu &= f_a(X, Y, \{v_1, v_2, \ldots\}) \text{ and} \\ \{0, 1\} &= f_v(X, Y, \{v_1, v_2, \ldots\}, \mu) \end{aligned} \tag{6.2}$$

the process of authentication (by module X, using $f_a()$) and verification (by module Y, using $f_v()$) of a message conveying values $\{v_1, v_2, \ldots\}$, from module X to module Y. Function $f_a()$ outputs a authentication code μ. Function $f_v()$ outputs a binary value (TRUE if authentication μ is consistent, or FALSE).

The identity $\mathbf{R}$ of the registry module is known to all AS owner modules. The registry module $\mathbf{R}$ delegates AS numbers and IP prefixes to AS owner modules. AS owner modules will only accept delegations from $\mathbf{R}$. AS owner modules in turn delegate IP address ranges they own to one or more BGP speaker modules.

Some of the specific desired assurances regarding the operation of BGP are as follows:

1. an AS number cannot have more than one owner; an IP address cannot be owned by one or more ASes. Such assurances should be guaranteed even if the computers employed by the registry have been compromised by an attacker.
2. AS owners can only delegate address ranges owned by the AS to BGP speakers.
3. Notwithstanding the possibility that a router/ BGP speaker may be under the control of an attacker, the following assurances are desired
   (a) The BGP speaker will only be able to create BGP update messages for prefixes delegated by the AS owner
   (b) No BGP update message can be created by violating any of the policies / preferences specified by the AS owner (neighboring speakers, local preference and MED, pre-path weights) or BGP rules (only the path with the best weight can be advertised).
   (c) A speaker will not accept paths which already includes its own AS (to ensure that routing loops cannot be created).
   (d) All BGP speakers will increment the hop count exactly by one.
   (e) A speaker will be able to aggregate only prefixes for which the next hop is the same speaker.

6.2.1 Mutual Authentication for BGP subsystems

The module $\mathbf{T}$ is assumed to possess limited protected storage, and expose well defined interfaces to the associated untrusted subsystem. Such interfaces can be used by an untrusted subsystem (say) A to demonstrate the integrity of databases stored by the subsystem, and request $\mathbf{T}_A$ associated with subsystem A to attest verified records.

For attesting records or contents of records (for verification by other subsystems, or security kernels in other subsystems) every module is assumed to possess a unique identity, and secrets used for authenticating messages. For example, the secret could be a private component of an asymmetric key pair, which is used for signing messages. In this case, the public key of the module is certified by a trusted key distribution center, attesting the integrity of the module. Alternately one or more secrets could be provided by a trusted key distribution center to each module. Only modules that have been verified for integrity and issued such secrets by the trusted key distribution centers will be able to use their secrets to compute a pairwise secret with other modules attested by the KDCs. Such pairwise secrets may be used to compute message authentication codes for attesting the integrity of the contents of a record.

6.2.2 OMTs Used by BGP subsystems

The registry and AS owners maintain an ROMT where each leaf indicates a range of IP addresses, and the third value is the AS number (of the AS that owns the address range).

BGP speakers maintain one ROMT, multiple IOMTs, and a plain Merkle tree. A plain Merkle tree is used to maintain a neighbor table with a static[1] number of records. The ROMT is used maintaining address ranges for which the speaker can originate BGP updates (owned prefixes and aggregated prefixes).

An IOMT is used for maintaining the RIDB-IN database. More specifically a nested IOMT is used where the root corresponds to a tree with leaves whose indexes are desti-

[1]For scenarios involving dynamic databases where records cannot be inserted or deleted (the dynamics come only from modification of records) OMT is an over-kill; a plain Merkle tree is adequate.

nation IP prefixes. Corresponding to each prefix the value (third field) is the hash of two IOMT roots θ and γ. The root θ of the "path tree" has one leaf for every path for the prefix. The root γ of the "weight tree" represents the weights of different paths, and enables the module to readily identify the path with the highest weight. The index of leaves in the path tree is a function of a quantity α that is itself the root of an IOMT. Specifically, the "path vector" IOMT with root α has a leaf corresponding to every AS in the AS path. Representing the AS path in this way makes it possible for the module to recognize that it is already in the path, and thereby prohibit creation of routing loops.

6.3 Registry Module R and AS Owner Modules

The registry module maintains an ROMT root ξ_r, where each leaf indicates ranges of IP addresses, and the AS number of the owner. Unassigned IP chunks have a leaf with (third) value 0.

The function $F^R_{ph}()$ can be utilized to insert/delete any place holders in the ROMT by providing a memoranda of type ER.

The registry employs the function $F^R_{as}()$ to convert the third value of any leaf from 0 to a non zero value.

A leaf (I, I', A) in the ROMT indicates that the IP addresses in the range I and $I' - 1$ have been assigned to AS A. The leaf (I, I', A) can be conveyed to an AS owner module A using interface $F^R_{dp}()$.

AS owner modules also maintain an ROMT with root ξ_r. The leaves indicate IP addresses owned by the AS. In the tree maintained by the owner of AS A who (for example)

```
$F^R_{ph}(x, \rho)$ {
    IF $(\rho = h(ER, \xi_r, x, \chi))$ $\xi_r \leftarrow x$;
}
$F^R_{as}(I, I', A, \rho, \xi'_r)$ {
    IF $(\rho = h(U1, H_L(I, I', 0), \xi_r, H_L(I, I', A), \xi'_r, \chi)$ $\xi_r \leftarrow \xi'_r$;
}
$F^R_{dp}(I, I', A, \rho)$ {
    IF $(\rho = h(V1, H_L(I, I', A), \xi_r, \chi))$ RETURN $f_a(U, A, \{x\})$;
}
$F^O_{ph}(x, \rho)$ {
    IF $(\rho = h(ER, \xi_r, x, \chi))$ $\xi_r \leftarrow x$;
}
$F^O_{ap}(I, I', \mu, \rho, \xi'_r)$ {
    $x \leftarrow H_L(I, I', 0); x' \leftarrow H_L(I, I', A)$;
    IF $(f_v(U, A, x', \mu) = 0)$ RETURN;
    IF $(\rho = h(U1, x, \xi_r, x', \xi'_r, \chi)$ $\xi_r \rightarrow \xi'_r$;
}
$F^O_{dp}(\xi'_o, \rho, S, \xi'_n)$ {
    IF $(\rho = h(V1, \xi'_o, \xi_r, \chi)$ RETURN $\mu = f_a(A, S, \{\xi'_o, \xi'_n\})$;
}
```

Figure 6.1

Security Kernel Functionality in Registry and AS Owner Modules.

owns two non consecutive chunks with addresses between $[a, a')$ and $[b, b')$ the ROMT leaves will be (a, a', A), $(a', b, 0)$, (b, b', A) and $(b', a, 0)$.

The function $F^{o}_{ph}()$ can be used to insert/delete place-holders in the tree. Once a place older $(a, a', 0)$ exists, a delegation (a, a', A) from the registry module $\mathbf{R}$ can be used to update the place holder to a leaf (a, a', A).

Any node in the tree with root ξ_r can now be sub-delegated to a BGP speaker. Depending on which prefixes need to be delegated to which BGP speaker the owner can use $F^{O}_{ph}()$ to subdivide owned prefixes and swap positions of prefix leaves, and choose the root of a subtree which includes all prefixes to be delegated to the speaker.

Apart from delegating IP prefixes, the AS owner also specifies various preferences as leaves of a hash tree (with root ξ'_n). The types of records in this tree include

1) Pre-path weight; a record of the form $[P, o]$ for each owned prefix P that can be originated by the speaker, indicating the pre-path weight o.

2) Neighbor preferences record for each neighbor. A record for neighbor F is of the form

$$\mathbf{N}_F = [F, s_f = 0, t_f = 0, A_f, L_f, M_f, \tau_f] \tag{6.3}$$

where A_f is the AS number of the neighbor, L_f and M_f are the local preference and MED weights, and τ_f is the maximum permitted duration between HELLO messages from the neighbor N.

The values s_f and t_f are set to zero by the AS owner. Such fields can be modified only by the module of a BGP speaker initialized using the value ξ'_n. The value s_f is the time at which a link to F was established. Value t_f is the time at which the F was last heard-from.

6.4 BGP Speakers

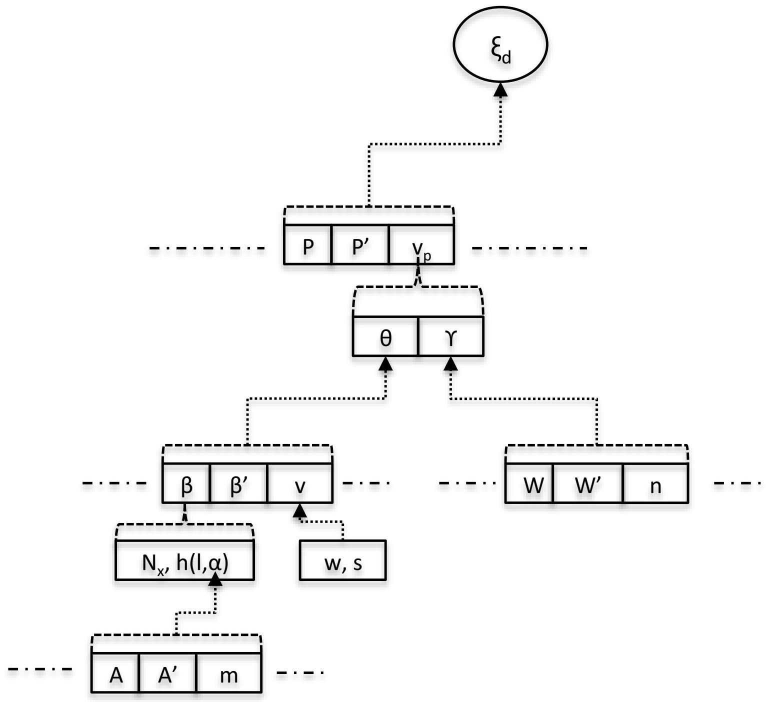
The security kernel of BGP speakers maintains 3 dynamic roots (see Figure 6.2)

1. the root ξ_o of an ROMT. This is initialized to a value ξ'_o communicated by the AS owner module.
2. the root ξ_n of a Merkle tree with a leaf corresponding to every neighbor, and a static leaf for every owned prefix corresponding to which the BGP speaker can originate BGP updates. This root is initialized to the value ξ'_n conveyed by the AS owner module.
3. the root ξ_d, an IOMT indexed by IP prefix — the RIDB tree. This value is initialized to zero.

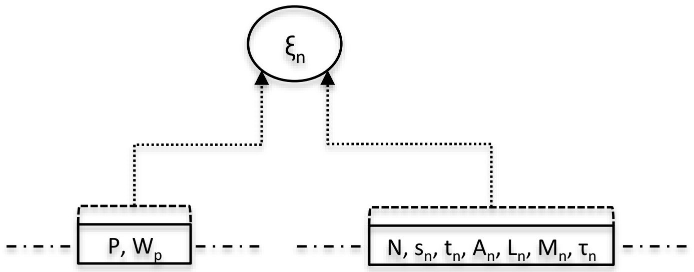
BGP speakers also maintain a static value A — initialized to the AS number represented by the speaker.

During regular operation of the BGP speaker the RIDB root ξ_d is updated whenever a BGP update message is received, or if a path is removed (for example, due to loss of link to neighbor).

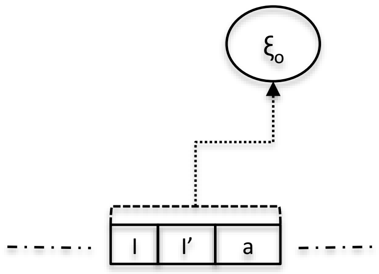
The neighbor/preferences tree root ξ_n is updated whenever a neighbor state is updated. Specifically, corresponding to each neighbor are two dynamic values: a connection identifier s (which is the time at which the connection was initiated) and a time-stamp t (time of last activity in the connection).



RIDB Tree



Policies/Preferences, Neighbor State



Origination Tree

Figure 6.2

OMTs Used by BGP Speakers.

The leaves of the ROMT are IP address ranges for which the speaker can *originate* BGP updates. Originated updates can be for owned IP address ranges or for aggregated prefixes. When initialized, the ROMT root ξ_o is a commitment to leaves corresponding to owned IP ranges (delegated by the AS owner module by conveying a root of a sub-tree from its tree of owned prefixes). In all such leaves the third value a is the AS number. The ROMT root ξ_o may also be updated for purposes of aggregating CIDR prefixes. Specifically, for any prefix in the RIDB tree the address range and the next hop in the best path to the prefix can be added to the ROMT. Thus, for leaves corresponding to foreign IP ranges the third value is the next hop. Two adjacent prefixes with the same next hop can now be aggregated. More specifically, aggregation corresponds to removing a place-holder. For example, two leaves (I_1, I_2, x) and (I_2, I_3, x) where $[I_1, I_2)$ and $[I_2, I_3)$ are two ranges with the same next hop x, can be converted to a single leaf (I_1, I_3, x) through an equivalence operation.

From the perspective of the BGP speaker modules, corresponding to a BGP update message from a speaker (with IP address) X to a speaker Y is an authenticated message from module X to module Y computed as

$$\mu = f_a(X, Y, \{P, \alpha, l, w_{pp}, w_{lp}, w_{med}]\}) \tag{6.4}$$

where P is the prefix for which the path is advertised, α is a one-way function of the AS path, l is the path length, w_{pp}, w_{lp} and w_{med} are respectively the pre-path weight, local preference, MED. The four weights are used to construct a weight represented as

$$W = [w_{pp} \parallel w_{lp} \parallel MAX - l \parallel w_{med}]. \tag{6.5}$$

Thus, for any prefix the path with the highest weight W is the best path.

Security kernel functions $F^S_{rel}()$ and $F^S_{orig}()$ are used to create such BGP update messages, and $F^S_{upd}()$ is used to process such messages from neighboring speakers and update the RIDB root. More specifically

1) $F^S_{orig}()$ is used to originate BGP updates (for own prefixes and aggregated prefixes). Specifically, a path for a prefix P (represented in the origin tree as a leaf with range $[I_1, I_2)$ and third value v) can be advertised only if

1. the third value v is its own AS number, *and* a leaf exists in the tree with root ξ_n for the prefix P, conveying the pre-path weight w_{pp} for prefix P; or
2. the third value v corresponds to a neighbor with a live link, *and* no leaf with prefix P exists in the RIDB tree.

