4-30-2011

Trustworthy Computing Approach for Securing Ad Hoc Routing Protocols

Vinay Thotakura

TRUSTWORTHY COMPUTING APPROACH FOR SECURING AD HOC ROUTING PROTOCOLS

By

Vinay Thotakura

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

April 2011

TRUSTWORTHY COMPUTING APPROACH FOR SECURING AD HOC ROUTING PROTOCOLS

By

Vinay Thotakura

Approved:

Mahalingam Ramkumar Associate Professor of Computer Science and Engineering (Major Professor)	Rayford B. Vaughn Bill and Carolyn Cobb Professor of Computer Science and Engineering Associate Vice President for Research (Committee Member)
David A. Dampier Associate Professor of Computer Science and Engineering (Committee Member)	Yoginder S. Dandass Associate Professor of Computer Science and Engineering (Committee Member)
Edward B. Allen Associate Professor and Graduate Coordinator Computer Science and Engineering	Dr. Sarah A. Rajala Dean James Worth Bagley College of Engineering

Name: Vinay Thotakura

Date of Degree: April 29, 2011

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Mahalingam Ramkumar

Title of Study: TRUSTWORTHY COMPUTING APPROACH FOR SECURING AD HOC ROUTING PROTOCOLS

Pages in Study: 154

Candidate for Degree of Doctor of Philosophy

Nodes taking part in mobile ad hoc networks (MANET) are expected to adhere to the rules dictated by the routing protocol employed in the subnet. Secure routing protocols attempt to reduce the ill-effect of nodes under the control of malicious entities who deliberately violate the protocol. Most secure routing protocols are reactive strategies which include elements like redundancies and cryptographic authentication to detect inconsistencies in routing data advertised by nodes, and perhaps explicit measures to react to detected inconsistencies.

The approach presented in this dissertation is a proactive approach motivated by the question "what is a minimal trusted computing base for a MANET node?" Specifically, the goal of the research was to identify a small set of well-defined low-complexity functions, simple enough to be executed inside highly resource limited trusted boundaries, which can ensure that nodes will not be able to violate the protocol.

In the proposed approach every node is assumed to possess a low complexity trusted MANET module (TMM). Only the TMM in a node is trusted - all other hardware and software are assumed to be untrusted or even hostile. TMMs offer a set of interfaces to the untrusted node housing the TMM, using which the node can submit data to the TMM for cryptographic verification and authentication. As other nodes will not accept packets that are not authenticated by TMMs, the untrusted node is forced to submit any data that it desires to advertise, to its TMM. TMMs will authenticate data only if the untrusted node can convince the TMM of the validity of the data.

The operations performed by TMMs are to accept, verify, validate data submitted by the untrusted node, and authenticate such data to TMMs housed in other nodes. We enumerate various TMM interfaces and provide a concrete description of the functionality behind the interfaces for popular ad hoc routing protocols.

DEDICATION

To my parents T. Suresh Babu, Ch. Vani, and my sister T. Sravanthi, who have given me immense love, care and encouragement that has guided me this far.

ACKNOWLEDGMENTS

When I first arrived in America, I was extremely excited, and to be frank was even nervous on how I would pursue my higher studies in a completely different country. I was also equally anxious to find out what future had in store for me in this new place. Today, when I look back, I can confidently say that five years back I had taken the best decision ever - “to get a higher degree in USA”. Now that I am recollecting my journey in USA, I would like to thank few people who played a commendable role in shaping me the way I am today.

Firstly, I would like to extend my earnest and humble gratitude to my mentor and PhD advisor Dr. Mahalingam Ramkumar. He has taught me the very basics of conducting research. I really could not have asked for a better advisor. I was heavily influenced by his way of approaching complex research problems, and the manner in which he analyses a given solution. He always created a friendly work environment, and was very patient in clarifying my questions. I am very thankful for the financial support and research opportunities that he has provided me during my tenure at MSU.

I would like to extend my thanks to Dr. Rayford A. Vaughn, Dr. David A. Dampier, and Dr. Yoginder Dandass for serving as members in my PhD Dissertation Committee, and for their insightful suggestions and feedback.

I would also like to thank the Department of Computer Science and Engineering at Mississippi State University for assisting me with the much required financial support. I would like to thank the departmental staff Ms. Shonda Cumberland, Ms. Jo Coleson, Ms. Courtney Blaylock, and Ms. Keri Chisolm for their support.

I also would like to extend my appreciations to my seniors K.A. Sivakumar and K. Anil who helped me to settle down during my initial days. In this regard I also would like to thank my friends Nandeesh, Sam, and Srinath for their support and encouragement.

Finally, I once again thank my parents and my sister for their continuous love and affection.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE

MANET	Mobile Ad hoc Networks
DSR	Dynamic Source Routing
AODV	Ad hoc On Demand Routing
DSDV	Dynamic Sequenced Distance Vector
TORA	Temporally Ordered Routing Algorithm
CR-TORA	Collision Resistant TORA
RREQ	Route Request
RREP	Route Response
RERR	Route Error
CLR	Clear Packet
UPD	Update Packet
OPT	Optimization Packet
DR	Destination Record
MAC	Message Authentication Code
KDC	Key Distribution Center
TMM	Trusted MANET Module
IOMT	Index Ordered Merkle Tree

$h()$	Hash function
K_X	Secret shared between TMM X and KDC
S_X	Secret known only to TMM X
t_x	Current time as seen by node X
K_{pw}	Shared secret between two TMMs
K_{AB}	Shared secret between TMM A and TMM B
P_{AB}	Public value shared between TMM A and TMM B
$\mathcal{N}$	Neighbor Table which is stored inside TMM
$\mathbf{N}$	A Neighbor record in neighbor table $\mathcal{N}$
id_n	Identity of a neighbor node
id_p	Identity of the provider
t'	Time when the latest authenticated MAC was received
l	Status of the neighbor
$\mathbf{A}$	An authentication Record
ACK	Flag for acknowledgment
μ_r	MAC used to authenticate a DR
μ_t	MAC used to authenticate a time-stamp
μ_a	MAC for an acknowledgment
μ_s	self-MAC used to authenticate a received DR
μ_{aux}	self-MAC used to denote information on stored source path
μ_{set}	self-MAC to denote set operations

$\mathbf{D}$	A destination record
q	Sequence Number sent in a DR
m	Hop Count sent in a DR
a	Auxiliary value sent in a DR
τ	Validity time included in DR
δ_n	Maximum age of neighbor time-stamp
Δ	Validity period for destination records
Q_x	Current sequence number of TMM X
MAX	Maximum size of neighbor table
INF	Hop count considered as unreachable
Δ_W	Waiting period
ω	specifies the protocol in use
r	Root of main IOMT
r_a	Root of auxiliary IOMT
$\mathbf{L}$	Leaf of IOMT
$\mathbf{v}$	Instructions used to map a leaf to the root
θ	Information about the identity represented by a leaf
a_m	Cumulative hash of a source path
Opt	Flag that denotes a set operation
d	Number of downstream links
u	cumulative hash of UPD-List

CHAPTER 1

INTRODUCTION

Wireless nodes forming mobile ad hoc networks (MANET) adhere to MANET routing protocols to route packets amongst each other. Design of MANET protocols is substantially more challenging compared to routing protocols for wired networks due to a variety of reasons like i) mobility of nodes leading to more frequent topological changes; ii) resource limitations inherent to mobile battery operated devices; and iii) several issues arising from using wireless instead of wired links between nodes. Perhaps one of the biggest challenge in the design of MANET protocols stems from the fact that mobile devices (acting as routers and relaying packets on behalf of other nodes) may be under the control untrusted or even malicious entities.