2) $F^S_{rel}()$ is used to relay stored BGP paths in the RIDB to neighbors. $F^S_{rel}()$ identifies the best path for a prefix, and only the best path may be advertised. Alternately, information regarding the best path can also be added to the origination tree to aggregate a prefix.

Neighboring BGP speakers maintain a TCP connection over which BGP update messages are exchanged. To keep the connection alive, and for testing the existence of the link, special HELLO messages are exchanged periodically. From the perspective of the security kernel in a speaker S the link to a neighbor F is associated with the link establishment time s_f and a time-stamp t_f. Once a link has been established, the module in F is expected to confirm their continued presence by periodically sending authenticated time-stamped messages for updating the time-stamp t_f.

In the RIDB-IN, multiple paths, each with possibly different weights, may exist for each prefix. To enable the security kernel to readily determine the path with the highest weight, the plurality of weights for each prefix are maintained as an ordered list.

In the weight IOMT, the index of a leaf is a weight, and the value (third field) is the number of occurrences of the weight in the list. For example, corresponding to a list with four weights $(21, 21, 34, 42)$, three leaves $(21, 34, 2), (34, 42, 1), (42, 21, 1)$ will exist in the weight tree (index 21 occurs twice as indicated by the value field). As in any IOMT, insertion of a place holder (say for index 5, which signifies “zero occurrences of value 5 in the list”) does not modify the list.

Within the RIDB IOMT a special IOMT is also used to represent AS paths. In the AS path IOMT the the index of leaves are ASes. A tree corresponding to a path of length 5 will have 5 leaves. The value field (third field) is the position in the path. As an example, corresponding to a path $A \rightarrow D \rightarrow B \rightarrow E$ the leaves of the tree will be $(A, B, 1), (B, D, 3), (D, E, 2)$ and $(E, A, 4)$ (note that the value for index D is 2 as D is the second AS in the path).

In the RIDB IOMT the index of leaves are IP prefixes. The value field in the IOMT is a one way function of two IOMT roots

1. IOMT root γ — is the root of a weight-IOMT; and
2. IOMT root θ — the root of an IOMT whose leaves like (β, β', v) characterize each path to the prefix.

In the IOMT with root θ

1. the index of leaves are functions of the path; more specifically, in the index $\beta = h(G, h(l, \alpha))$, G is the next hop, l is the path length, and α is the root of an AS-path IOMT root.
2. The value v corresponding to an index β is a function of two values — the weight W of the path, and the connection identifier of the next hop that provided the path. If the connection identifier in a path is not the same as the identifier in the neighbor record for that neighbor, then the path is considered as stale (and the weight is set to 0).

6.4.1 Using Security Kernel Functions in BGP Speaker Module

BGP speaker modules expose a function $F^S_{init}()$ which is invoked to initialize the module. An authenticated message from AS module A (created by using function $F^A_{dp}()$ in

Figure 6.3) is necessary for initializing the roots of the neighbor tree to ξ_n', and the origin tree to ξ_o'.

Any place holder can be added to the IOMT with root ξ_r or the ROMT with root ξ_p, using function $F_{ph}^{S}()$. Any place holder can also be added to the path tree or weight tree corresponding to any prefix. This can be accomplished using function $F_{ph2}^{S}()$ which issues a equivalence memoranda of type $E2$ identifying two roots corresponding to before and after insertion of a place holder in a tree with root θ, or a tree with root γ, or both.

Function $F_S^{hlo}()$ can be invoked to create authenticated messages that can be sent to other speakers. This function ensures that speaker S can only connect to speakers explicitly authorized by the AS owner (by providing the initial root ξ_n). Such authenticated messages can be used to create a connection (with a new value of s deemed sufficiently close to the current time t), and for updating time stamps of neighbors.

6.4.2 Processing BGP Updates

Function $F_{upd}^{S}()$ is invoked to update the RIDB-IN tree — either due to a BGP update message received from a neighbor, or due to loss of link to the next hop. From the perspective of the security kernel the link to the next hop is broken if the time-stamp in the neighbor record is stale. If the current neighbor session identity is different from the session identity of the next hop in the stored path, then the path is assumed to be invalid (as the path was provided during an earlier session). If the neighbor is no longer active, or if the path is invalid, the path weight will be set to 0.

```
  $F^S_{init}(A', \xi'_o, \xi'_n, \mu)$   {
  IF $(f_v(A', S, \{\xi'_o, \xi'_n\}, \mu))$
      $\xi_n \leftarrow \xi'_n;\ \xi_o \leftarrow \xi'_o;\ \xi_d \leftarrow 0; A \leftarrow A';$
}
  $F^S_{ph}(\xi', \rho, o)$   {
  IF $(o = 1) \wedge (\rho = h(ER, \xi_o, \xi', \chi))\ \xi_o \leftarrow \xi';$
  ELSE IF $(o = 2) \wedge (\rho = h(EI, \xi_r, \xi', \chi))\ \xi_d \leftarrow \xi';$
  ELSE IF $(o = 3) \wedge (\rho = h(E2, \xi_r, \xi', \chi))\ \xi_d \leftarrow \xi';$
}
  $F^S_{ph2}(\theta, \theta', \rho_1, \gamma, \gamma', \rho_2, P, P', \rho, \xi, \xi')$   {
  IF $(\rho_1 \neq 0) \wedge (\rho_1 \neq h(EI, \theta, \theta', \chi))$ RETURN;
  IF $(\rho_1 \neq 0) \wedge (\rho_1 \neq h(EI, \gamma, \gamma', \chi))$ RETURN;
  IF $(\rho_1 = 0)\ \theta' \leftarrow \theta;$
  IF $(\rho_2 = 0)\ \gamma' \leftarrow \gamma;$
  $v_p \leftarrow H_L(P, P', h(\theta, \gamma)); v'_p \leftarrow H_L(P, P', h(\theta', \gamma'));$
  IF $(\rho = h(U1, v_p, \xi, v'_p, \xi', \chi))$ RETURN $h(E2, \xi, \xi', \chi);$
}
  $F^S_{hlo}(G, s_g, t_g, A_g, L_g, M_g, \tau_g, \rho, \xi'_n, s'_g, t'_g, \mu, t')\{$
  $v \leftarrow h(G, s_g, t_g, A_g, L_g, M_g, \tau_g);$
  IF $(\mu = 0) \wedge (|t - t'| < \delta)\ \{s'_g = t', t'_g = 0\};$
  ELSE IF $(\mu = 0) \wedge (t' = 0)\ \{s'_g = 0, t'_g = 0;\}$
  ELSE IF $(\mu = 0)\ \{s'_g = s_g, t'_g = t_g;\}$
  ELSE IF $(f_v(G, S, \{s'_g, t'_g\}, \mu) = 0)$ RETURN :
  IF $((s'_g < s_g) \vee ((s'_g = s_g) \wedge (t'_g < t_g)))$ RETURN :
  IF $(\rho = h(U1, v, \xi_n, h(G, s'_g, t'_g, A_g, L_g, M_g, \tau_g), \xi'_n, \chi))$
      $\xi_n \rightarrow \xi'_n;$ RETURN$f_a(S, G, \{s'_g, t\});$
}
```

Figure 6.3

Security Kernel Functions for BGP Speakers.

$F^S_{upd}()$ is invoked to update a path for a prefix P. Recall that a prefix P is associated with a path tree root θ and a weight tree root γ. A path in the path tree is uniquely identified as a function of the AS-path α, path-length l, and next hop N: the index of the path is $\beta = h(N, h(l, \alpha))$. The path is associated with a path weight W_c and the session identity s_n of the next hop.

Updating the path implies modifying the current weight W_c associated with the index β to a weight W. In addition, modification of the weight requires the weight tree to be modified. Specifically

1. if $W_c = 0$ and $W \neq 0$ (inserting a path), then the value W has to be added to the IOMT with root γ;
2. if $W = 0$ and $W_c \neq 0$ (setting path weight to 0) then the value W_c has to be removed from the tree with root γ.
3. if $W = W_c = 0$, then $F^S_{upd}()$ is invoked to delete a path with zero weight. In this case no change is necessary to the weight tree root γ.

For inserting a path $F^S_{upd}()$ is invoked by submitting a received BGP update from a neighbor N specifying path vector α, path length l, and weights $w_{pp} \parallel w_{lp} \parallel w_{med}$. The weight for the inserted path is then

$$W = w_{pp} \parallel x \parallel MAX - l \parallel w_{med} \tag{6.6}$$

where $x = L_n$ or $x = w_{lp}$. Specifically, if the neighbor N providing the update belongs to from a foreign AS, the $x = L_n$ (the local preference of X); if N belongs to the same AS, the local preference w_{lp} advertised by N is retained.

For setting weight to zero $F^S_{upd}()$ may be invoked without a BGP message, or a BGP message that withdraws a previously advertised path. A withdraw message from a neighbor indicates $w_{pp} = w_{lp} = w_{med} = 0$.

In general, updating a path with index β (for prefix P) will require modification to the path tree root θ and weight tree root γ (in the leaf for prefix P). For incorporating the change in values α and γ associated with leaf index P, the RIDB root ξ_d will need to be modified.

The inputs to $F^S_{upd}()$ include

1. a neighbor record $\mathbf{N}_N$ for N and a $V1$ memoranda ρ_n to verify the integrity of the record against the root ξ_n.
2. $U1$ memoranda ρ_t, necessary to update a leaf with index β in a tree with root θ,
3. $U1$ memoranda ρ_w, necessary to increment the counter in leaf with index W (when a path with weight W is inserted), or decrement the counter in a leaf with index W_c (when the current weight W_c of the path is reset to 0), in the weight tree with root γ,
4. $U1$ memoranda ρ_d, necessary to update the RIDB-IN root ξ_d due to the changes to values γ and θ associated with index P; and
5. a received authenticated BGP update message $[\alpha, l, w_{pp} \parallel w_{lp} \parallel w_{med}, \mu]$ from neighbor N.

6.4.3 Advertising BGP Paths

Function $F^S_{adv}()$ is invoked to identify the best path for a prefix and a) advertise the best path (create BGP update) to a neighbor, or b) add the prefix for the path (along with the next hop and session identity of the next hop) to the origination[2] tree. $F^S_{adv}()$ can also be invoked to create a BGP update to withdraw a path with weight 0;

If W is the best weight for prefix P then a leaf (W, W', m) with $W' < W$ and $m \neq 0$ should exist in the tree with root γ. This is demonstrated using a memorandum of type $V1$. There should also exist a leaf for an index $\beta = h(G, h(l, \alpha))$ in the tree with root

[2]This is to enable aggregation of prefixes. Two adjacent prefixes with the same next hop and session identity can be aggregated by removing a place-holder in the ROMT.

```
$F^S_{upd}([P, \alpha, l, w_{pp} \parallel w_{lp} \parallel w_{med}], \mu,$ // Update regarding prefix $P$
    $\rho_n, \mathbf{N}_N = [N, s_n, t_n, A_n, L_n, M_n, \tau_n],$ // from neighboring speaker $N$
    $\beta', (W_c, s_c), \theta, \theta', \rho_t,$ // insert path in path tree
    $\gamma, W', m, \gamma', \rho_w,$ // and weight in weight tree
    $P', \rho_d, \xi'_d)$ //and update RIDB root $\xi_r\{$
  IF $(\rho_n \neq h(V1, h(\mathbf{N}_N), \xi_n, \chi))$ RETURN;
  IF $(\mu = 0) \wedge ((s_n \neq s_c) \vee (t > l_n + \tau_n))$ $W \leftarrow 0;$
  ELSE IF $(\mu = 0) \wedge (W_c = 0)$ $W \leftarrow -1;$
  ELSE IF $(f_v(N, S, \{s_n, P, \alpha_n, w_{pp} \parallel w_{lp} \parallel w_{med}\}, \mu) = 1)$
    IF $(A \neq A_n)$ $W \leftarrow [w_{pp} \parallel L_n \parallel MAX - l \parallel w_{med}];$
    ELSE $W \leftarrow [w_{pp} \parallel w_{lp} \parallel MAX - l \parallel w_{med}];$
  $v \leftarrow (s_c = 0)?0 : h(W_c, s_c); v' \leftarrow (W = -1)?0 : h(W, s_n);$
  $\beta = h(N, h(l, \alpha)); v \leftarrow H_L(\beta, \beta', v); v' \leftarrow H_L(\beta, \beta', v');$
  IF $(\rho_t \neq h(U1, v, \theta, v', \theta', \chi))$ RETURN;
  IF $((W \leq 0) \wedge (m > 0) \wedge (W_c > 0))$
    $v \leftarrow H_L(W_c, W', m); v' \leftarrow H_L(W_c, W', m - 1);$
  ELSE IF $((W_c = 0) \wedge (W < 0))$ $v \leftarrow v' \leftarrow 0;$
  ELSE IF $(W > 0)$ $v \leftarrow H_L(W, W', m); v' \leftarrow H_L(W, W', m + 1);$
  IF $(\rho_w \neq h(U1, v, \gamma, v', \gamma', \chi)))$ RETURN;
  $v \rightarrow H_L(P, P', h(\alpha, \gamma)); v' \rightarrow H_L(P, P', h(\alpha', \gamma'));$
  IF $\rho_d = h(U1, v, \xi_d, v', \xi'_d, \chi)$ $\xi_d \leftarrow \xi'_d;$
}
```

Figure 6.4

BGP Speaker Security Kernel Functionality for Accepting BGP Updates.

θ associated with values s_g and W. For this purpose a $V1$ memoranda is necessary to demonstrate the integrity of a neighbor record for G against ξ_n, and another $V1$ memoranda is required to demonstrate the integrity of the leaf with index β against root θ. Finally, another $V1$ memoranda is required to demonstrate the integrity of values θ and γ associated with index P against the RIDB root ξ_d.

Now that the best path (described by next hop G, AS vector α, path length l and weight W) has been identified,

1. a leaf with range $[x, y)$ corresponding to prefix P can be added to the origination tree indicating next hop and session identity $G \parallel s_g$, or
2. a BGP update for prefix P can be created and sent to a neighbor F.