1.1 Secure MANET Protocols

Early MANET routing protocols simply assume that all nodes will follow the rules dictated by the protocol. Rules governing secure MANET routing protocols include additional elements aimed at improving the resilience of MANET subnets to non cooperative nodes. The additional elements in secure MANET protocols can be broadly classified into four categories:

1. redundancy in routing information;
2. cryptographic authentication;

3. monitoring neighbors for possible violations; and
4. using trustworthy component(s) in MANET nodes.

Several secure extensions of MANET routing protocols like dynamic source routing (DSR) [1], ad hoc on-demand distance vector (AODV) [2], and destination sequenced distance vector (DSDV) protocol [4] have been proposed in the literature. However, most secure MANET protocols address only a subset of the four elements. Such independent efforts cannot be combined to build a comprehensive secure routing protocol due to the interdependencies between the different elements. For example,

1. the nature of cryptographic authentication can affect the ability to monitor;
2. the use of trustworthy components can reduce the redundancy required to provide the same assurances; and
3. the redundancies employed can affect the choice of cryptographic authentication schemes.

1.2 Goals and Hypothesis

Securing MANET routing protocols involves providing some tangible assurances that nodes will abide by rules. Realizing assurances towards securing any system is achieved by amplifying the trust in a trusted computing base (TCB) [7]. Most strategies to secure MANETs employ cryptographic authentication of routing data to limit an attackers ability to disseminate inconsistent routing information. The TCB for facilitating cryptographic authentication includes a set of cryptographic algorithms (which are assumed to be unbreakable), and a trusted authority (TA) who distributes cryptographic material to all nodes of a MANET network. Secure MANET protocols that leverage this limited TCB i)

fail to provide some important assurances, and ii) typically impose substantial overhead for resource limited battery operated mobile devices.

The primary goal of the research carried out for this dissertation is the identification of a TCB for MANET nodes to improve the performance of MANETs by

1. providing assurances that reduce the scope of attacks while preserving the optimizations offered by the routing protocol, and
2. by reducing the overhead required for leveraging the TCB.

It is assumed that the TCB functions are executed inside trustworthy MANET modules (TMM) housed in every MANET node; only the TMMs are trusted; the rest of the node - all other hardware and software - are untrusted.

An important prerequisite for a trustworthy module to *warrant* trust is that the TCB functions executed inside the module are simple, and consequently, easily verifiable. Simple TCB functions can also be implemented as hardwired logic (software-free), thus rendering moot a wide range of attacks that attempt to modify software. It is also desirable that the modules consume as little power as possible, and consequently disseminate negligible heat, as such modules can be physically well shielded from deliberate and accidental intrusions. With these self-imposed limitations on TCB functions aimed at improving the reliability of TMMs while simultaneously lowering their cost, we seek a set of simple MANET TCB functions.

The research hypothesis is that it is possible to design MANET TCB functions using only logical and cryptographic hash operations, and that the TCB functions can be leveraged

1. to ensure that nodes cannot advertise routing information that is inconsistent with information assimilated from other nodes, and
2. to provide several useful assurances, including several assurances that are not provided by current secure MANET protocols.

1.3 Specific Contributions of this Dissertation

As several MANET routing strategies exist, the choice of the MANET routing protocol for an ad hoc subnet scenario should consider several factors like subnet size, density, mobility etc. For example, on-demand protocols like dynamic source routing (DSR), ad hoc on-demand distance vector (AODV) are widely regarded as well suited for scenarios involving larger subnet sizes; the proactive dynamic sequence distance vector (DSDV) protocol may be better suited for smaller subnet sizes. While most MANET protocols assume that any pair of nodes in a subnet is equally likely to communicate, temporally ordered routing algorithm (TORA) was designed explicitly for scenarios where most or all nodes in the subnet communicate with one or a few "base-stations."

While no *one-size-fits-all* MANET protocol exists, three protocols - DSR, AODV and TORA - can together cater for the needs of most MANET subnets - especially since DSDV can be seen as a special case of TORA. For this reason our research was restricted to identifying a TCB for DSR, AODV and TORA.

The first contribution of this dissertation is a new protocol - collision resistant TORA - designed to offset many of the limitations of TORA. Specifically, CR-TORA addresses two fundamental limitations of TORA - TORA's susceptibility to collisions, and that TORA is

difficult to secure due to complex rules that govern the "height of a node." The results of this research has been published in a Journal article [8].

The second contribution is the specific functionality of TMMs necessary to secure on-demand protocols like DSR and AODV to provide significantly enhanced assurances compared to existing protocols. Specifically, current secure on-demand protocols which simply strive to ensure the integrity of hop-count (in AODV) or path vector (in DSR) information by carrying over authentication, are susceptible to colluding nodes. Furthermore, such protocols do not include explicit measures to i) inhibit supercilious route error packets; and ii) ensure that routing data will be accepted from (and relayed to) only nodes with tested bidirectional links. The research results have been published in a recent conference article [9].

The third contribution of this dissertation is a useful data structure, an index-ordered merkle tree (IOMT) as an extension of the popular merkle hash tree. A merkle hash tree permits a resource starved trusted entity "virtually store" a large database of records by storing a single value - the root of the tree - inside the trusted boundary. The dynamic records of the database are the leaves of a binary tree.

For true virtual storage of a large number of data records it is not sufficient for the trusted entity to be capable of determining if a record (leaf) is indeed a part of the tree. An important requirement is also the ability to determine that a record for some index is *not* a part of the tree (does not exist in the database). An IOMT is an extension of the merkle tree which is capable of verifying such negative queries efficiently.

The fourth contribution of this dissertation is an extension of the TMM based approach for securing AODV and DSR to realize some additional useful assurances (beyond the assurances provided in [9]), by employing an IOMT for storing routing records. The novel IOMT, and a strategy to leverage the IOMT to secure AODV was presented in a recent conference [10]. In addition to the TCB functions required for maintaining a routing record for each destination, for securing DSR it is necessary to maintain a database of paths which can be used to propagate route requests. An expanded TCB capable of securing both AODV and DSR has been submitted to a journal [11].

The fifth contribution of this dissertation is the identification of an efficient TCB for securing CR-TORA. One of the main difference between securing AODV vs securing CR-TORA stems from the fact that in the former (AODV) a node needs to maintain only one routing record for every possible destination. However, in CR-TORA a node needs to maintain a record corresponding to some or all neighbors for every destination. For this purpose a separate IOMT is used to store the heights of every neighbor for each available destination. These research results have been submitted to a journal [12].

1.4 Organization of this Report

The rest of this dissertation report is organized as follows. In Chapter 2 we provide a survey of relevant work in the literature.

Chapter 3 outlines the shortcomings of existing secure routing protocols, and the drawbacks of currently proposed trustworthy computing schemes to secure MANETs. Chapter 3 also provides an introduction to the proposed trusted MANET modules (TMMs) by

presenting their broad specifications, and some generic functionality that could be used by all MANET routing protocols.

Chapter 4 presents a simple TCB to secure AODV and DSR. While trivial, this TCB can nevertheless realize many of the assurances that are not possible using strategies that rely on carrying over cryptographic authentication. While this is a substantial improvement over current solutions Chapter 4 also presents various limitations of this approach - all of which result from the fact that the the TMM cannot "remember" routing data.

In Chapter 5 we present the most important of the contributions of this dissertation - an efficient index ordered merkle tree (IOMT), which can enable a resource limited module to virtually store large amounts of data by storing a single value inside the module.

Chapter 6 presents the design of the TMMs that employ IOMTs to store routing data. It also includes various functions that can be used to securely realize AODV and DSR. This chapter also includes an analysis of the security offered by the TMMs that employ IOMTs.

Chapter 7 begins with a description of the shortcomings of TORA, and proposes a new routing protocol CR-TORA which addresses the main limitations of TORA. We also present simulation results that compare the performance of both TORA and CR-TORA. Chapter 8 presents the TMM design to secure CR-TORA.

Conclusions are offered in Chapter 9. Chapter 9 also points out some possible future extensions to this dissertation research, and the publications that resulted from this work.

CHAPTER 2

LITERATURE SURVEY

Resource limited nodes forming wireless mobile ad hoc networks (MANET) rely on each other for routing packets amongst themselves, and thereby eliminate the need for an infrastructure for this purpose. Due to their reduced dependence on infrastructure, they have useful applications in scenarios where it is impractical to deploy such infrastructure.

Small scale applications include dynamic MANETs created for specific purposes like search and rescue operations. Larger scale medium-area and wide-area networks can consist of many ad hoc subnets interconnected through mesh networks or the wired Internet. Existing practical rural applications of MANETs include the popular one-laptop-per-child (OLPC) project [13]; which utilizes multi-hop networks to provide interconnectivity between laptops in under-developed countries. In urban areas, due to the high cost of acquiring real estate for erecting communication infrastructure, it will be more economical for a network operator to enlist the support of distributed wireless hot-spot operators, and home users with Internet access, to provide wide-area voice and data services to mobile users.

2.1 Ad Hoc Routing Protocols

MANET routing protocols are rules imposed on nodes to route packets amongst themselves. The primary goal of all efficient routing protocols is to maximize information

transmission while minimizing the usage of resources. Apart from many issues like correctness, stability, and fairness to be addressed by all routing protocols, MANET protocols have to address some additional constraints like 1) the resource constrained nature of mobile devices, 2) rapid changes in topology due to mobility, and 3) issues specific to wireless links.

MANET routing protocols can be classified into proactive and reactive protocols. Proactive approaches like DSDV [4] and link state routing [5], [14] strive to maintain a consistent view of the entire network at all times. In *reactive* protocols like dynamic source routing (DSR) [1], and ad hoc on-demand distance vector (AODV) [2] routes are determined on-demand. Typically the discovery of routes starts with a Route Request (RREQ) query which is flooded in a controlled fashion, and trigger a Route Response (RREP) from nodes which have knowledge of a route to the destination. Combinations of proactive and reactive approaches [6],[15] have also been proposed. Furthermore, some protocols like Temporally Ordered Routing Algorithm (TORA) [6] can operate in both proactive and reactive modes.

In general, proactive routing protocols, where each node pro actively maintains forwarding paths to reach every node in the subnet, can demand substantial overhead if the subnet size is large. On-demand protocols tend to be more efficient for large subnets. Many practical applications of MANET will involve extending the reach of base stations by providing multi-hop connectivity to the base-stations. For scenarios where a large number of nodes have to maintain connectivity to a small number of destinations (base-stations), TORA [6] may be preferable.

2.2 Secure Ad Hoc Routing Protocols

Apart from numerous considerations listed earlier which render design of MANET routing protocols more challenging, another important consideration stems from the lack of trust in routers. Unlike networks that have established infrastructure, where routers are operated by large organizations, and thus implicitly trusted, some MANET nodes may be under the control of malicious attackers. Secure MANET routing protocols also have to cater for the possibility of malicious nodes in the subnet which may deliberately attempt to sabotage the efficacy of the subnet routing mechanism.

2.2.1 Attacks

Attacks on ad hoc routing protocols that can be inflicted by non co-operating nodes can be classified into passive, active and semi-active attacks.

In passive attacks, the attacker does not transmit any data. The aim of such attacks could be to merely observe traffic patterns, or snoop on data exchanged between nodes. Passive attacks can also result from selective participation of nodes. Some nodes may participate in the routing process only if they have selfish reasons to do so, for example, if they desire to be an end-point in an exchange.

Active attacks are performed by taking an active part in relaying messages and routing information. Such attacks would attempt to modify relayed data with the intention of disrupting routing protocols, or more generally, to prevent nodes from communicating with each other. Some specific instances of active attacks include modification of hop

counts in routing tables, reporting of fictitious neighbors, or randomly corrupting routing information / data packets that are relayed.

Nodes performing semi-active attacks transmit and receive data, and do not change data that is forwarded. One example of such an attack is for a node to just re-broadcast packets that it receives without any modification, and thereby acting as an "invisible" relay. Such relays can cause many nodes which are not actually within the range of each other to conclude that they are. After taking part in such an activity for some duration, the semi-active attacker may then suddenly decide to remain silent. This can result in serious disruptions of the routing tables in the vicinity, and consequently result in substantial bandwidth overhead. Two physically well separated invisible relays can also form worm-holes [16]. By periodically turning on and off the worm-hole they can cause severe disruptions in the perceived subnet topology.

2.2.2 Countermeasures

Secure routing protocols can be broadly classified into three categories. In the first category are protocols which rely on cryptographic authentication techniques for verifying the integrity of routing information. In the DSR-based secure routing protocol (SRP) [17] only the source and destination share a secret; which is used to authenticate RREQ and RREP packets. Any modification in these packets by a malicious intermediate node can be identified at the end point, and hence will be discarded.

For a broadcast communication, appended authentication data needs to be validated by multiple verifiers. Digital signatures can be used for this purpose, but their high resource

requirements make their usage impractical. A broadcast authentication scheme TESLA [32] can be used to address this issue. It makes use of symmetric cryptography, and yet retains the required asymmetric property by clock synchronization, and delayed key disclosure. In TESLA each node picks an initial key K_N, and computes a one-way chain of secrets using $K_{N-1} = H(K_N)$. Initially, a node releases K_0, and later periodically release the remaining keys. In order to authenticate a message, a node selects a secret (say K_i), which is scheduled to be released only after the message is delivered to all the intended receivers. On receiving the message a node verifies the *freshness* of K_i, and buffers it (the received message) if key K_i is not yet disclosed. After receiving K_i, a node first verifies K_i using K_j (a key which is previously released by the sender, and $j < i$) by checking $K_j = H^{i-j}(K_i)$, and later authenticates the previously buffered message with K_i.

Ariadne [18], a popular secure extension of DSR [1], employs TESLA for authentication of intermediate nodes in the path, and a per-hop hashing technique to prevent deletion of nodes from the path. Ariadne assumes a pair of shared secrets between source and destination nodes, which are used for mutual authentication. Along with the RREQ packet, source node appends a MAC (using the shared secret), which can be verified by the destination node. Every intermediate node appends a MAC while forwarding a RREQ packet, and the appropriate TESLA key is released in the corresponding RREP packet. On receiving a RREQ, destination node verifies the source MAC, and also the hash-chain to detect any deletion of intermediate nodes. On successful verification, a destination responds with a RREP packet along with an appended MAC, which can be checked by the source node.

After receiving a RREP packet, the source node authenticates the destination, and also validates every intermediate node (using their appended TESLA keys).

SRP and Ariadne do not cater for the optimizations provided in DSR (for example, using cached routes, route response by intermediate nodes, etc). In [39] Sivakumar et al. provide a more comprehensive extension of Ariadne which supports some of the optimizations offered by DSR, due to the use of pairwise secrets instead of TESLA. Rather than encrypting packets with a network wide group secret (to prevent covert attacks) one-hop secrets (a key shared by each node with it's neighbors) are used. In [53] digital signatures are used to provide non repudiable proof of active attacks. Appended signatures are verified only by immediate neighbors. Each node strips the received signature and appends its own, before forwarding the received RREQ packet.