In the former case, updating the origination tree will require a leaf $(x, y, 0)$ to be modified to $(x, y, G \parallel s_g)$ where $P \equiv [x, y)$. For updating the leaf of the origination tree, a $U1$ memoranda is required as input to $F^S_{adv}()$.

Before a BGP message for a path can be advertised to a *foreign* neighbor F, the path vector and path length have to be modified (to insert own AS number). If the path vector root is currently α, and the length is currently l, the value l should be incremented, and a new leaf needs to be inserted into the IOMT with root α. Specifically, the new leaf will have index A (AS number of the speaker) and value $l + 1$. More specifically, a place holder for A needs to be inserted in a tree with root α, following which the place holder can be updated to modify the third field from 0 to $l + 1$. Thus, a memoranda of type EI (for inserting a place holder) and a memoranda of type $U1$ (for updating the place-holder) are required as inputs.

```
$F^S_{adv}(\alpha, l, \beta', W, \theta, \rho_t, W', m, \gamma, \rho_w, P, P', \rho_d,$
    $\rho_n, \mathbf{N}_G, \alpha_i, \rho_i, \alpha', \rho_{as}, \rho_f, \mathbf{N}_F, \rho_o, \xi'_o)\{$
  IF $((F = 0) \wedge (\rho_n \neq h(V1, h(\mathbf{N}_G), \xi_n, \chi)))$ RETURN;
  ELSE IF $(\rho_n \neq h(V2, h(\mathbf{N}_F), h(\mathbf{N}_G), \xi_n, \chi))$ RETURN;
  $\beta \leftarrow h(G, h(l, \alpha)); v \leftarrow H_L(\beta, \beta', h(W, s_g))$
  IF $(\rho_t \neq h(V1, v, \theta, \chi))$ RETURN;
  IF $(\rho_w \neq h(V1, H_L(W, W', m), \gamma, \chi))$ RETURN;
  IF $(m < 1) \vee (W > W')$ RETURN ;
  IF $(\rho_d \neq h(V1, H_L(P, P', h(\theta, \gamma)), \xi_d, \chi))$ RETURN;
  IF $(F = 0)$//let $[x, y)$ is the address range of prefix $P$
    $P \rightarrow [x, y); v \leftarrow (x, y, 0); v' \leftarrow (x, y, G \parallel s_g);$
    IF $(\rho_o = (U1, v, \xi_o, v', \xi'_o, \chi))$ $\xi_o \leftarrow \xi'_o$; RETURN;
  IF $(A_f \neq A)$
    IF $(\rho_i \neq h(EI, \alpha, \alpha_i, \chi))$ RETURN;
    $l \leftarrow l + 1; v \leftarrow H_L(A, A', 0); v' \leftarrow H_L(A, A', l);$
    IF $(\rho_{as} \neq h(U1, v, \alpha_i, v', \alpha')$ RETURN ;
  ELSE $(\alpha' \leftarrow \alpha)$;
  IF $(W = 0)$ RETURN $f_a(S, F, \{s_f, \alpha', l, 0 \parallel 0 \parallel 0\})$;
  ELSE IF $((t < t_f + \tau_f) \wedge (t < t_g + \tau_g))$
    $[w_{pp}, w_{lp}, w_{len}, w_{med}] \leftarrow W;$
    IF $(A = A_f)$ RETURN $f_a(S, F, \{s_f, P, \alpha, l, w_{pp}, w_{lp}, w_{med}\})$
    ELSE RETURN $f_a(S, F, \{s_f, P, \alpha', l, w_{pp} \parallel 0 \parallel M_f\})$
}
```

Figure 6.5

BGP Speaker Security Kernel Functionality for Relaying BGP Updates.

6.4.4 Originating BGP Updates

$F^S_{orig}()$ is used to advertise path information for two categories of prefixes 1) prefixes owned by the AS; and 2) aggregated prefixes. Specifically, in leaves corresponding to owned prefixes in the origination tree, the third value will be its own AS number A. Corresponding to other leaves the third value will be a neighboring speaker G (next hop for the prefix) and a session identity s'_g of G (at the time the prefix was added to the origination tree).

```
 $F^S_{orig}(P, P', W_p, \rho_o, \rho_r, \rho_n, \mathbf{N}_F, \rho_f, \mathbf{N}_G, s'_g, \xi'_o)\{$
 IF $(A_f = A)$ RETURN;
 IF $(G = 0)$ $v \leftarrow A$; ELSE $v \leftarrow G \parallel s'_g$;
 $P \rightarrow [x, y); v \leftarrow H_L(x, y, v)$;
 IF $((F = 0) \wedge (s_g \neq s'_g))$ // Remove aggregated prefix
   IF $(\rho_n \neq h(V1, h(\mathbf{N}_G), \xi_n, \chi)))$ RETURN;
   IF $(\rho_o = h(U1, v, \xi_o, H_L(x, y, 0), \xi'_o, \chi)$ $\xi_o \leftarrow \xi'_o$; RETURN;
 IF $(t > t_f + \tau_f)$ RETURN ;
 IF $(\rho_o \neq h(V1, v, \xi_o, \chi)$ RETURN;
 IF $(G = 0) \wedge (\rho_n = h(V2, h(P, W_p), h(\mathbf{N}_F), \xi_n, \chi))$
    RETURN $f_a(S, F, \{s_f, P, H_L(A, A, 1), 1, W_p \parallel 0 \parallel M_f\})$
 IF $(\rho_n \neq h(V2, h(\mathbf{N}_F), h(\mathbf{N}_G), \xi_n, \chi)))$ RETURN;
 IF $(\rho_r \neq h(V1, H_L(P, P', 0), \xi_r, \chi))$ RETURN;
 IF $(t < t_g + \tau_g)$
    RETURN $f_a(S, F, \{s_f, P, H_L(A, A, 1), 1, 0 \parallel 0 \parallel M_f\})$
}
```

Figure 6.6

BGP Speaker Security Kernel Functionality for Originating BGP Updates.

An owned range $[x, y)$ can be converted into a prefix P and advertised to a neighbor F only if a record (P, w_{pp}) exists in the neighbor/policies tree with root ξ_n. A certificate of type $V2$ is provided as input to simultaneously verify the integrity of the neighbor record $\mathbf{N}_F$ and record (P, w_{pp}) in the neighbor tree.

To advertise an aggregated prefix P a $V1$ memoranda attesting the integrity of the next hop neighbor record $\mathbf{N}_G$ is required. In addition, a $V1$ certificate is required to demonstrate that prefix P does *not* exist in the RIDB-IN tree.

If the next hop F (to whom the origination message is to be sent) is set to $F = 0$, then $F^S_{orig}()$ interprets this as a request to delete an aggregated leaf for prefix P with third value $G \parallel s'_g$. To remove the aggregated prefix the third value $G \parallel s'_g$ is set to 0. For this purpose a certificate ρ_o of type $U1$ is required as input.

When a BGP message is originated for an owned prefix or an aggregated prefix the MED weight is set to to value M_f (for the intended receiver F) provided by the AS owner; the local preference is set to 0; for owned prefixes the pre-path weight is set to the value W_p prescribed by the AS owner, and for aggregated prefixes the pre-path-weight is set to 0.

6.5 Related Work

The current approach to secure BGP is based on the Secure BGP [31] protocol proposed by Kent et. al. This approach employs public key certificates to authenticate communication between ASes (BGP updates) and delegation of AS numbers/IP prefixes. More specifically, a dual certificate system (supported in the back-end by a public key infrastructure (PKI)) is used where the one certificate binds the public key of the AS owner to the operating address space (IP prefix) and AS number, and a second certificate binds routers to an AS. Apart from such static certificates, dynamic certificates are also created by BGP speakers along with every update message. Specifically, such certificates created by every AS in the path seeks to assure the integrity of the AS path vector. Whenever a router receives an update message, it verifies the dual certificates to ascertain the validity of the message. In order to advertise the received message it extends the path by adding itself to the path and signing it (along with the nested signatures of the previous hops) with its own public key. To prevent deletion attacks a speaker in AS A sending an update message to a speaker in AS B also includes the next hop B in the signature.

While S-BGP approach is successful in its claims for identity verification (AS owner, routers) and update message integrity, it fails to provide any assurances for the overall operation of a subsystem in the protocol. For example, there are no assurances provided by the protocol guaranteeing that a router will indeed select the best path and that it will strictly abide by the policies and preferences prescribed by the AS owner. The security features of S-BGP protocol does not extend to aggregated prefixes as it is impractical to create static certificates to validate "ownership" of aggregated prefixes. This is a severe disadvantage of S-BGP as much of the advantages of CIDR stem from the ability to aggregate prefixes.

In the proposed approach the simple security kernel associated with BGP speakers ensure that the speakers can only advertise the best path, that all preferences and policies of the AS owner will be strictly adhered to. More importantly, the assurances also extend to aggregated prefixes.

CHAPTER 7

TRUSTED COMPUTING BASE FOR MANET DEVICES

A mobile ad hoc network (MANET) [55] is a dynamic subnet constituted of mobile computers (or devices) with limited wireless transmission range. MANET devices rely on each other for routing packets among themselves, and consequently, do not depend on a dedicated routing infrastructure.

A MANET routing protocol is a set of rules which dictates the tasks to be performed by every device in a MANET subnet to enable discovery of multi-hop paths for relaying data packets. Such rules govern processes like discovery of neighbors (devices within limited wireless transmission range), exchange of routing information between neighbors, maintaining a destination table (DT) and a neighbor table (NT) at every device, using information in the DT and NT to forward data packets, etc.

The rules that govern the actions of a MANET device are typically encoded as software executed by the device. Unintended functionality — either deliberately hidden malicious functionality, or accidental bugs — in any component of a mobile device could potentially be exploited by attackers. Specifically, they could be exploited to a) modify the functionality (routing software) of the device or b) modify the information stored by the device (in DT and NT), or c) expose secrets of the device, thereby enabling the attacker to impersonate the device to advertise arbitrary "routing information." Attackers could be legitimate owners of a device, or entities who may have exploited some hidden functionality in the device to acquire some extent of control over the device. Many such attackers may even collude together to wreak havoc on the ad hoc subnet.

A practical MANET device can come in several shapes and sizes, ranging from laptops to smart phones to special purpose sensors, constructed using a wide range of hardware/software components. It is obviously far from practical to rule out the presence of undesired functionality in *every* hardware and software component of *every* device that could take part in a MANET subnet, or malicious behavior / incompetence in every *entity* that has the ability to gain control of a device. It is, however, far more practical to assure the integrity of a *single component in every device*. For example, every MANET device could be required to possess a trustworthy chip/module.

In the rest of this chapter we shall refer to such a module/chip as a trusted MANET module (TMM). It is assumed that secrets protected by TMMs cannot be exposed, and the functionality of TMMs cannot be modified. All other software and hardware in every MANET device, and the user in control of the device (either through legitimate or illegitimate means), are assumed to be untrusted/hostile. The trust in TMMs, and more importantly, *only* the trust in TMMs, is leveraged to realize the assurance that "any device that does not strictly abide by the protocol will *not* be able to participate in the MANET."

7.1 Background

Any routing protocol can be seen as an extension of two basic routing strategies — distance vector (DV) or link-state (LS) approaches. In LS protocols information regarding a destination D is in the form of the state of all links of D (to neighbors of D). All nodes in the subnet possess the same information regarding D.

In DV protocols information regarding D is a destination record (DR) that indicates the hop count to D, and the next hop in the path to D. In general, every node possesses a different DR for destination D. Neighbors exchange DRs amongst themselves, and by comparing DRs for a destination D obtained from all neighbors, a node can determine its best DR for D, which is then stored in a table of DRs — the destination table (DT).

7.1.1 Distance Vector Protocols

A majority of popular MANET routing protocols are based on the DV approach in which every node maintains a destination table (DT) and a neighbor table (NT). The DT consists of a destination record (DR) for each possible destination. The NT consists of a neighbor record (NR) for each neighbor.

A DR $[q, x, m, n]$ for a destination D (mobile device with address D) indicates a sequence number q, the time of expiry x, hop-count m to the destination, and the next-hop neighbor n in the path to the destination. For an unreachable destination, the hop-count is ∞ (an integer constant larger than the maximum allowable hop-count). A "neighbor" of a device is another device to which the existence of a bidirectional path has been confirmed. Neighbors are expected to confirm each other's continued presence, possibly by exchanging periodic HELLO messages. A neighbor who has has been silent for a long duration may no longer be considered a neighbor.

In all DV protocols, a DR for destination D is initiated by D, indicating a fresh sequence number, hop count (to itself as) $m = 0$, and time of expiry x. The next hop is set to itself (or $n = D$). This DR may then be advertised to all its neighbors. Neighbors of D store the DR in their respective tables after incrementing the hop-count field m to 1, and setting the next hop as D. The stored DR may then be advertised to *their* neighbors. For example, in a path $D \rightarrow C \rightarrow B$, device B receives a DR for D from C indicating hop count $m = 1$ and next hop as $n = D$. Device B stores the DR with hop-count set to $m = 2$ and next hop as $n = C$. In this fashion, any node in a connected subnet can acquire a DR for destination D: a device r hops from D will have a DR for D in it's table, with hop count $m = r$.

More specifically, a device may receive a DR for D from each of its neighbors. In general, the stored DR is replaced by the received DR (after incrementing hop count and modifying the next hop to the neighbor that sent the DR) if the received DR is fresher (higher sequence number), or equally fresh and better (lower hop-count), or equally fresh and provided by the current next hop.

Ultimately, the purpose of maintaining the DT and NT is to relay data packets. A device S which desires to relay a data packet to D can do so only if it has a *usable* DR for D. A DR $[q, x, m, n = R]$ for a destination D is considered as usable only if $m < \infty$, the DR has not expired, and the next hop $n = R$ continues to be a neighbor. The data packet may now be relayed to the next hop $n = R$. The device R at the next hop can likewise relay the data packet onwards to *it's* next hop, indicated in a usable DR for D in *its* DT, and so on, until the data packet reaches D. When a device X receives a data packet intended for a destination D from a neighbor Y, and finds that it no longer has a usable DR for D, it responds by advertising a route error (RERR) packet. Y may now update it's stored DR for D (by setting $m = \infty$) and relay the RERR to it's upstream neighbor (who had earlier sent the data packet to Y), and so on.