Most of the proposed secure extensions of ad hoc routing protocols assume bi-directional links. Unfortunately this is not a valid assumption. Sivakumar et al. [33] analyzes the effect of one-way links on DSR protocol. They explain how efficiency of the network is effected by the presence of one-way links, and also propose secure strategies to deal with one-way links.

In the secure AODV (SAODV) protocol [20] intermediate nodes are not required to append any form of authentication. A per-hop hashing strategy is used to prevent attacks involving shortening of paths (reducing the hop count value), but does not prevent the attacker from increasing the path length. Digital signatures are used to authenticate non-mutable fields in the RREQ and RREP packets. SAODV proposes the concept of dual-signatures, which can allow an intermediate node (that has a valid route to the speci-

fied destination) to respond to a received RREQ packet. In Du et al [21] secure extension of AODV, one-hop and two-hop group secrets are employed for authentication of intermediate nodes; where the group secrets are conveyed individually using certificates based on asymmetric schemes. Two hash values (H_1 and H_2) are appended to each routing packet, where H_1 (verified by the node's one-hop neighbors) is used to authenticate the received packet, and H_2 (verified by the node's two-hop neighbors) prevents the current node from tampering routing data.

Wan et al [22] and Hu et al [23] have proposed secure extensions of DSDV where each node shares a secret with every other node in the network, which is used for node authentication. The former (S-DSDV) [22] classifies advertised routes into: authoritative (routes that have either 0 or ∞ hop count) and non-authoritative Routes (routes with finite hop count other than 0). S-DSDV performs node authentication for validating all the authoritative routes. For non-authoritative routes, along with node authentication, a receiver cross-checks obtained information with the node mentioned in the next-hop field. The latter approach (SEAD) [23] employs hash chains, where the depth of the pre-image is controlled both by the distance of a node from the source and the current sequence number of the source. These hash chains are applied to authenticate the minimal hop counts included in a routing update. Like SAODV, an attacker can not decrease the hop count value, but can increase it.

While a majority of secure routing protocols focus on the need for cryptographic *verification* of the integrity of the established paths, the problem of *identifying* the perpetrators responsible and strategies for "routing around" malicious nodes have received little atten-

tion. Awerbach et al [24] and Burmerster et al [31] propose tracing algorithms to detect Byzantine faults. First approach [24] is applied on AODV, where each intermediate node initializes a timer T_i during route creation (which is based on its distance from the destination). After forwarding a data packet, an intermediate node expects an acknowledgment from the destination within T_i cycles. An intermediate node, whose T_i expires before receiving the acknowledgment forwards an error report to the source, by which it (the source) can locate the faulty link. The second mechanism [31] aims at identifying the malicious node, by repeatedly probing intermediate nodes to send an acknowledgment. This issue is also investigated in [39], in the context of DSR.

In the second category are schemes which rely on assigning trust metrics to nodes based on first-hand and second-hand observations. In [40] Bucheggar et al. proposed a monitoring scheme where each node comprises of the following components: The monitor (which oversees the traffic generated by neighbors), Trust Manager (responsible of generating alarms when malicious behavior is identified), The reputation system (where the gathered information is analyzed to identify malign nodes), and the Path Manager (handling routes and traffic related to malicious nodes). Liu et al. [48] presents a *reputation* mechanism which is used to rank nodes based on their observed behavior, and received recommendations. Some researchers have proposed operation in the promiscuous mode [40] - [42] as a strategy to identify misbehaving nodes. Pirzada et al [43] propose a mechanism for securing TORA, by employing trust metrics obtained through analysis of local traffic.

In the third category are schemes which rely on trusted computing modules to enforce compliance to routing protocols [49] - [52]. Song et al [50] include the wireless transceiver inside the trust boundary, by implementing it using tamper resistant hardware. In [49] by Jarrett et al, the trusted computing module has complex features built into the wireless driver (executed within the confines of the trusted module) to verify the integrity of wireless transceiver. In [52] Buttyan et al "nuglets of currency" are protected by smart-cards to promote faithful forwarding of packets. The authors provide explicit consideration to the need for lowering the complexity of tasks to be performed inside the trusted boundary.

2.3 AODV and DSR

AODV is an ad hoc on-demand routing protocol where the distance to a destination is stored in a node's routing table as a destination record (DR), indexed using the destination identity. Other fields in the DR include a sequence number, hop-count, next-hop to reach the destination, and validity time.

When a node desires to communicate with a destination, and finds that it does not have a fresh route to the destination, a RREQ is flooded, indicating a fresh sequence number of the initiator, the last known sequence number of the destination, and a hop-count field which is initially set to zero. Every node that receives the RREQ

1. updates the sequence number of the source, and adds a DR (for the source) to a table of DRs.
2. if the node does not have a path to the requested destination it forwards the RREQ after incrementing the hop count field by one;
3. if the node has a fresh enough path to the destination (or if the node is the destination itself), it responds by unicasting a RREP towards the source by sending the RREP to the neighbor from which it received the RREQ.

In the case of RREP by an intermediate node the hop count in RREP is set to the stored value, and in the case of RREP by the destination, it is initialized to zero. Every node receiving the RREP adds a DR for the destination, increments the hop count, and unicasts the packet towards the source node.

Every entry in the routing table has a validity time after which it cannot be used. However, due to the mobility the information in the DRs can become invalid even before the expiry period. AODV handles such premature expiry using route error (RERR) messages.

DSR also employs a similar RREQ, RREP and RERR packets. The difference is that while in AODV intermediate nodes (that forward the RREQ) increment the hop-count, in DSR every intermediate node inserts its identity. Thus, compared to AODV, DSR provides some additional topology information. The primary disadvantage of DSR is the additional bandwidth overhead for RREQ and RREP packets (which indicate the entire path instead of a single field - hop count). The advantage accrued from the knowledge of the entire path is that multiple paths between the source and destination can be established.

2.3.1 Secure Extensions of AODV

In the Secure AODV (SAODV) protocol [29] every node has a public-private key pair with a certified public key. Digital signatures are used to authenticate immutable fields in RREQ, RREP and RERR messages, which are specified by the source of the packet (an end-point). The immutable field includes a commitment to a hash chain of length $x_0 \cdots x_n$, (where $x_i = h(x_{i-1})$, and $h()$ is a cryptographic hash function like SHA-1) where n is the maximum length of the path. The RREQ from the source (indicating hop-count 0) is

accompanied by a value x_0. Nodes at the first hop are required to increment the hop-count by 1, and propagate the RREQ along with the value $x_1 = h(x_0)$, and so on.

In order to secure route responses by intermediate nodes SAODV employs double signature extensions for RREQ and RREP packets. The RREPs generated by an intermediate node includes the signature of the destination to validate the immutable fields, and also a signature of the intermediate node to authenticate the new validity time.

In the SAODV-2 protocol [38] only symmetric cryptography is used. SAODV-2 argues that schemes for establishment of pairwise secrets between two nodes demand substantially lower computational and bandwidth overhead. Further, SAODV-2 uses two hop authentication to thwart illegal hop changes. In two-hop authentication a node (say A) appends a MAC which is verifiable by its two-hop neighbors (say C). When C receives this packet via B (neighbor of A) it can compare the hop count announced by B to the one included by A, and thereby identify illegal modifications.

Asad Amir Pirzada and Chris McDonald proposed a secure variant of AODV [61] where authorized neighboring nodes share a common secret, which is used to encrypt all the forwarded control packets. They use asymmetric cryptography to arrive at shared secrets. Their scheme does not address the issue of an authorized node behaving maliciously. Leiyuan Li and Chunxiao Chigan proposed Token Routing Protocol(TPR) [62] with a motivation of reducing the computation costs involved in SAODV (as the later uses asymmetric cryptography). Unlike SAODV they suggest the usage of pairwise secrets between ad hoc nodes. TPR uses two hash chains, one to protect hop count (which is identical to the one used in SAODV) and another to provide node authentication. Node

authentication is provided using *tokens*, which in turn are hash chains computed using the shared secret between source and destination.

2.3.1.1 Secure Extensions of DSR

A well known secure extension of DSR is Ariadne [18] which uses the TESLA broadcast authentication protocol [32] for authenticating network packets. Every intermediate node forwarding an RREQ appends a TESLA MAC which can be verified at the end of the reverse path. As the authentication appended by every node will be verified by the RREQ source, nodes cannot be inserted into the path. To prevent nodes from deleting other nodes in the path a per-hop hashing strategy is used which leverages a shared secret between end-points (the RREQ source and the destination).

In [39] an improved Ariadne (iAriadne) was proposed which mandates every node to maintain a "private logical neighborhood" (PLN) and introduces an additional value to be inserted by every node forwarding the RREQ - an "encrypted up-stream per-hop hash." Unlike Ariadne with TESLA, iAriadne relies on pairwise secrets between nodes.

2.4 TORA

TORA is a hybrid ad hoc routing protocol which can operate in both reactive and proactive modes to determine multiple paths to some destinations. Unlike AODV and DSR, where the route discovery process is to enable a specific node to find a path / forwarding information to a specific destination, in TORA, all nodes in the subnet learn forwarding information to the destination. Additionally, unlike proactive protocols like DSDV where

forwarding information is sought for all nodes in the subnet, in TORA forwarding information is sought only for a few specific destinations. TORA can also operate in proactive mode, where destinations periodically advertise their presence.

In TORA [6] the route to a destination from any node is based on the "height" of a node with respect to the destination. Every node maintains a set of parameters which specifies its height (for a specific destination). The set of parameters which reflect the height of a node is a 5 tuple $(\tau, oid, r, \delta, id)$. The first three values (τ, oid, r) form the "reference level" where:

1. τ is the logical time of creation of the reference level (which is usually a result of the failure of a link);
2. oid is the identity of the node that created the reference level; and
3. r is a reflection bit (more on this later);

The value δ is incremented at every hop from the node which created the reference level. The field id is the identity of the node. Together, the five fields reflect the "height" of a node from the destination.

In a subnet with d destinations the TORA protocol can be seen as d independent instances of the protocol - one for each destination. For each destination every node stores heights of each of its neighbors. Based on the heights of its neighbors, a node determines its own height. If a node A has greater height than a neighbor B, then A is regarded as the upstream neighbor of B (and B is the downstream neighbor of A). Data can flow only downwards; i.e from upstream to downstream.

If the height of a node A for a destination is NULL (or height $(-, -, -, -, A)$), this implies that no forwarding path is available for A. If the height is any value other than

NULL, the node can forward a packet to the destination by sending it to any downstream neighbor. The height of the destination (for itself) is ZERO (or $(0, 0, 0, 0, id)$, where id is the destination identity).

From a broad perspective the goal of TORA is to ensure that as long as a physical path exists from a node to a destination, the height of the node (for that destination) should be non NULL, and moving downstream at every hop should ultimately lead to the destination (though not necessarily over a minimal number of hops).

2.4.1 Route Discovery, Maintenance and Erasure

TORA consists of three phases in its routing:

1) Route Discovery (Query, Update and OPT): A node which has a NULL height to a destination can broadcast a query (QRY) packet for the destination. Each node receiving the QRY packet checks whether it has a non NULL height for the destination. If the node also has a NULL height it rebroadcasts (forwards) the QRY packet. In a scenario where the QRY packet reaches the destination, it broadcasts its ZERO height in an update packet (UPD). On receiving an UPD packet from its neighbor, every node updates the neighbor's height. A node sets its own height to the received value, and increments the δ value by 1, and broadcasts its own height. For instance, node A receiving $(0, 0, 0, 2, B)$ from a neighbor B, stores the received tuple as B's height; sets its own height to $(0, 0, 0, 3, A)$ and forwards it in an UPD packet. Now A is the upstream neighbor of B (for the specific destination).

In proactive mode where destinations pro actively advertise their presence, every destination periodically broadcasts its existence in an special update packet termed as OPT (which includes the height of the destination D- $(0, 0, 0, 0, D)$). Each OPT packet from the destination indicates a sequence number (referred to as mode sequence number). Thus, in TORA the heights of nodes are established during QRY-UPD process, and in proactive versions of TORA, periodically refreshed through OPT packets originating from destinations. Changes in topology that occur between two OPT packets (with different mode sequence numbers) are handled by the route maintenance process.

2) Route Maintenance : Route maintenance is triggered only when a node no longer has a path to the destination (has no downstream neighbors). To understand the rules governing TORA's route maintenance, consider a node i, and let $\mathbb{N}_i$ be the set of neighbors of the node i.

Generating a new reference level: Let us assume that the current height of node i is $(\tau_0, oid, r, \delta_i, i)$. If node i looses a downstream link with a node $j \in \mathbb{N}_i$, and if j is *not* the last downstream node, i does nothing - except remove the height entry for the node j. On the other hand, if i looses its last downstream link, i creates a new reference level $(\tau, i, 0)$ where $\tau > \tau_0$ (where τ_0 is the first field in the current reference level). Node i sets its height to $(\tau, i, 0, 0, i)$. The second field i indicates that node i generated this new reference level. By generating a new reference level, node i *reverses its links* to its upstream neighbors. In other words, i requests the nodes which were previously upstream of i to become i's downstream nodes.

Propagating a new reference level: If node i loses its last downstream link due to a link reversal (by its previously downstream node), node i determines the neighbor with the *highest* reference level. A reference level $(\tau_1, oid_1, r_1) > (\tau_2, oid_2, r_2)$ if $\tau_1 > \tau_2$. If $\tau_1 = \tau_2$, then $oid_1 > oid_2$. If both these fields are equal then the level with a larger r is higher. If multiple neighbors have the same highest reference level, say (τ_j, oid_j, r_j), i chooses the one with the least δ value. Let the height of that neighbor $j \in \mathbb{N}_i$ be $(\tau_j, oid_j, r_j, \delta_j, j)$. Now i sets its height to $(\tau_j, oid_j, r_j, \delta_j - 1, i)$, and sends its height in a UPD packet. By setting its δ lower than j, i becomes a downstream node of j.

Reflecting a reference level: If node i loses its last downstream link due to a link reversal, and all it's neighbors have the *same* reference level (τ_j, oid_j, r_j), and if $r_j = 0$, node i reflects the reference level by setting $r_j = 1$ in its height. In this case the height of i becomes $(\tau_j, oid_j, 1, 0, i)$. Note that $(\tau_j, oid_j, 1) > (\tau_j, oid_j, 0)$. This height is announced to its neighbors in an UPD packet.

Detecting a partition: Now let us consider the scenario where node i looses its last downstream link due to link reversal, and all its neighbors have the same reference level $(\tau_j, oid_j, 1)$ with the reflector bit set.

If i did not originate the reference level (or $i \neq oid_j$) then i generates a new reference level and sets its height to $(\tau', i, 0, 0, i)$ where $\tau' > \tau_j$. On the other hand, if the reference level oid_j was created by i (or $i = oid_j$), then i detects a partition in the network rendering the destination unreachable. In this case i sets its height to NULL - or $(-, -, -, -, i)$. In the former case the height of i is announced in an UPD packet. In the later case the height is announced in a clear (CLR) packet.