In proactive DV protocols like destination sequenced DV (DSDV) [49] every node initiates a DR periodically — each time with a higher sequence number q. In reactive protocols like *ad hoc on demand* DV (AODV) [48], nodes initiate DRs only if they desire to send/receive data to/from another node. For this purpose, a node S desiring to send data packets to D (and finds that it does not have a usable DR for destination D) advertises a route request (RREQ) packet which includes a fresh DR for itself, and it's unusable DR for D. The RREQ is flooded in the subnet. Any node with a usable DR for D, (or D itself) may include the DR in a route response (RREP) packet which is relayed back to the source.

7.1.2 Covert and Overt Attacks

Broadly, an attack by a participating device (say, A) on a MANET can be seen as an attempt to send a routing/data packet $\mathbf{P}$ to a neighbor (say, B) where the contents of packet $\mathbf{P}$ were *not generated in strict compliance with the protocol*. Even more generally, an attack may also include the act of *not* sending a packet $\mathbf{P}$ (when the protocol calls for a packet to be sent).

An attack is *successful* if B is unable to distinguish between packets created strictly according to the protocol and packets created in violation of the protocol. If B is able to determine that a packet has been created by A in violation of the protocol, or that A should have sent a packet (when it did not), the attack is deemed unsuccessful (as B now has the ability to penalize A by no longer entertaining A as a neighbor).

Attacks that can be inflicted on a MANET subnet by a participating device can be broadly classified into *overt* and *covert* attacks. Overt attacks include incorrect packet formats, and illegal modifications to a relayed DR (for example, changing sequence number, expiry time, or changing hop count in any other way except incrementing by one, etc.). The reason that such attacks are overt is that

1. attacks like incorrect formats can be readily identified, and
2. *if* suitable cryptographic authentication strategies[1] are used to protect the integrity of DRs advertised by devices, then the receiver of such a packet can readily detect illegal modifications and drop the offending packet.

The main challenges in the practical realization of assurances against overt attacks are two fold. The first stems from the overhead for cryptographic authentication schemes — especially for carrying over authentication over multiple hops. The second is that cryptographic strategies are (ultimately) at most only as strong as the mechanism used for protection of a device's secrets. The absence of reliable mechanisms to protect secrets assigned to devices from attackers (who may even be the owner of a device) implies that attackers may even share secrets exposed from multiple devices to advertise misleading (but duly authenticated) "routing information" at will.

Unlike overt attacks, covert attacks may not be readily discernible by the receiver of a packet — even with sophisticated cryptographic authentication schemes. Examples of such attacks include i) replaying DRs that were invalidated (or rendered sub-optimal) due

[1]For example, digital signature of the originator of the DR for protecting integrity of immutable DR fields, and carrying over authentication for protecting the integrity of the hop-count field.

to recent changes in topology; ii) rebroadcasting RREQs (instead of responding with an RREP) when a usable path exists; iii) not relaying data packets, or relaying data packets incorrectly (for example, to a device that is *not* the next hop in the best DR for the destination); iv) invoking unwarranted RERR packets (even when the link to the next hop is *not* broken); v) accepting packets from (or relaying packets to) "neighbors" to whom a bidirectional link does not exist (thereby, making sure that the reverse path will fail), etc. vi) attacks based on misrepresentations of current time; for example, the clock of a device may be modified to make it think that a DR has expired.

The reason that such attacks may not be easily detectable is that in a scenario where a device C receives a packet from a device A, there is no tangible way for the routing process in a device C to confirm that A *does* have a link to it's neighbor B (when A claims that the link is broken) or that A *does* have a better path (when A advertises a suboptimal path). While it might appear that a fairly sophisticated monitoring process in a neighbor of C, (say) Y, which had earlier sent the better path to A, may be capable of detecting A's malicious intention it is entirely possible that Y's advertisement was not received by A (for example, due to collision).

Furthermore, it is also possible for an attacker A to exploit collision-avoidance mechanisms used in wireless medium access protocols to send some information to it's all neighbors while simultaneously ensuring that the information will not be heard by a *specific* neighbor B. For example, A can get to know of an impending reception by B from a CTS (clear to send) packet from B. By transmitting information at a time that overlaps with B's reception, A can ensure that it's suspicion-raising transmission will not be heard by B. An attacker A can also exploit this ability to carry out other possible attacks like i) claiming it has lost it's link to B to all its other neighbors (but continue to use B as a neighbor for it's own purposes); ii) faking relay of a data packet or route error packet to the next hop, etc.

From a broad perspective, covert attacks can be seen as *replay* attacks where the sender is able to make *contradictory statements to different entities*. Any rogue process in a MANET device may be able to perpetrate such attacks by either modifying the routing process, or hardware drivers, or modifying the contents of the DT/NT, or by modifying MANET parameters like τ, ∞, τ_s etc. Even a less sophisticated rogue process that does *not* have the ability to do so, may be able to modify the *interpretation* of the contents of a DT / NT by resetting the device's clock.

7.2 Overview of TMM Based Approach

To our knowledge, the proposed approach is the first to address both overt *and* covert attacks, under the following reasonable assumptions:

1. a secure pre-image resistant cryptographic hash function $h()$ exists; and
2. every MANET device possesses a TMM that is read-proof and write-proof.

All TMMs are identical except that each has a unique identity, and possess unique secrets which enable any two TMMs to compute a pair-wise secret. Every TMM has a clock which ticks at (very close to) the same frequency. However, the clocks of TMMs are *not* synchronized. We shall assume that the identity of the TMM in a device A is also A. However, to distinguish between the device and it's TMM, we shall refer to "device A" as $\bar{A}$ and "TMM A" as A.

It is assumed that all components of MANET devices and the user(s) in control of the device are untrusted/hostile. In other words, the untrusted user/device is able to modify the routing software, and/or the device's clock, and/or the DT/NT, and may possess complete control over the wireless interface. Notwithstanding such capabilities attributed to the rogue software/hardware in the device or the user controlling the device, the goal is to ensure that "all devices will indeed abide by the protocol rules."

In the proposed approach, any MANET packet sent by a device $\bar{A}$ to device $\bar{B}$ should be accompanied by a corresponding *message* from TMM A to TMM B. Pairwise secrets between TMMs are used to compute message authentication codes (MAC) for assuring the integrity of such messages. As devices cannot impersonate TMMs, device $\bar{A}$ is required to request it's TMM A to create a message, and deliver the message to device $\bar{B}$; device $\bar{B}$ is similarly expected to submit the message to *it's* TMM B and receive an acknowledgement message that can be conveyed back to A through $\bar{A}$.

7.2.1 Two-Step Approach

The TMM approach to secure MANET routing can be seen as consisting of two broad steps

1. representation of protocol rules as a simple algorithm $f()$ that can be executed even inside the confines of severely resource limited TMMs, and
2. ensuring the integrity of inputs and outputs of the algorithm;

Protocol Rules: Towards the first step we outline an algorithm $f_{dv}()$ suitable for any distance vector based protocol. The inputs to the $f_{dv}()$ are restricted to one DR for a destination, up to two NRs (corresponding to two neighbors), protocol specific constants, and parameters associated with an *event*.

An *event* can be the a message received from another TMM, or a request from the device. The occurrence of an event may necessitate i) a modification to the DR (say, for destination D) and/or two NRs (say, for neighbors F and G) and ii) creation of up to 2 messages — one intended for neighbor F and one for G. For each type of event (specified by event parameters) the algorithm $f_{dv}()$ specifies a) the manner in which a DR for D and up to two NRs for F and G will need to be modified; and b) the type of messages to be created and sent to F and G.

7.2.2 Integrity of Inputs and Outputs

Towards assuring the integrity of inputs/outputs of $f_{dv}()$ it is required to a) assure the integrity of all dynamic DRs and NRs stored in the DT/ NT; b) protect the integrity of the (static) constants, and c) assure the integrity of messages exchanged between TMMs.

7.2.2.1 Integrity of DT and NT

Resource limited TMMs cannot store the entire DT and NT inside their protected boundary. TMMs maintain a succinct summary of the DT and NT in the form of two cryptographic hashes ξ_{dt} and ξ_{nt} respectively. More specifically, ξ_{dt} and ξ_{nt} are roots of two Index Ordered Merkle Trees (IOMT): root ξ_{dt} corresponding to the DT, with DRs as leaves of the tree, and root ξ_{nt} corresponding to the NT, with NRs as leaves.

Similar to the Merkle tree [38], the IOMT is a binary hash tree maintained by an untrusted *prover* to demonstrate the integrity of dynamic records (also maintained by the prover) to a resource limited *verifier* that stores only a single cryptographic hash — the root of the tree. For a database with N records, the prover stores all N records, and in addition, $2N - 1$ cryptographic hashes, which are nodes of the binary tree, distributed over $\log_2 N$ levels. The verifier stores only a single node — the root of the tree.

In order for a resource limited verifier (for our purposes, the TMM) to be able verify the integrity of *any* number of dynamic records stored in an untrusted location (the untrusted device), even a plain Merkle hash tree can be used. However, to address covert attacks, it is not sufficient for a TMM to be able to merely verify the integrity of a DR for a destination D or a neighbor record for a neighbor R; it is also essential for the TMM to be able to verify that an NR for R or a DR for D *does not exist*. If TMMs cannot readily verify non-existence, the untrusted device will be able to *hide* a DR or NR that *does* exist, to perpetrate covert attacks. The use of IOMT instead of a plain Merkle tree prevents such attacks.

7.2.2.2 Integrity of Messages

The integrity of constants are assured by including a one-way function of the constants in the process of computing shared secrets between TMMs. Protocol specific constants are used in the process of initializing a TMM to operate in a specific MANET. All TMMs in a MANET will need to be initialized with the same constants, as TMMs initialized with different constants will not be able to agree on a shared secret. Such pairwise secrets are used for computing message authentication codes (MAC) for TMM messages to assure the integrity of messages in transit.

However, protecting the integrity of messages in transit is not sufficient. It is also necessary to have proactive strategies to guarantee that messages created by a TMM A for consumption of TMM B are actually delivered to TMM B. Note that in the path between the two TMMs A and B are two untrusted middle-men - the devices $\bar{A}$ and $\bar{B}$ that house the TMMs, who can easily drop messages.

This issue is addressed by employing "locks". A lock is a special field s in the NR. When a TMM A sends a message to a TMM B it sets the lock s in the NR of B. The lock can be reset only if an acknowledgement is received from B. As TMM messages cannot be impersonated, the acknowledgement from B can be provided to A only if the message from A was actually delivered to B. If the lock is not reset, A will no longer consider B as a neighbor. It does not matter if the misbehaving device (that dropped the message) was A or B. Both A and B will stop regarding each other as neighbors, and thus, will not be able to exchange messages.

7.2.3 Records and Messages

From the perspective of a TMM, a record is of the form $\mathbf{r} = [v_1, v_2, v_3, v_4]$, and is associated with a record hash

$$\omega = h_r(\mathbf{r}) = \begin{cases} 0 & v_1 = 0 \\ h(\mathbf{r}) & v_1 \neq 0 \end{cases} \tag{7.1}$$

A destination record (DR) for a destination D is of the form $\mathbf{r}_D = [q_d, x_d, m_d, n_d]$ where the four values are the sequence number, time of expiry, hop count, and next hop. The DR $[q_a, x_a, m_a, n_a]$ for a destination A is created by TMM A. As the clocks of different TMMs are not synchronized, the time of expiry is in terms of the clock of the TMM of the device in which the DR is stored. A DR with sequence number 0 is interpreted as a *empty* record $[0, 0, 0, 0]$

The NR $\mathbf{r}_F$ for a neighbor F specifies 3 values $[l_f, o_f, s_f, 0]$ (time neighbor was last-heard-from, the offset of the neighbor's clock, and lock s (the fourth value is always zero, and is ignored). Once again, all values of time are according to the clock of the TMM in the device storing the NR. An NR with first field $l = 0$ is interpreted as an empty record.

TMMs exchange authenticated messages (authenticated using pairwise secrets) of three types — HLO messages, DR messages that convey a DR, or data messages that convey the hash of a data packet. All messages have a common format

$$\mathbf{M} = [R, y_r, t_r, a_r, D_r, \nu_r] \tag{7.2}$$

where R is the identity of the sender (TMM that created the message), y_r is the type of message (HLO, DR or DATA); t_r is a time-stamp of the sender R, a_r is an acknowledgement field, which is 0 for a spontaneous message, and for an ACK, set to the time-stamp of the message that is acknowledged. The value D_r is the identify of a destination, and ν_r is a cryptographic hash.

In DR messages, D_r is destination that created the DR conveyed by the message. In DATA messages, D_r is the ultimate destination of the data packet whose hash is conveyed by the message. In HLO messages $D_r = 0$. In a DR messages the value ν_r is the record hash of the received DR for D_r. In a DATA message ν_r is a one way function of the source S of the data packet and the hash γ of the data packet.

7.2.4 TMM Functions

The functional components of TMM can be broadly classified into

1. IOMT functions that enable the TMM to maintain two virtual databases — a DT and NT — by storing only the IOMT roots;
2. Functionality than enables TMMs to establish pairwise secrets, and use such secrets to authenticate messages; and
3. Protocol specific functionality expressed as an algorithm $f_{dv}()$ executed inside TMMs.

These functional components are accessed through interfaces exposed by the TMM. The interfaces/functions exposed by TMMs include the following:

1) IOMT related functions $F_{mt}()$, $F_{cat}()$ and $F_{eq}()$ that perform simple sequences of hash operations and issue various types of *self-memoranda*. A self-memoranda issued by a TMM to itself (for use at a later time) is authenticated using a secret χ known only to the TMM.

2) Function $F_{init}()$ to initialize a TMM to operate in a MANET.

3) Function $F_{msg}()$ to notify the TMM of the occurrence of an "event." An event can be receipt of a message from another TMM, or a request from the device (for example to send a DR, initiate a data packet, remove a stale DR/NR, etc); $F_{msg}()$ stores event (message) specific parameters like R, y_r, t_r, a_r, D_r, ν_r and the event time $\tilde{t}$ (time at which the occurrence of the event was notified) in a reserved *event register* **E** inside the TMM.

4) Function $F_{upd}()$ which executes $f_{dv}()$. Inputs to $F_{upd}()$ include two DRs for D (a stored DR and a received DR) and two NRs (for F and G). Depending on the nature of the event, execution of $f_{dv}()$ may result in the modification of up to the three records (a DR

and two NRs). Accordingly, $F_{upd}()$ updates the IOMT roots and creates messages dictated by two outputs O_f and O_g of algorithm $f_{dv}()$.