The basic algorithm of TORA's route maintenance phase is detailed in Figure 2.1.

```
IF (link-failure)
   GENERATE-RL
ELSE //(link reversal)
   IF (all-neighbors-not-at-same-RL)
     PROPAGATE-RL
   ELSE
     IF (r == 0)
       REFLECT-RL
     ELSE
       IF (RL-created-by-me)
         CLEAR-RL
       ELSE
         GENERATE-RL
```

Figure 2.1

Route Maintenance of TORA

Route erasure: The CLR packet contains the destination and the reference level of the node in addition to the five-tuple height. Every node that receives the CLR packet and has a matching reference level erases its height (sets it to NULL) for the destination indicated, and re-broadcasts the CLR packet.

2.4.2 IMEP

Internet MANET Encapsulation Protocol (IMEP) [70] was primarily meant for aggregation of upper layer protocol (routing protocols) packets to encapsulate smaller control packets into fewer IMEP packets, and thereby reduce channel access delays. IMEP also provides other useful services like broadcast reliability, link status sensing, and authentication. Connection status of neighboring nodes is determined by exchanging BEACON and

ECHO packets. When reliable delivery is requested, IMEP mandates acknowledgments to verify accurate reception of packets. IMEP uses a *selective repeat* algorithm to deliver lost packets. It employs digital signatures to provide authentication, and also describes a message format for exchanging certificates. TORA was intended to be layered over IMEP.

2.4.3 Secure Extensions of TORA

While multiple secure protocols have been proposed for cryptographic authentication of routing fields for other ad hoc routing protocols like AODV, DSR and DSDV, none have been proposed for TORA.

Vee Liem Chee et al. [60] analyzed security flaws in TORA, and identified three attacks - route disruption, route invasion and resource consumption, that a malicious participant can launch. These attacks are carried out by faulty processing of control messages in the network. Asad Amir et. al [43] propose a mechanism to secure TORA based on trust development. They ignore the need for cryptographic authentication to monitor neighbors. In their scheme, a node passively monitors the packets and data it receives from its neighbors. It then assigns trust values (-1 to 1) to the collected information, which is later classified into various groups. Finally weights are assigned to each identified group, and a weighted total of trust for a particular node is computed. They propose an extension to their trust model in [35] where nodes can exchange the calculated trust metrics.

The key security requirement in any network is authentication; a receiver should be able to verify the identity of the sender. Without authentication any node can send incorrect

packets with spoofed identity, which will result in assigning low trust values to the node whose identity is spoofed.

While IMEP, which is the underlying layer for TORA, can provide cryptographic authentication of transmissions by neighbors, this feature is optional. Furthermore, that a node A receives a cryptographically authenticated height value from a neighbor B only implies that the source of the broadcast is B - not that the height provided by B is correct. Verifying integrity of routing messages also mandates some redundancies. Such requirements are also ignored in [35].

2.5 Trustworthy Computing

2.5.1 Trusted Computing Base

The trusted computing base (TCB) of a system is "a small amount of software and hardware we rely on, and that we distinguish from a much larger amount that can misbehave without affecting security" [7].

As an example, consider a generic communication system where an important assurance sought is the ability to verify that a message sent from one entity to an another cannot be modified in transit by intermediaries. To realize such an assurance we typically rely on a TCB which includes a certificate authority (CA), who (we assume) does due diligence before signing public key certificates, and ensures that it's private key is well protected. We also rely on the assumption that cryptographic algorithms like RSA, DSA, AES, SHA-1 etc., are unbreakable. When one receives a message authenticated using the secrets be-

longing to an entity A, it is assumed that the message is from A, as it is implicitly assumed that the secrets of A are privy only to A.

As a more concrete example, the widely used web-security protocol, SSL, leverages such a TCB to provide an assurance that data sent by clients will be privy only to SSL servers. However, when a client sends some sensitive information (like a credit-card number) to a server over an SSL connection, it only ensures that the information remains private till it reaches the server. There is no assurance that such information cannot be abused *after* it reaches the server, say by entities who have unfettered access to the server. Thus, in many practical scenarios, the limited TCB which caters only for cryptographic authentication, is not sufficient as a basis for realizing important assurances.

2.5.2 Trustworthy Computing Modules

The most common approach to expand the Trusted Computing Base (TCB) is by employing trustworthy computing modules which provide some "specialized" TCB functions, performed inside trustworthy boundaries.

In the trustworthy computing group (TCG) model [25] for realizing trusted platforms a trustworthy platform module (TPM) performs several specialized *fixed* functions to provide i) the ability for remote parties to verify that the platform equipped with the TPM is in an "acceptable state" - that only authorized software has been loaded and executed by the CPU (even though the TPM does not have direct control over the CPU); and ii) the ability to provide secrets to the TPM, bound to some platform states, which will be released by the TPM only when the platform is in that specific state.

Unlike TPMs, the IBM 4758 [26] trustworthy computing module (TCM) sports a general purpose processor inside a protected boundary, running a specialized operating system, and can execute application code unmolested *inside* the trusted boundary. The rich set of programmable functions that can be executed inside the boundary can provide a rich TCB that can be leveraged to realize assurances that may not be possible otherwise.

Unlike inexpensive TPM chips (a few dollars) with fixed functionality, the programmable TCB offered by IBM module comes at a substantially higher cost (a few thousand dollars). It is for this reason, that in this paper we seek a set of fixed functionality suitable for securing MANETs. We deliberately impose some restrictions on such fixed functionality to ensure that TMMs which offer such functionality can be easily verified, will consume negligible power, and thus can be simultaneously trustworthy and inexpensive to realize.

TMMs that offer the TCB for securing MANETs will demand substantially lower complexity compared to even inexpensive TPM chips. Unlike TPMs which offer a set of about 120 fixed functions, TMMs will offer substantially fewer number of such functions. Furthermore, unlike TPM chips, TMMs will not require to perform asymmetric cryptographic computations. TMMs will merely perform fixed sequences of logical and cryptographic hash operations.

2.5.3 Existing Trustworthy Computing Schemes for MANETs

Many schemes have been proposed in the literature that propose the usage of trustworthy computing to protect ad hoc networks. Trusted modules can be used to verify the state of the currently loaded routing protocol, and can also attest this information to a remote

verifier. In [55] Mingsheng et al have proposed to use this remote attestation feature of a trusted module to secure ad hoc networks. The initial state of the routing protocol is calculated, and is stored securely in a platform configuration register (PCR). When ever the operating system loads the routing protocol, the TPM compares its state with the value stored in its PCR. A TPM can also attest this information to a remote verifier using public key cryptography.

Remote attestation can expose the identity of a user, and hence has some privacy concerns. The approach proposed in [56] employs direct anonymous attestation (DAA), a cryptographic protocol that provides remote attestation while preserving user identity.

The scheme presented in [57] proposes a Tamper Resistant Module (TRM), which can be used to protect both the routing module and the MAC layer of the ad hoc node. They also assume a secure channel between the routing module and MAC layer, that can be used to exchange data securely.

Michael et al in [58] employs trustworthy computing to address issues related to both selfish nodes, and secure routing. They assume that the routing agent, which includes all the software and operating system components that are responsible for managing routing, and wireless driver are executed within trusted boundaries. The routing agent is responsible for implementing all the routing rules, and securely communicates with the wireless driver to obtain reports about the packets forwarded by the node. The routing agent refuses to take part in future routing when the metrics obtained in the reports drop below a predefined threshold value.