Inputs to $F_{upd}()$ also include self-memoranda which simultaneously enable the TMM to a) verify the consistency of the DRs and NRs against the current IOMT roots, and b) modify the roots in accordance with the changes to the DR/NRs resulting from execution of $f_{dv}()$.

$F_{upd}()$ ensures that only DRs/NRs consistent with the current IOMT state can be provided as inputs and modified only as specified by the algorithm $f_{dv}()$. On completion of $F_{upd}()$ the device is expected to take the following steps:

1) Modify the DR and two NRs in exactly the same manner. If not, the DR and the NRs will no longer be recognized as consistent with the IOMT roots stored inside the TMM. Inconsistent DRs cannot be advertised to other devices or used for forwarding data packets. No messages will be accepted from / sent to neighbors with inconsistent NRs.

2) Send the messages to F and G. If a device does not send the message to a neighbor $F = Y$, an acknowledgement from Y cannot be submitted to the TMM (as TMM messages cannot be faked) to reset the lock s in the neighbor record of Y, which will result is then the loss of the link to Y.

7.2.5 Distance Vector Algorithm

Protocol specific components of the TMM based approach include a specification of constants, and the algorithm $f_{dv}()$. (Figure 7.1). The inputs to $f_{dv}()$ include a DR for D two NRs (F and G), event related parameters, and constants $\infty, \tau, \tau_s, \tau_r$ and τ_p. If $D = 0$ the implication is that no DR has been provided as input (if $F = 0$ no NR for F is provided). $q_d = 0$ it signifies an empty DR. $l_f = 0$ implies an empty NR for F. $D = I$ implies self-DR (as I is the TMM identity). Most often, the neighbor F is the source of a event (or $F = R$), and the neighbor G is the next hop n_d in the DR for D (or $G = n_d$) .

In the algorithm in Figure 7.1 the events can be classified into three broad categories: 1) request from the device ($R = 0$, lines 3-16); 2) message from F (or $R = F$, lines 1-2 and 17-29) and 3) message from G ($R = G$, lines 30-33). Exection of $f_{dv}()$ may result in the modification of the DR/NRs and two outputs O_f and O_g which specify the nature of the message to be sent to F and G respectively.

The constant τ_r is the maximum permitted round trip time to recognize the existence of a neighbor. If $l_f = 0$ (empty record for F) and if the received message from $F = R$ is an ack., such that $\tilde{t} - a_r < \tau_r$, a successful handshake has occurred (lines 1-2). The time t_r according to the sender is roughly $l_f = (\tilde{t} + a_r)/2$ in terms of the receivers clock. The clock offset of the sender is then $o_f = l_f - t_r$.

Adding a NR for F after a successful handshake causes the NR for F to change from $[0, 0, 0] \rightarrow [l_f, o_f, 0]$. Once a NR has been added, the neighbor is expected to periodically affirm it's continued presence by sending messages (HLO messages if there is no reason the send other messages). On receipt of a message from F with a time stamp t_f the last-heard-from field is updated to $l_f = t_f + o_f$.

The value τ_s is the maximum period of silence. If the time stamp l_f in the NR for F is older than a duration τ_s, F will no longer be considered an *active* neighbor. Messages will not be accepted from inactive neighbors. Consequently, their time-stamps cannot be updated. However, an NR for an inactive neighbor cannot be removed as soon as they become inactive. If no ack. is outstanding ($a_f = 0$) a stale NR for F can be removed if F has been inactive for duration τ. If the neighbor has been inactive due to failure to send an acknowledgement, the inactive NR is retained for duration $\tau_p >> \tau$ (lines 3-5). The reason for retaining a stale NR of F for some duration is to ensure that F cannot be added back as a neighbor (after performing a handshake).

DR and DATA messages can be sent only to active neighbors, and only if the neighbor does not have the lock s set. A neighbor F is active at event time $\tilde{t}$ if $\tilde{t} - l_f < \tau_s$. To send a DR message to F to send the DR for D a value O_f is set to 1. To send the DR to G the value O_g is set to 1. When a DR or DATA message is sent to active neighbor F with $s_f = 0$, the lock s_f is set to $\tilde{t}$. Later, when an ack. is received from F with $a_r = s_f$, the lock is reset.

An acknowledgement for a DR / DATA message from F can be sent by setting $O_f = 2$ or $O_f = 3$. Specifically, $O_f = 2$ implies a simple acknowledgement (the only purpose of which is to reset the lock). $O_f = 3$ is a DR message that simultaneously acknowledges a received message and conveys a DR. $O_g = 4$ implies initiation of a DATA message to G; $O_g = 5$ implies relaying a DATA message to G.

For example, when a DATA message is received from F to indicating $D_r = D$ as the destination, if the DR for D is usable $m_d < \infty$, and the next hop $n_d = G$ is active, then a simple acknowledgement is sent to F (by setting $O_f = 2$); to forward the DATA message to G the value O_g is set to 5. However, if no valid DR for D exists, the acknowledgement sent to F also conveys the bad DR for D ($O_f = 3$) so that F can update it's DR for D (as the DR has been invalidated). Similarly, when an invalid DR message is received for F and the stored DR is good, the good DR is sent along with the ack.

```
INPUTS
$(D, [q_d, x_d, m_d, n_d]), (F, [l_f, o_f, s_f]), (G, [l_g, o_g, s_g])$ //DR,2 NRs
$(R, y_r, t_r, a_r, D_r, \tilde{t}), (\infty, \tau, \tau_r, \tau_s, \tau_p)$ //Event, Constants
$f_{dv}()\{$
  IF $(y = \text{HLO}) \wedge (\mathrm{l_f} = 0) \wedge (\tilde{\mathrm{t}} - \mathrm{a_r} < \mathrm{\tau_r}) \wedge (\mathrm{R} = \mathrm{F})$
    $l_f \leftarrow (\tilde{t} + a_r)/2; o_f \leftarrow l_f - t_r; s_f \leftarrow 0;$
  ELSE IF $(R = 0) \wedge (l_f < \tilde{t} - \tau)$
    IF $(s_f = 0)$ $\{l_f \leftarrow 0;\}$
    ELSE IF $(l_f < \tilde{t} - \tau_p)$ $\{l_f \leftarrow 0;\}$
  ELSE IF $((R = 0) \wedge (D = I))$
    IF $(F = 0)$ $\{q_d \leftarrow ++c;\ x_d \leftarrow \tilde{t} + \tau;\ m_d \leftarrow 0;\ n_d \leftarrow I;\}$
    ELSE IF $((l_f < \tilde{t} - \tau_s) \wedge (s_f = 0))$ $\{O_f \leftarrow 1; s_f \leftarrow \tilde{t};\}$
  ELSE IF $(R = 0) \wedge (D \neq 0) \wedge (n_d = G)$
    IF $((m_d < \infty) \wedge (x_d < \tilde{t}))$ $\{m_d = \infty; n_d \leftarrow 0;\}$
    ELSE IF $(m_d \geq \infty) \wedge (x_d < \tilde{t})$ $\{q_d \leftarrow 0;\}$
    ELSE IF $(l_g < \tilde{t} - \tau_s)$ $\{m_d = \infty; n_d \leftarrow 0;\}$
    ELSE IF $((l_f < \tilde{t} - \tau_s) \wedge (s_f = 0))\{O_f \leftarrow 1; s_f \leftarrow \tilde{t};\}$
    ELSE IF $((0 < m_d < \infty) \wedge (l_g < \tilde{t} - \tau_s) \wedge (s_g = 0)$
      $O_g \leftarrow 4; s_g \leftarrow \tilde{t};$
  ELSE IF $(R = 0)$ RETURN ERROR;
  ELSE IF $(R = F) \wedge (l_f > \tilde{t} - \tau_s)$
    IF $(s_f = 0) \vee (s_f = a_r)$ $l_f \leftarrow \max(l_f, o_f + t_r);$ $\{s_f \leftarrow 0;\}$
    IF $(D_r = I) \wedge (y_r = \text{DATA})$ $\{O_f \leftarrow 2;\}$
    ELSE IF $((y = \text{DR}) \wedge (D = D_r) \wedge (t_r + o_f > l_f) \wedge (n \neq I))$
      IF $(q > q_d) \vee (q = q_d) \wedge (m < m_d + 1)$
        $q_d \leftarrow q;\ x_d \leftarrow x + o_f; m_d = m + 1; n_d \leftarrow F;\ O_f \leftarrow 2$
      ELSE IF $((m_d < \infty) \wedge (a_r = 0) \wedge (s_f = 0))$ $\{O_f \leftarrow 3;\}$
      ELSE $O_f \leftarrow 2;$
    ELSE IF $((y = \text{DATA}) \wedge (D = D_r) \wedge (a_r = 0))$
      IF $(0 < m_d < \infty)$ $\{O_f \leftarrow 2; O_g \leftarrow 5; s_g \leftarrow \tilde{t};\}$
      ELSE IF $(m_d \geq \infty) \wedge (s_f = 0)$ $\{O_f \leftarrow 3; s_f \leftarrow \tilde{t};\}$
    ELSE IF $(y = HLO);$
    ELSE RETURN ERROR;
  ELSE IF $((R = G) \wedge (y = \text{DR}) \wedge (\mathrm{D} = \mathrm{D_r}) \wedge (\mathrm{t_r} + \mathrm{o_g} > \mathrm{l_g}))$
      IF $(q \geq q_d) \wedge ((n_d = 0) \vee (n_d = G))$
        $q_d \leftarrow q;\ x_d \leftarrow x + o_g; m_d = m + 1;\ n_d \leftarrow G;$
      IF $(s_g = 0) \vee (s_g = a_r)$ $\{l_g \leftarrow t_r + o_g; s_g \leftarrow 0; O_g \leftarrow 2;\}$
}
```

Figure 7.1

DV Algorithm.

Creating a new DR for itself implies incrementing the monotonic counter c and using it as the sequence number for the freshly created DR. The time of expiry is set to $\tau + \tilde{t}$. Hop count is set to 0, and next hop is set to itself (line 7). The self DR can be sent to F if F is active (line 8).

Lines 9-13 depicts events for which DR D needs to be updated on request by the device: setting height to ∞ in an expired DR (line 10); deleting an expired DR (line 11); setting hop count to ∞ as the next hop $n_d = G$ is inactive (line 12).

Lines 13-15 depicts events for sending a DR to an active neighbor F (line 13); initiate a data packet to D by creating a DATA message $(O_g = 4)$ to next hop G in a valid DR (lines 14-15).

Lines 17-29 correspond to events where a message has been received from active neighbor $R = F$. Update time stamp if no lock has been set, or if the message is an ack. that clears the lock ($a_r = s_f$) (line 18); DATA message with the receiver I as the destination D_r; send ack to F ($O_f = 2$) (line 19). DR message (line 20-24), updated and acknowledged ($O_f = 2$) as the received DR is better or fresher (lines 21-22). When a DR is updated the expiry time is converted from the sender's clock to receiver clock. If the stored DR is better (line 23), ack with stored DR ($O_f = 3$) only if F has no outstanding acks (else a simple ack is sent - line 24).

DATA message (lines 25 to 27) from F with $D_r = D$ which is *not* an ack ($a_r = 0$); relayed to the next hop G (if a path exists to D) along with an ack to F (or $O_g = 5, O_f = 2$); if path does not exist DR message is sent to F as ack ($O_f = 3$).

Lines 30-33 G is the source of a DR message. The DR is updated (lines 31-32) if fresher or equally fresh. The time stamp is updated and an ack created ($O_g = 2$). G can be the source of DR messages under three conditions: a) when no DR currently exists and the DR supplied by G makes G the next hop; b) when G provides an update (which could be a shorter or longer path); or c) if in response to a DATA message sent to the next hop G, a DR message is received (route error).

7.3 IOMT used by MANET nodes

An IOMT in a MANET nodes takes the form

$$\mathbf{L} = (A, A', \omega_A), \tag{7.3}$$

where A is the index of a record bound to the leaf, A' is the *next* index, and ω_A is the record-hash for a record corresponding to index A.

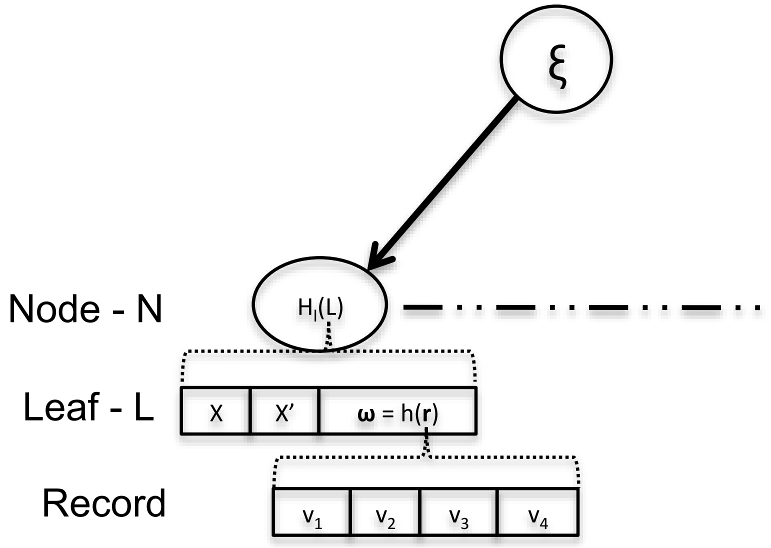


Figure 7.2

IOMT in MANET model [2]

The record ω_A (corresponding to the leaf index A) is of the form $[v_1, v_2, v_3, v_4]$, where $v_1 = 0$ signifies an empty record. Associated with a record is a *record-hash* computed as

$$\omega = h_r(v_1, v_2, v_3, v_4) = \begin{cases} h(v_1, v_2, v_3, v_4) & v_1 \neq 0 \\ 0 & v_1 = 0 \end{cases} \tag{7.4}$$

A DR is of the form $[v_1 = q, v_2 = x, v_3 = m, v_4 = n]$ and an NR is of the form $[v_1 = l, v_2 = o, v_3 = s, v_4 = 0]$ (the fourth value is not used in NRs). Two IOMT roots ξ_{dt} and ξ_{nt} (representing the DR and NR records respectively) are stored in internal registers of TMMs. Such roots can be modified due to different events using $F_{upd}()$, by providing appropriate $U1$ and $U2$ certificates. The IOMT roots can also be changed to an equivalent root (for purposes of inserting/deleting place holders in either tree).

Specifically, function $F_{ph}()$ exposed by TMMs can be used to insert or delete place holders in the DR tree with root ξ_{dt} or NR tree with root ξ_{nt}.

```
$F_{ph}(r, r', \rho)${
    IF $(\rho \neq h(EI, r, r', \chi))$ RETURN ERROR;
    IF $(r = \xi_{dt})$ $\xi_{dt} \leftarrow r'$;
    ELSE IF $(r' = \xi_{dt})$ $\xi_{dt} \leftarrow r$;
    ELSE IF $(r = \xi_{nt})$ $\xi_{nt} \leftarrow r'$;
    ELSE IF $(r' = \xi_{nt})$ $\xi_{nt} \leftarrow r$;
}
```

[2]Figure Notes: Values of the record $r = [v_1, v_2, v_3, v_4]$, for a destination record take the form of $[v_1 = q, v_2 = c, v_3 = m, v_4 = n]$, where q is the sequence number, time of expiry for the record is x, m is the hop-count and n is the next hop neighbor. A neighbor record is of the form $[v_1 = l, v_2 = o, v_3 = s, v_4 = 0]$, where l is last-heard-from time, o is the offset value, and lock s (the fourth value is always zero, and is ignored)

Form the perspective of TMMs a self-certificate satisfying $\rho = h(U1, x, \xi_{dt}, x', \xi'_{dt}, \chi)$ is proof that x is leaf node in the DR tree. Now, if there exists values (say) (A, A', ω_A) satisfying $x = H_L(A, A', \omega_A)$, the TMM concludes that ω_A is the hash of record $\mathbf{r}_A$ in the DR tree. Similarly, a self certificate satisfying $\rho = h(U2, x_f, x_g, \xi_{nt}, x'_f, x'_g, \xi'_{nt}, \chi)$ along with IOMT leaves (F, F', ω_F) and (G, G', ω_G) that are pre-images of x_f and x_g respectively, and records $\mathbf{r}_F$ and $\mathbf{r}_G$ that are pre-images of record hashes ω_F and ω_G respectively, can be provided as proof of existence of two NRs for neighbors F and G in the NT.

7.4 TMM Architecture

TMMs are resource limited modules that have only modest computational and storage abilities. They perform only simple logical operations necessary to execute $f_{dv}()$ and hash operations required to maintain IOMTs, compute pairwise secrets, and MACs.

	Non-volatile Registers
I	TMM Identity
κ	KDC secret
c	session counter
	Volatile Registers
t	Clock-tick counter
χ	Self-Secret
ξ_{dt}, ξ_{nt}	Roots of DR and NR IOMT
$\mathbf{C}$	Constants
ϑ	Constant hash
$\mathbf{E}$	Event Register $[R, y_r, t_r, a_r, D_r, \nu_r, \tilde{t}]$
O_f, O_g	Outgoing message to G and F
$\mathbf{I}$	Input Register $[(D, D', \mathbf{r}_D), \mathbf{r}'_D,$ $(F, F', \mathbf{r}_F), (G, G', \mathbf{r}_G)]$

Figure 7.3

TMM Registers.

Figure 7.3 depicts the internal registers of TMMs. Protected non-volatile registers are reserved for the TMM identity I, a secret κ issued by a trusted key distribution center (KDC), and a monotonic counter c. The self-identity I is used for creating DRs for I. Every time a DR is created, or whenever a TMM is initialized, the monotonic counter c is incremented. The secret κ is used for computing pairwise secrets shared with other TMMs.

The volatile registers include the IOMT roots ξ_{dt} and ξ_{nt}. TMMs possess a volatile clock tick counter t which can be set to any value when a TMM is powered on/initialized, and thereafter, incremented at the same rate in all TMMs. χ is the self-secret generated whenever a TMM is initialized (which is used for self-MACs). Contents of the event register $\mathbf{E}$, and input register $\mathbf{I}$ and constants $\mathbf{C}$ are used by algorithm $f_{dv}()$ to modify records in the input register, and set values O_f and O_g.

The function $F_{init}()$ initializes a TMM and sets the contents of a registers $\mathbf{C}$ and ϑ, sets the clock to a value provided by the device, initializes the roots of the NT and DT to zero, increments the monotonic counter c, and generates a new self-secret χ using a random sequence generator RSG$()$.

```
$F_{init}(t', \mathbf{C}')\{$
    $\vartheta \leftarrow h(\mathbf{C}'); \mathbf{C} \leftarrow \mathbf{C}'; \chi \leftarrow \mathrm{RSG}();$
    $\xi_{nt} \leftarrow \xi_{dt} \leftarrow 0; c \leftarrow c + 1; t = t';$
$\}$
```

7.4.1 Mutual Authentication of Messages

The secret κ_x is used by TMM X to compute shared secrets with other TMMs. Specifically, any two TMMs X and Y can using their respective KDC secrets κ_x and κ_y to compute a common secret

$$K_{xy} = h(\kappa_x, Y) \oplus \pi_{xy} = h(\kappa_y, X) \oplus \pi_{yx} \tag{7.5}$$

where π_{xy} and π_{yx} are pairwise *public* values made available to the untrusted devices that house TMM X and Y respectively. If the modified Leighton Micali scheme (MLS) [50] is used to compute the pairwise secret K_{xy} only one of the two nodes ($\bar{X}$ or $\bar{Y}$) requires access to a public value; the other node employs the public value of 0 (if $\pi_{xy} \neq 0$, then $\pi_{yx} = 0$; if $\pi_{xy} = 0$ then $\pi_{yx} \neq 0$).

The MAC secret shared between two TMMs X and Y is a function of K_{xy}, the respective monotonic counter values c_x and c_y, and a value ϑ used to initialize all TMMs taking part in the same application (for example, TMMs in all devices taking part in the same MANET). The value ϑ is a one-way function of protocol-specific constants.

The MAC secret K_{out} used by TMM X for computing MACs for outgoing messages *to* TMM Y, and the MAC secret K_{in} used by TMM X for verifying MACs for messages *from* Y are computed as

$$K_{out} = h(K_{xy}, c_x, c_y, \vartheta), K_{in} = h(K_{xy}, c_y, c_x, \vartheta), \tag{7.6}$$

If the MAC secret employed for a message is K, the MAC for the message created by X, viz., $[X, y, t, a, D_x, \nu]$ is computed as

$$\mu = h(y, t, a, D_x, \nu, K). \tag{7.7}$$

Note that as the MAC secret K if a function of ϑ, *TMMs initialized with different values of ϑ will not agree on the MAC secret (and thus cannot exchange messages).* Furthermore, as K is a function of the session counters c_x and c_y, a message generated when the counters of the sender X and receiver Y were c_x and c_y cannot be replayed if either c_x or c_y has been modified.

TMMs accept messages from other TMMs, or request from the device, and store contents of the message/request in the event register

$$\mathbf{E} = [R, y_r, t_r, a_r, D_r, \nu_r, \tilde{t}] \tag{7.8}$$

Note that apart from the contents of the message, the time at which the message was submitted is also recorded in the field $\tilde{t}$ as the "event time."

A function $F_{msg}()$ accepts messages from a neighbor M, verifies the MAC and stores the message contents in register $\mathbf{R}$. In addition, $F_{msg}()$ can also be used to create HLO messages —either on request by the device, or as an acknowledgement for a received HLO message. If $F_{msg}()$ is invoked with message source $M = 0$ this is construed as a request from the device that conveys a type of request y and a value ν. If $F_{msg}()$ is invoked with input MAC μ set to zero, this is interpreted as a request to create a HLO message for a neighbor M. Else, the contents M, t_m, a_m, M', ν of the message are verified to be consistent with the MAC μ. If consistent the message register $\mathbf{R}$ is populated. In addition, if the message is of type HLO, then a MAC for an acknowledgement message with time stamp $\tilde{t}$ is created.

```
$F_{msg}(y, M, t_m, a_m, M', \nu, \mu, p_m, c_m)\{$
  $\mu' \leftarrow 0; t' \leftarrow t;$
  IF $(M = 0)$
    $[R, y_r, t_r, a_r, D_r, \tilde{t}, \nu_r] \leftarrow [0, y, 0, 0, 0, t', \nu];$
    RETURN $t'$;
  $K = h(\kappa, M) \oplus p_m; K_{in} \leftarrow h(K, c_m, c, \vartheta); K_{out} \leftarrow h(K, c, c_m, \vartheta);$
  IF $(\mu = 0)$ $\mu' \leftarrow h(\mathrm{HLO}, \mathrm{t}', 0, 0, 0, \mathrm{K_{out}});$
  ELSE IF $(\mu = h(y, t_m, a_m, M', \nu, K_{in}))$
    $\mathbf{R} \leftarrow [M, y, t_m, a_m, M', t', \nu];$
    IF $(y = \mathrm{HLO})$ $\mu' \leftarrow \mathrm{h}(\mathrm{y}, \mathrm{t}', \mathrm{t_m}, 0, \mathrm{K_{out}});$
  ELSE RETURN ERROR;
  RETURN $t', \mu'$;
}
```

7.4.2 Updating TMM State Using $F_{upd}()$

Function $F_{msg}()$ is invoked to notify the TMM of the occurrence of an event — either a message received from another TMM, or a request from the device, and populate event register $\mathbf{E}$ in the TMM. Processing the event involves execution of $f_{dv}()$, which requires access to a currently stored DR for a destination D, a received DR for D consistent with ν_r (if the event is a DR message), currently stored NRs for up to two neighbors F and G, and protocol specific constants.

After processing the event, the end result is a change in the TMM state, and creation of up to two message (one for F and one for G).

As the response to the event may require modification of 3 records, the TMM will expect a $U1$ certificate that simultaneously certifies the integrity of the current state (before the event) and future state (after processing the event) of a DR for D, and a $U2$ certificate that simultaneously certifies the integrity of the current state (before the event) and future state (after processing the event) of two NRs (F and G). Specifically, the $U1$ certificate instructs the TMM as to how the IOMT root ξ_{dt} should be changed due to the change to the DR $\mathbf{r}_D$; the $U2$ certificate instructs the TMM as to how the IOMT root ξ_{nt} should be changed due to the change to the NRs $\mathbf{r}_F$ and $\mathbf{r}_G$.

The register **I** is used to store other inputs necessary to process an event by executing $f_{dv}()$. Specifically, the inputs include

1) a leaf from the DT IOMT for a destination D, except that instead of the third value ω_D, the pre-image $\mathbf{r}_D$ is provided as input;

2) a record $\mathbf{r}'_D$ consistent with value ν_r in the received DR message; and

3) two IOMT leaves (F and G) from the NT tree; once again, instead of the value ω, the pre-images are provided;

The contents of the input register are provided as inputs to a function $F_{upd}()$ which makes the appropriate changes to records $\mathbf{r}_D, \mathbf{r}_F$ and $\mathbf{r}_G$, accordingly modifies TMM state, and outputs MACs for messages.

Apart from contents of the input register, other inputs to $F_{upd}()$ are a) the next states ξ'_{dt} and ξ'_{dt}; and b) a $U1$ certificate ρ_d and a $U2$ certificate ρ_n.

```
$F_{upd}(D, D', \mathbf{r}_D, \mathbf{r}'_D, F, F', \mathbf{r}_F, G, G', \mathbf{r}_G,$
            $\xi'_{dt}, \xi'_{nt}, \rho_d, \rho_n, p_f, c_f, p_g, c_g)\{$
  IF $((y_r = \mathrm{DR}) \wedge (\mathrm{h_r}(\mathbf{r}'_\mathrm{D}) \neq \nu_\mathrm{r}))$ RETURN ERROR;
  $\omega_D \leftarrow h_r(\mathbf{r}_D); \omega_F \leftarrow h_r(\mathbf{r}_F); \omega_D \leftarrow h_r(\mathbf{r}_F);$
  $\mathbf{I} \leftarrow [(D, D', \mathbf{r}_D), \mathbf{r}'_D, (F, F', \mathbf{r}_F), (G, G', \mathbf{r}_G)];$
  IF $(f_{dv}() = ERROR)$ RETURN ERROR;
  $\omega'_D \leftarrow h_r(\mathbf{r}_D); \omega'_F \leftarrow h_r(\mathbf{r}_F); \omega'_D \leftarrow h_r(\mathbf{r}_F);$
  $x_d = h(D, D', \omega_D); x'_d \leftarrow h(D, D', \omega'_D);$
  IF $(\rho_d \neq h(U1, x_d, \xi_{dt}, x'_d, \xi'_{dt}, \chi)$ RETURN ERROR;
  $x_f = h(F, F', \omega_F); x'_f \leftarrow h(F, F', \omega'_F);$
  $x_g = h(G, G', \omega_G); x'_g \leftarrow h(G, G', \omega'_G);$
  IF $((\rho_n \neq h(U2, x_f, x_g, \xi_{nt}, x'_f, x'_g, \xi'_{nt}, \chi) \wedge$
     $(\rho_n \neq h(U2, x_g, x_f, \xi_{nt}, x'_g, x'_f, \xi'_{nt}, \chi))$
     RETURN ERROR;
  $\xi_{dt} \leftarrow \xi'_{dt}; \xi_{nt} \leftarrow \xi'_{nt};$
  $\mu'_f \leftarrow \mu'_g \leftarrow 0;$ //Initialize Output MACs
  IF $(O_f > 0)$
     $K \leftarrow h(\kappa, F) \oplus \pi_f; K_f \leftarrow h(K, c, c_f, \vartheta);$
  IF $(O_f = 1)$ $\mu'_f \leftarrow h(\mathrm{DR}, \tilde{\mathrm{t}}, 0, \mathrm{D}, \omega'_\mathrm{D}, \mathrm{K_f});$
  IF $(O_f = 2)$ $\mu'_f \leftarrow h(y_r, \tilde{t}, t_r, 0, 0, K_f);$
  IF $(O_f = 3)$ $\mu'_f \leftarrow h(\mathrm{DR}, \tilde{\mathrm{t}}, \mathrm{t_r}, \mathrm{D_r}, \nu_\mathrm{r}, \mathrm{K_f});$
  IF $(O_f = 4)$ $\mu'_f \leftarrow h(\mathrm{DATA}, \tilde{\mathrm{t}}, 0, \mathrm{D}, \mathrm{h}(\mathrm{I}, \nu_\mathrm{r}), \mathrm{K_f});$
  IF $(O_f = 5)$ $\mu'_f \leftarrow h(\mathrm{DATA}, \tilde{\mathrm{t}}, 0, \mathrm{D}, \nu_\mathrm{r}, \mathrm{K_f});$
  IF $(O_g > 0)$
     $K \leftarrow h(\kappa, G) \oplus \pi_g; K_g \leftarrow h(K, c, c_g, \vartheta);$
  IF $(O_g = 1)$ $\mu'_g \leftarrow h(\mathrm{DR}, \tilde{\mathrm{t}}, 0, \mathrm{D}, \omega'_\mathrm{D}, \mathrm{K_g});$
  IF $(O_g = 2)$ $\mu'_g \leftarrow h(y_r, \tilde{t}, t_r, 0, 0, K_g);$
  IF $(O_g = 3)$ $\mu'_g \leftarrow h(\mathrm{DR}, \tilde{\mathrm{t}}, \mathrm{t_r}, \mathrm{D_r}, \nu_\mathrm{r}, \mathrm{K_g});$
  IF $(O_g = 4)$ $\mu'_g \leftarrow h(\mathrm{DATA}, \tilde{\mathrm{t}}, 0, \mathrm{D}, \mathrm{h}(\mathrm{I}, \nu_\mathrm{r}), \mathrm{K_g});$
  IF $(O_g = 5)$ $\mu'_g \leftarrow h(\mathrm{DATA}, \tilde{\mathrm{t}}, 0, \mathrm{D}, \nu_\mathrm{r}, \mathrm{K_g});$
  RETURN $\mu'_f, \mu'_g;$
}
```

Figure 7.4

Updating TMM State.