2.5.3.1 Dual Agent Approach

In [54] Gaines et al argued the need for a *dual agent approach* to secure ad hoc routing protocols. In this approach any MANET device is seen as consisting of two independent agents: 1) an *untrusted, selfish user agent* satisfying the needs of the owner of the device, and 2) a *trusted, selfless network agent* working towards the overall good of the network. The network agent (for example, a tamper-sensitive chip in the mobile device) is entrusted with the responsibility of ensuring that the mobile device does not violate the protocol.

In the dual agent model with a trusted network agent (NA) and an un-trusted user agent (UA), the Network Agent is a tamper-proof secure co-processor, which is physically in the possession of the User Agent. Every packet sent or action taken by the User Agent is verified and signed by the Network Agent. Packets not authenticated by a network agent are dropped at the receiving end. Unlike other secure protocols which attempt to overcome the effects of malicious nodes, the intention in dual agent approach is to prevent nodes from misbehaving. However, elements from other secure routing protocols can still be used to cater for the failure of network agents.

While many researchers have proposed similar models, they did not take into consideration some of the constraints that need to be imposed on the network agents. Jarrett and Ward proposed Trusted Computing Ad hoc On-demand Distance Vector protocol (TCAODV) [49], which is a secure variant of AODV that employs a trusted module. Asymmetric cryptography is used to provide authentication between nodes (trusted modules), and each control packet is signed by the sender's trusted module (routing agent). In

their approach a node's routing agent is aware of the protocol functionalities, and ensures a node's adherence to the employed routing protocol.

The dual agent approach by Gaines et al was motivated by the following reasons:

1. It is not possible for the network interface to be under the control of the trusted agent. The user agent can simply drop signed packets.
2. It is impractical for the network agent to perform all tasks - network agents have to rely on the user agent to observe neighbors.
3. It is essential to reduce the complexity of the network agent to the extent feasible in order to improve its reliability.

Motivated by the need to reduce complexity of network agents Gaines et al divided the tasks to be performed by a MANET node into two categories: selfish tasks and selfless tasks [54]. Sending its own data, generating routing traffic in search of a route, can be considered as selfish tasks. Routing network traffic, relaying information generated by other nodes, constitute selfless tasks. Security mechanisms employed should acknowledge this difference, and provide means to secure each type of task separately. Such a distinction is lacking in existing approaches to secure routing protocols. The role of the network agent should be limited to securing self-less tasks.

Gaines et al [54] apply dual agent approach to secure DSDV [4]. More specifically, the intent of [54] was investigation of efficient strategies to protect the integrity of the distance vector routing tables stored by the user agent. A combination of NSEC [64] (a mechanism used for authenticated denial of existence of DNS resource records) and Bloom filters [65] are used for this purpose. More specifically, NSEC employs a hash which links one element to the next element in an ordered list. Thus the ability to prove that (for example) 10 follows 5, is a proof that 6, 7, 8, and 9 do not exist. A HMAC (computed using a key

stored with the network agent) is appended to each NSEC record by the network agent. This would make it tamper-proof from the user agent.

Routing information rendered stale prematurely, due to the availability of new information, are stored in a Bloom filter (which are stored in network agent) to ensure that the user agent cannot replay such stale data. Their simulations show that for a network of 400 nodes, about 2.5kb of storage is required by each network agent (to hold information that is required to counter the attacks mentioned); and the efficiency of the network is about 90% (at any time instant in the network, on an average 90% of the routes that physically exist are reflected consistently in the routing tables of all nodes).

CHAPTER 3

SECURING MANET ROUTING USING TRUSTWORTHY MANET MODULES

As seen in the previous chapter, several secure extensions of various MANET routing protocols have been proposed in the literature. From a broad perspective the main shortcomings of current secure routing protocols are

1. High overhead for the additional security measures; and
2. inability to provide several important assurances.

3.1 Shortcomings of Current Secure Protocols

Even while demanding substantial overhead, SAODV still leaves an AODV MANET susceptible to a wide range of attacks. Some of the main shortcomings of SAODV (which are addressed in SAODV-2) are

1. lack of mechanisms to authenticate intermediate nodes (intermediate nodes are not required to append any authentication)
2. inability to prevent some misrepresentations of the hop-count: it can only ensure that a node receiving an RREQ/RREP with hop count r cannot relay a hop count less than r. A malicious node can incorrectly (or maliciously) relay the same hop count r or a hop count greater than r, and
3. lack of mechanisms to address one-way links.

While SAODV-2 addresses many of the pitfalls of SAODV, SAODV-2 is still susceptible to many attacks. SAODV-2 assumes that nodes will not collude together. Colluding nodes

can easily thwart two hop authentication. Secondly, there is no mechanism for regulating creation of RERR packets.

Unlike SAODV (where intermediate nodes are not required to append authentication), in Ariadne authentication of intermediate nodes is mandatory. Unfortunately, in most secure extensions of DSR the verification of the authentication occurs very late in the RREQ-RREP process. For this reason, malicious nodes in the path can simply send random RREQ packets which even though will not be accepted by end-points, can preempt propagation of genuine RREQs.

The main pitfalls of Ariadne that are addressed by iAriadne are addition of mechanisms for i) link-layer (one-hop) authentication, and ii) preventing abuse of one-way links - both of which are achieved by imposing a private logical neighborhood (PLN). iAriadne also introduces an additional upstream per-hop hash to be introduced by every node relaying the RREQ to facilitate the RREQ destination to narrow down intermediate nodes that engage in active attacks.

3.1.1 Collusion

However, the security of all four protocols (SAODV, SAODV-2, Ariadne and iAriadne) are based on the assumption that nodes will not collude. In both SAODV and SAODV-2 colluding nodes can ensue that even shorter hop counts can be relayed. In Ariadne and iAriadne colluding nodes can trivially delete nodes from the path.

The primary reason that facilitates easy collusion is that two nodes A and B simply need to share their secrets to do so. Through their ability to send packets impersonating A

or B, both A and B can create packets with misleading information - with the misleading information provided by A consistent with the misleading information provided by B. Obviously, three such nodes A, B, C sharing their secrets can engineer an even broader variety of attacks.

3.1.2 Carrying Over Authentication

Security mechanisms that employ carry over authentication would increase the packet size, as each packet needs to hold authentication information about multiple nodes. Now Consider an example where a packet was relayed by node A, and later was forwarded by B to node C. In schemes based on two hop authentication, apart from validating the received information, node C should also verify that A is a neighbor of B, and that B indeed is relaying the information it directly received from A. This requires C to accurately obtain the neighborhood information of node B, which would incur a lot of overhead.

3.1.3 Unregulated RERR Creation

Another pitfall of all secure on-demand protocols is the lack of mechanisms to regulate the creation of route-error (RERR) packets. An effective strategy for an attacker to introduce unnecessary additional overhead is to participate faithfully in establishing a path and then send a supercilious RERR packet - mandating a fresh route creation process.

For example, an attacker C in a path $A \rightarrow B \rightarrow C \rightarrow X$ can simply send a RERR claiming to have lost the link to X (even while the link exists). The risk the attacker C faces in doing so is that if X hears the RERR packet, X may disregard future packets from

C (which might affect C's ability to connect to some nodes). A selfish node desiring to retain X as a neighbor can however still manage to send such a RERR packet by ensuring that the RERR will not be heard by X. This can be achieved easily by exploiting the medium access control protocol. For example, if a collision avoidance protocol is used, C can send the packet as soon as X sends a clear to send (CTS) or a request to send (RTS) packet, thereby ensuring that C's RERR suffers collision at X (but will be received collision-free by other neighbors of C).

3.1.4 Limited Optimizations

Original versions of AODV and DSR include several optimizations to improve their efficacy. Such optimizations include route responses by intermediate nodes, local path repairs, using cached information for relaying messages etc. Most secure extensions of protocols do not support such optimizations as they introduce new avenues of attacks. Thus, the overhead for using secure routing protocols have two sources - direct overhead for cryptographic authentication, and indirect overhead resulting from the fact that several useful optimizations cannot be used.