Function $F_{upd}()$ verifies that if the message is of type DR, then the record $\mathbf{r}'_D$ is consistent with ν_r and notes down the current record hashes ω_D, ω_F and ω_G of the three records provided as inputs, and copies the contents of the leaves to the input register **I**.

The protocol specific function $f_{dv}()$ performs necessary modifications to the records. $f_{dv}()$ expects specific pre-conditions to be satisfied, failing which $f_{dv}()$ return error and terminates $F_{upd}()$. On successful execution of $f_{dv}()$ the DR and two NRs may be modified. In addition, $f_{dv}()$ instructs the types of messages to be sent to F and G by setting values O_f and O_g.

Till this point $F_{upd}()$ had simply assumed that the leaves for D, F and G provided as inputs were consistent with the current IOMT roots. The four values ξ'_{dt}, ρ_d and ξ'_{nt} and ρ_n simultaneously permits the TMM to verify this assumption, and modify the IOMT roots in accordance with the changes made to the DR and two NRs.

By providing contents of a leaf in the DT and two leaves in the NT, the device claims that such such leaves are consistent with the IOMT roots ξ_{dt} and ξ_{nt} respectively. Let $\omega_D = h_r(\mathbf{r}_D)$, $\omega_F = h_r(\mathbf{r}_F)$, and $\omega_G = h_r(\mathbf{r}_G)$.

Specifically, if $x_d = H_L(D, D', \omega_D)$ and $x'_d = H_L(D, D', \omega'_D)$ (where ω'_D is the record hash after modification of the record in response to the event) then the certificate ρ_d should satisfy

$$\rho_d = h(U1, x, \xi_{dt}, x', \xi'_{dt}, \chi). \tag{7.9}$$

Similarly, if the leaf hashes corresponding to F and G are x_f and x_g before the event, and x'_f and x'_g after the event, the certificate ρ_n should satisfy

$$\rho_d = \begin{cases} h(U2, x_f, x_g, \xi_{nt}, x'_f, x'_g, \xi'_{nt}, \chi) \text{ or} \\ h(U2, x_g, x_f, \xi_{nt}, x'_g, x'_f, \xi'_{nt}, \chi) \end{cases} \tag{7.10}$$

If the preconditions (before execution of $f_{dv}()$) and post conditions (after execution) are consistent with the current state (ξ_{dt}, ξ_{nt}) and the next state (ξ'_{dt}, ξ'_{nt}) the IOMT roots are modified to ξ'_{dt}, ξ'_{nt}.

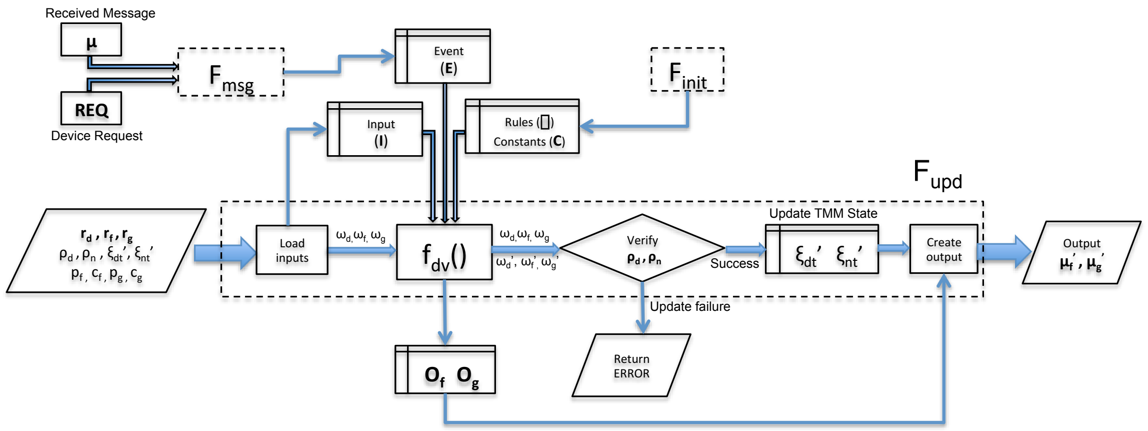


Figure 7.5

Overall operation of TCB in MANET model.

Finally, output messages are created depending on the values O_f and O_g. Inputs p_f and c_f (p_g and c_g) to $F_{upd}()$ are necessary for computing the MAC secret to be used for the message to F (G). Recall that $O = 1, 4, 5$ are spontaneous (not ACK) messages: $O = 1$ instructs creation of a DR message. $O = 4$ instructs creation of a DATA message to initiate a data packet to D. $O = 5$ instructs creation of a DATA message to *relay* a received DATA message to the next hop. $O = 2, 3$ are acknowledgments; $O = 2$ is a simple acknowledgement; $O = 3$ is a DR message that simultaneously acknowledges a received DR/DATA message and conveys a DR.

Figure 7.5 depicts overall operation of the system-specific TCB functions. Events are triggered by either a received message $\boldsymbol{\mu}$ or a request $\mathbf{REQ}$ (from the device) submitted to F_{msg} which populates the event register E. Function $F_{upd}()$ is then invoked to process the event (and change state and/or create messages). The inputs provided to $F_{upd}()$ are $\mathbf{r_d}$ (DR record), $\mathbf{r_f}$ (NR record of F), $\mathbf{r_g}$ (NR record of G), update certificates (ρ_d and ρ_n) along with next next states (ξ'_{dt} and ξ'_{nt}), and values p_f, c_f, p_g, and c_g necessary for computing MACs for verification by F and G. The inputs are loaded onto register I, and hash of the current records (DR and 2 NR) are computed (ω_d, ω_f and ω_g). All inputs necessary for the protocol specific function $f_{dv}()$ are now available in registers I (Input), E (Event), ϑ (Rules) and C (Constants). Function $f_{dv}()$ updates the records in place (according to protocol-specific rules) and also sets the output registers (O_f, O_g).

Function $F_{upd}()$ computes the new record hashes as ω'_d, ω'_f and ω'_g. Current state (ω_d, ω_f, ω_g, ξ_{dt}, ξ_{nt}) and the next state (ω'_d, ω'_f, ω'_g, ξ'_{dt}, ξ'_{nt}) are verified using certificates ρ_d and ρ_n. If successful, TMM states are updated to ξ'_{dt} and ξ'_{nt}. O_f and O_g direct the output generation for F and G in the form of MACs μ'_f and μ'_g.

7.4.3 Deploying TMM Based MANETs

For MANET secured using TMMs every device has a TMM. A (perhaps off-line) administrator/operator of the MANET specifies the constants to be used. The off-line administrator is also responsible for ensuring that every device has access to public values necessary to facilitate computation of pairwise secrets between TMMs in devices (see Section 3.1).

To operate in a MANET the device is required to initialize it's TMM. At this point the device has no records, and the IOMT roots are zero. From this point onwards the device has to maintain it's DT/NT and the corresponding IOMTs to be in sync with the roots stored inside the TMM. Any number of place holders can be added in either tree by using certificate generation functions to create equivalence certificates and using $F_{ph}()$ to modify the IOMT roots accordingly.

As soon as a TMM is initialized the device can use $F_{msg}()$ to perform handshakes with TMMs in neighbors, and then invoke $F_{upd}()$ to insert neighbor records. Once a TMM recognizes neighbors, the TMMs own DRs can be incremented, and sent to active neighbors.

Whenever a message is received from a neighbor the device submits the message to it's TMM using $F_{msg}()$ and invokes $F_{upd}()$ to modify the TMM state / create messages. Before $F_{upd}()$ is invoked, as the device is aware of the precise changes that should be made to a DR and two NRs, the device can use IOMT functions to generate the necessary $U1$ and $U2$ certificates.

The broad assurance that all devices taking part in a MANET will modify DRs and NRs in exactly the same manner as dictated by the protocol (algorithm $f_{dv}()$) is enforced as follows:

1. Only TMMs initialized with the same set of protocol-specific constants will agree on a pairwise secret. This feature is used to assure the integrity of protocol-specific constants.
2. IOMTs are used to ensure that only DRs/NRs consistent with the IOMT roots stored inside the TMM will be considered as valid;
3. As the function $f_{dv}()$ cannot be modified, the DR/NRs consistent with the current IOMT roots can be modified only according to the algorithm $f_{dv}()$.

4. After modification of the records the device is forced to modify it's copy of the records in the same way.
5. If messages mandated by $f_{dv}()$ are not delivered by the device, the device cannot retain the intended recipients of such messages as neighbors.

7.5 Related Work

Devices taking part in MANET offer an enormous attack surface that can be exploited by attackers. Specifically, hidden malicious functionality in component of the MANET device could be exploited to illegally modify a) the MANET software, or b) the DT/NT or the device's clock, or expose secrets protected by the device to advertise arbitrary routing information, and/or construct clone devices with arbitrary functionality. Current efforts to secure MANETs simply ignore a wide range of attacks stemming from such issues. In the proposed approach to secure MANETs all such attacks are side stepped as no component of the device is trusted in the first place.

Prior efforts in the literature that attempt to secure MANET by leveraging a trustworthy computing module include [61], [29] that attempt to offer some assurances regarding the medium access control layer; the "nuglets of currency" approach in [28]; the Bloom filter [9] approach in [23], and the predecessor of the current work in [63].

In [61], the wireless transceiver is assumed to be inside a trusted boundary; in [29] the wireless driver is executed inside a trusted boundary. In [28], nuglets of currency are maintained inside the trusted boundary; they are incremented / decremented based on selfish/selfless acts performed by the device. However, [28] does not address specific mechanisms that will enable a trusted module to unambiguously infer that the associated device has indeed performed a selfish/selfless task.

Gaines et al. [23] argued that mechanisms to guarantee integrity of MANET require resource limited trusted modules to be able to vouch for the integrity of destination table and neighbor table stored by the device. They also recognized the need for the trusted module to identify non existence of routing information. For this purpose, [23] suggested the use of Bloom filters to store succinct summaries of routing records.

In [63] the authors employed an Index Ordered Merkle Tree (IOMT) for keeping track of routing records. The specific improvements in our approach compared to [63] are as follows:

1. In [63] the authors assumed that TMMs have synchronized clocks before they can take part in a MANET; we do not make this assumption.
2. In [63] the NT is stored entirely inside the TMM; to eliminate restrictions on the NT size the NT is also stored outside the TMM.
3. In [63] the goal is merely to ensure integrity of the destination table: no effort was made to ensure that data packets will be relayed only in a manner consistent with the DT/NT.
4. In [63] no strategy was proposed to enable TMMs to verify that devices indeed deliver TMM messages. This issue was addressed here by using a locking mechanism (setting a field s in an NR) to ensure reliable delivery (as failure to deliver/accept messages will result in the inability of the device to use the link for some period into the future).

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

With increasing complexity of hardware and software components, the presence of undesired functionality cannot be ruled out. As it is far from practical to validate the integrity of all components of a system, we adopted an alternative approach to secure systems where any subsystem is associated with a trustworthy security kernel that enforces of system specific rules to be adhered to by the subsystem.

A crucial step in assuring the integrity of any subsystem (belonging to any system) is assuring the integrity of dynamic data entrusted to the subsystem. In the approach adopted in this dissertation all dynamic databases associated with a subsystem are succinctly captured as roots of one or more OMTs. The OMT roots can be seen as the *state* of the subsystem.

From this perspective, the tasks performed by the security kernel is to ensure the consistency of system-specific messages created by a security kernel with it's current state, and ensuring that the receiver of the message (the security kernel of another subsystem of the same system) modifies it's state in a manner consistent with system-specific rules.

For the systems considered in this dissertation,

1. File storage servers need to assure the integrity of different versions of different files, the ACLs for files, and protect the privacy of file encryption secrets.

2. Central servers in content distribution systems need to protect the integrity of content provided by publishers, the ACL, and the privacy of content encryption secrets

3. Lookup servers need to protect the integrity of dynamic records and ensure that only correct answers are provided for any query.

4. Different subsystems in BGP maintain different databases. IP/AS Registry needs to assure the integrity of IP prefix and AS number assignments to various autonomous systems. AS owners need to ensure that only owned prefixes can be delegated to their BGP speakers. BGP speakers maintain several nested databases to enable identification of path (to any prefix) with the best weight, to to ensure the integrity of the path vector, and to ensure that all policies and preferences specified by the AS owner are adhered to.

5. Devices taking part in MANET are required to ensure the integrity of the destination table and neighbor table (as data packets for any destination can be sent only to an active neighbor which is also the next hop in the best path to the destination).

As illustrated by the designs of security kernels for such substantially different systems (in Chapters 4,5,6,7) it is possible to efficiently utilize Ordered Merkle Trees. By using a set of generalized security kernel functionality for maintaining OMTs (verification of integrity of leaf nodes and insertion/deletion of place-holders), and simple system-specific functions for *updating* OMTs can be adapted to realize assurances for a wide range of systems.

The security kernel for remote file storage system leverages an IOMT to provide necessary assurances like integrity, privacy and availability of files. The security kernel for the look-up server and content distribution server employed IOMTs to ensure integrity and availability of records. Specifically, in file storage systems and CDS, IOMTs were used for representing files *and* ACLs. In MANETs, IOMTs were used to maintain a destination table and a neighbor table. BGP security kernels employed both IOMTs and ROMTs. IOMTs were used to maintain the RIDB, maintain ordered list of weights, and as a succinct representation of the path vector. ROMTs were used to store ranges of IP addresses. Simply by using place-holder operations it is possible to ensure that only owned subset of IP address ranges can be delegated, and that only aggregate-able addresses can be aggregated.

8.1 Contributions of the Research

Apart from overcoming the limitations of current approaches to securing systems (viz., ignoring the issue of hidden malicious functionality) the proposed security measures provide substantially improved scope of assurances. For example, current security measures for remote file storage systems do not cater for authenticated denial of existence. Apart from catering for authenticated denial, the proposed security measures also ensure that no information that is not explicitly queried need to be revealed to prevent attacks similar to DNS-walk [67], [33].

None of the current approaches for securing routing protocols — either in MANETs or BGP routing for the Internet, guarantee that only the best path will be advertised, or that all rules that govern the protocol will be adhered to. The current approaches for securing BGP do not assure the integrity of aggregated prefixes, which is an especially severe limitation that is overcome by the proposed approach. The work done towards securing Mobile Ad-Hoc network employed a very generic model to enforce protocol-specific rules, which could be easily adapted to other protocols (by specifying the set of rules as a function that replaces $f_{dv}()$).

The following publications resulted from the work carried out towards this dissertation:

- Securing File Storage in an Untrusted Server - Using a Minimal Trusted Computing Base, *CLOSER*, 2011 [42].

- An Efficient TCB for a Generic Content Distribution System, *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2012.[43]
- An Efficient TCB for a Generic Data Dissemination System, *International Conference on Communications in China: Communications Theory and Security (CTS)*, ICCC12-CTS, [65].
- Realizing a Secure File Storage Service, *submitted for IEEE Transactions on Reliability - Special Section on Information, System, and Software Assurance: Research & Practice* (March 2013) [40].
- An Efficient Trusted Computing Base for MANET Security, *submitted for IEEE Transactions on Mobile Computing*, (May 2013) [41].
- Ordered Merkle Tree: A Dynamic Authenticated Data Structure for Security Kernels, *submitted for IEEE Transactions on Dependable and Secure Computing*, (May 2013) [39].

8.2 Future Work

The proposed research is a first step towards arriving at a specification for a *universal security kernel* or a *universal TCB* that can be used for *any* subsystem/system. The advantage of an universal TCB is that trustworthy chips with fixed TCB functionality can be mass produced at low cost. Furthermore, the overhead for establishing a trusted infrastructure for verification of integrity and certification of such chips will also be low if all chips to be tested have identical functionality.

Currently we expect the functional components of such a chip to include the generic OMT functionality and mutual authentication functionality discussed in Section 3.4 and Section 3.1. Realization of a universal TCB demands efficient strategies for generalizing of subsystem-specific functionality. Some of the work towards reducing such system-specific functionality was attempted in our approach to secure MANETs, where protocol specific rules were confined to a simple algorithm.

REFERENCES

[1] "TCG Specification: Architecture Overview, Specification Revision 1.4," August 2007.

[2] "Building a Secure Platform for Trustworthy Computing," *White paper, Microsoft Corporation*, Dec. 2002.

[3] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed I/O," *Intel technology journal*, vol. 10, no. 3, 2006, pp. 179–192.

[4] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent Authenticated Dictionaries and Their Applications," *Proceedings of the 4th International Conference on Information Security*, London, UK, UK, 2001, ISC '01, pp. 379–393, Springer-Verlag.

[5] W. A. Arbaugh, D. J. Farbert, and J. M. Smith, "A Secure and Reliable Bootstrap Architecture," *IN PROCEEDINGS OF THE 1997 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*. 1997, pp. 65–71, IEEE Computer Society.

[6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," *IETF RFC 4033*, Mar. 2005.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and The Art of Virtualization," *Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003, SOSP '03, pp. 164–177, ACM.

[8] T. Bates, R. Chandra, D. Katz, and Y. Rekhter, *Multiprotocol extensions for BGP-4*, Tech. Rep., RFC 2283, 1998.

[9] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, July 1970, pp. 422–426.

[10] S. Bratus, E. Sparks, and S. W. Smith, "TOCTOU, Traps, and Trusted Computing," *In Trust 08: Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies*, 2008, pp. 14–32.

[11] E. Bugnion, S. Devine, and M. Rosenblum, “DISCO: Running Commodity Operating Systems on Scalable Multiprocessors,” *ACM Transactions on Computer Systems*, 1997, pp. 143–156.

[12] A. Buldas, P. Laud, and H. Lipmaa, “Accountable Certificate Management Using Undeniable Attestations,” *Proceedings of the 7th ACM conference on Computer and communications security*, New York, NY, USA, 2000, CCS ’00, pp. 9–17, ACM.

[13] D. Davis and R. Swick, “Network Security via Private-Key Certificates,” *SIGOPS Oper. Syst. Rev.*, vol. 24, no. 4, Sept. 1990, pp. 64–67.

[14] P. T. Devanbu, M. Gertz, C. U. Martel, and S. G. Stubblebine, “Authentic Third-party Data Publication,” *Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security: Data and Application Security, Development and Directions*, Deventer, The Netherlands, The Netherlands, 2001, pp. 101–112, Kluwer, B.V.

[15] A. M. Devices, *AMD64 architecture programmers manual: Volume 2: System programming*, AMD, December 2005.

[16] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, 1976, pp. 644–654.

[17] M. V. Dijk, L. F. G. Sarmenta, C. W. Odonnell, and S. Devadas, *Proof of Freshness: How to efficiently use on online single secure clock to secure shared untrusted memory*, Tech. Rep., 2006.

[18] M. V. Dijk, L. F. G. Sarmenta, J. Rhodes, and S. Devadas, *Securing Shared Untrusted Storage by using TPM 1.2 Without Requiring a Trusted OS*, Tech. Rep., 2007.

[19] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. Van Doorn, and S. W. Smith, “Building the IBM 4758 secure coprocessor,” *Computer*, vol. 34, no. 10, 2001, pp. 57–66.

[20] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” *Advances in Cryptology*. Springer, 1985, pp. 10–18.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, 2003, pp. 114–131.

[22] A. Fiat and M. Naor, “Broadcast encryption,” *Advances in CryptologyCRYPTO93*. Springer, 1994, pp. 480–491.

[23] B. Gaines and M. Ramkumar, “A Framework for Dual-Agent MANET Routing Protocols,” *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, 2008, pp. 1–6.

[24] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," *Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003, SOSP '03, pp. 193–206, ACM.

[25] R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, and T. Rabin, "Algorithmic tamper-proof (ATP) security: Theoretical foundations for security against hardware tampering," *Theory of Cryptography*, Springer, 2004, pp. 258–277.

[26] M. Goodrich, R. Tamassia, and A. Schwerin, "Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing," *DARPA Information Survivability Conference Exposition II, 2001. DISCEX '01. Proceedings*, 2001, vol. 2, pp. 68 –82 vol.2.

[27] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen, "Authenticated data structures for graph and geometric searching," *Topics in CryptologyCT-RSA 2003*, Springer, 2003, pp. 295–313.

[28] J.-P. Hubaux, L. Buttyán, and S. Capkun, "The quest for security in mobile ad hoc networks," *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, New York, NY, USA, 2001, MobiHoc '01, pp. 146–155, ACM.

[29] M. Jarrett and P. Ward, "Trusted Computing for Protecting Ad-hoc Routing," *Proceedings of the 4th Annual Communication Networks and Services Research Conference*, Washington, DC, USA, 2006, CNSR '06, pp. 61–68, IEEE Computer Society.

[30] B. Kauer, "OSLO: Improving the Security of Trusted Computing," *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, Berkeley, CA, USA, 2007, pp. 16:1–16:9, USENIX Association.

[31] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (S-BGP)," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 4, 2000, pp. 582–592.

[32] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, 1992, pp. 265–310.

[33] B. Laurie, G. Sisson, R. Arends, D. Blacka, et al., *DNS security (DNSSEC) Hashed Authenticated Denial of Existence*, Tech. Rep., RFC 5155, February, 2008.

[34] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an Untrusted Operating System on Trusted Hardware," *ACM SIGOPS Operating Systems Review*. ACM, 2003, vol. 37, pp. 178–192.

[35] U. Maheshwari, R. Vingralek, and W. Shapiro, "How to Build a Trusted Database System on Untrusted Storage," *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, Berkeley, CA, USA, 2000, OSDI'00, pp. 10–10, USENIX Association.

[36] C. Martel, G. Nuckolls, M. Gertz, P. Devanbu, A. Kwong, and S. G. Stubblebine, "A General Model for Authentic Data Publication," *Algorithmica*, 2004.

[37] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for tcb minimization," *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, New York, NY, USA, 2008, Eurosys '08, pp. 315–328, ACM.

[38] R. C. Merkle, "Protocols for Public Key Cryptosystems," *IEEE Symposium on Security and Privacy*, 1980, p. 122.

[39] S. Mohanty and M. Ramkumar, "Ordered Merkle Tree: A Dynamic Authenticated Data Structure for Security Kernels," submitted for IEEE Transactions on Dependable and Secure Computing, 2013.

[40] S. Mohanty and M. Ramkumar, "Realizing a Secure File Storage Service," submitted for IEEE Transactions on Reliability - Special Section on Information, System, and Software Assurance: Research & Practice, 2013.

[41] S. Mohanty, V. Thotakura, and M. Ramkumar, "An Efficient Trusted Computing Base for MANET Security," submitted for IEEE Transactions on Mobile Computing, 2013.

[42] S. D. Mohanty and M. Ramkumar, "Securing File Storage in an Untrusted Server - Using a Minimal Trusted Computing Base," *CLOSER*, 2011, pp. 460–470.

[43] S. D. Mohanty, A. Velagapalli, and M. Ramkumar, "An Efficient TCB for a Generic Content Distribution System," *2012 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2012, pp. 5–12.

[44] C. Mundie, P. de Vries, P. Haynes, and M. Corwine, "Trustworthy computing," *Microsoft white paper*, vol. 10, 2002.

[45] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *ACM Transactions on Storage (TOS)*, vol. 2, no. 2, 2006, pp. 107–138.

[46] D. Noar, M. Noar, and J. Lotspiech, "Revocation and tracing routines for stateless receivers," *Lecture Notes in Computer Science, Advances in Cryptology, Springer-Verlag*, vol. 2139, 2001.

[47] M. Peinado, Y. Chen, P. Engl, and J. Manferdelli, "NGSCB: A Trusted Open System," *In Proceedings of 9th Australasian Conference on Information Security and Privacy ACISP*. 2004, pp. 86–97, Springer.

[48] C. Perkins and E. Royer, "Ad-hoc on-demand distance vector routing," *Second IEEE Workshop on Mobile Computing Systems and Applications*, 1999, pp. 90–100.

[49] C. E. Perkins and P. Bhagwat, "Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers," *Proceedings of the conference on Communications architectures, protocols and applications*, New York, NY, USA, 1994, SIGCOMM '94, pp. 234–244, ACM.

[50] M. Ramkumar, "On the scalability of a "non-scalable" key distribution scheme," *IEEE SPAWN, Newport Beach, CA*, 2008.

[51] M. Ramkumar, "Trustworthy Computing Under Resource Constraints With the DOWN Policy," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 1, 2008, pp. 49–61.

[52] Y. Rekhter and P. Gross, *Application of the border gateway protocol in the internet*, Tech. Rep., RFC 1772, 1995.

[53] Y. Rekhter and T. Li, *A border gateway protocol 4 (BGP-4)*, Tech. Rep., RFC 1771, 1995.

[54] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, 1978, pp. 120–126.

[55] E. M. Royer and C. K. Toh, "A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks," *IEEE Personal Communications*, vol. 6, 1999, pp. 46–55.

[56] D. Safford, "Clarifying misinformation on TCPA," *White paper, IBM Research*, vol. 30, 2002.

[57] D. Safford, "The need for TCPA," *White paper, IBM Research*, 2002.

[58] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," *Proceedings of the 13th conference on USENIX Security Symposium*, 2004, vol. 13, pp. 16–16.

[59] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS," *Proceedings of the first ACM workshop on Scalable trusted computing*, New York, NY, USA, 2006, STC '06, pp. 27–42, ACM.

[60] S. Smith, *Trusted Computing Platforms: Design and Applications*, Springer Verlag, 2005.

[61] J.-H. Song, V. Wong, V. Leung, and Y. Kawamoto, "Secure routing with tamper resistant module for mobile Ad hoc networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 7, no. 3, July 2003, pp. 48–49.

[62] E. R. Sparks, "A Security Assessment of Trusted Platform Modules Computer Science Technical Report," *Power*, 2007, pp. 1–29.

[63] V. Thotakura and M. Ramkumar, "Minimal trusted computing base for MANET nodes," *2010 IEEE 6th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, oct. 2010, pp. 91–99.

[64] P. C. Van Oorschot, A. Somayaji, and G. Wurster, "Hardware-assisted circumvention of self-hashing software tamper resistance," *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, 2005, pp. 82–92.

[65] A. Velagapalli, S. Mohanty, and M. Ramkumar, "An Efficient TCB for a Generic Data Dissemination System," *International Conference on Communications in China: Communications Theory and Security (ICCC'12–CTS)*. IEEE, 2012.

[66] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf, "Security issues and requirements for internet-scale publish-subscribe systems," *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*. IEEE, 2002, pp. 3940–3947.

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