3.2 Shortcomings of Current Schemes Based on Trustworthy Computing

The above mentioned limitations of currently proposed secure extensions can be addressed by using a trusted module (like TPMs) that governs the routing actions taken by a node. A trusted module is assumed to be tamper resistant, and can securely execute the operations encoded into it.

The idea of using trusted modules to secure ad hoc routing protocols is widely being investigated. However, most of the current work make unrealistic assumptions about the trusted module, and also does not provide a clear explanation on how they can used to realize existing MANET routing protocol.

3.2.1 Questionable Trusted Boundaries

The components included within the boundaries of a trusted module are assumed to be tamper proof. Hence it is required to deliberately limit elements enclosed within trusted boundaries. However, the proposed schemes violate this requirement by assuming the trusted modules to include several system level components. The mechanisms proposed in [55, 56] include BIOS, boot loader procedures, and operating system kernel inside trusted boundaries. In [57, 58] the MAC layer of a MANET node is also enclosed by a trusted module. Such unrealistic assumptions are hard to realize, and drastically increase the trusted boundary space.

3.2.2 Computational Overhead

Trusted modules are assumed to be physically well shielded in order to prevent unauthorized modifications to encoded logic, and also to protect internally stored secrets. Therefore a trusted module is required to be severely resource limited, to offer itself for proper shielding. Contrarily, the proposed trusted computing schemes assume the trusted module to execute complex algorithms like public key cryptography. Further they also assume to store hugely accumulated routing data (like routing tables) within trusted boundaries.

3.2.3 Lack of Proper Interface Definitions

None of the currently existing schemes present a clearly defined interfaces which can be used to realize an existing MANET routing protocol. Instead, they either conveniently assume that the entire routing logic is implemented within trusted boundaries, or that a trusted module verifies the correctness of the routing protocol, when it is loaded by the operating system.

3.3 Trustworthy MANET Modules

In our research we designed a trustworthy MANET module (TMM) which can be used to secure MANET routing protocols. These TMMs are designed to address the shortcomings of existing secure extensions. Additionally, we clearly define the functionalities of a TMM, by providing a detailed description of the interfaces exposed by them (TMMs). The routing logic of a MANET routing protocol is distributed among these interfaces, and a node can invoke them in a particular order to realize the routing protocol supported by the TMM. Finally, we deliberately limit the computational and storage capabilities of these TMMs.

3.4 Specifications of TMMs

In the generic approach outlined in the rest of this dissertation it is assumed that a TMM is housed in every mobile node. In the interest of rendering the module trustworthy, TMMs are constrained to possess low complexity. By performing simple operations TMMs attempt to ensure that mobile nodes will not be able to violate the routing protocol.

It is assumed that every TMM has a unique identity (which is the same as the identity assigned to the node). Several light-weight key distribution schemes exist to permit any two TMMs to establish a shared secret. In the rest of this dissertation it is assumed that the modified Leighton-Micali scheme is used for this purpose, where computing any pairwise secret will require nodes to perform a single hash operation.

TMMs recognize a simple data structure for destination records (DR). TMMs receive authenticated records from TMMs of other nodes, modify some fields in the destination records subject to some simple rules (for example, incrementing a hop-count field), and authenticate records to other nodes. Message authentication codes (MAC) based on pairwise secrets are used for authentication of records.

TMMs also maintain a table of neighboring nodes. A node A is recognized as a neighbor by a node B only if an authenticated packet from A is received. Every packet is acknowledged. Only if an authenticated acknowledgment is received from A (for a packet sent by B) does B consider A as a neighbor with bidirectional link. TMMs honor destination records only from neighbors with bidirectional links, and send authenticated routing records only to neighbors with bidirectional links.

By maintaining a table of neighbors, and by performing simple and fixed sequences of logical and hash function operations, TMMs ensure adherence to the rules that govern a MANET protocol. More specifically, the assurances realized using the proposed approach is only based on the assumptions that

1. the secret stored inside the TMM cannot be exposed, and
2. the simple and fixed functionality of the TMM cannot be modified.

More specifically, the nodes themselves, and the users in control of the nodes are not trusted. In the rest of this dissertation we describe the precise functionality of TMMs required for securing various MANET routing protocols.

3.4.1 High Level Architecture of TMMs

It is assumed that every TMM possesses

1. an in-built cryptographic compression function $h()$ (for example, SHA-1);
2. protected non-volatile memory for storing one or a few symmetric secrets, and one or a few non-secret values which have to be remembered across reboots of the TMM;
3. limited volatile RAM for storing some dynamic values like a table of neighbors, some protocol parameters etc.;
4. I/O registers;
5. a clock-tick counter; and
6. control logic which drives the functionality of the TMM.

A TMM assigned identity X is assumed to possess a secret K_X, provided to the TMM by a trusted key distribution center. This secret is used for computing pairwise secrets - for example, a pairwise secret K_{XY} privy only to TMMs X and Y. The TMM X also possesses a secret S_X known only to itself (generated by TMM X).

It is assumed that the clock of the TMM runs even when the TMM is powered off. An offset o_x, provided to the TMM by a trusted authority, is subtracted by the TMM to compute the current time t. We represent the current time as seen by X as t_x. It is assumed that all TMMs agree on the current time "reasonably well" (for example, within a few tenths of a second).

TMM functionality necessary for time synchronization and boot-strapping of TMMs are *not* a concern of this dissertation. The main focus is identifying simple TMM functionality to ensure adherence of nodes to the MANET routing protocol.

3.4.2 Pairwise Secrets Between TMMs

Several key distribution schemes for facilitating pairwise secrets between trustworthy modules have been proposed in the recent past. For scenarios involving trustworthy modules there are compelling reasons to reduce the computational overhead inside the module for the operations performed using protected secrets. For the scheme in [27] each module will be required to store a few tens of keys and perform a few tens of block cipher operations for computing any pairwise secret. For the scheme in [28] each module will need to store a single secret and perform a single block-cipher operation.

While both schemes support asynchronous induction of nodes, the former [27] can support unlimited network sizes; the latter [28] imposes a soft limit on the maximum number of nodes (for example, not much more than a few tens of millions). In this paper we assume that the scheme in [28] is used for facilitating pairwise secrets. Specifically the pairwise secret K_{AB} between A and B is computed by A as

$$K_{AB} = h(K_A \parallel B) \oplus P_{AB}. \tag{3.1}$$

where P_{AB} is a pairwise public value. The TMMs do not have to worry about the integrity of the public values. The public values are maintained by nodes. Modifications to the public values cannot result in exposure of the secrets.

3.4.3 Neighbor Table

The neighbor table of a TMM X consists of one or more neighbor records of the form

$$\mathbf{N} = [id_n \parallel K_{pw} \parallel t' \parallel l] \tag{3.2}$$

where id_n is the identity of a neighbor, K_{pw} is a pairwise secret, t' is the freshest authenticated time-stamp from id_n, and $l \in \{0, 1, 2, 3\}$ is the *status* of the link to neighbor id. Status 0 is unverified link. Status 1 indicates verified link, which has not been tested for bi-directionality. Status 2 reflects a verified bidirectional link. Neighbors with status 3 are those that are revoked by the node. These neighbors can not be used for routing until the period t' expires.

The neighbor table $\mathcal{N}$ is a set of neighbor records. As an example, the contents of the neighbor table of a TMM X can be as follows:

$$\begin{array}{cccc} A & K_{XA} & t'_a & 2 \\ B & K_{XB} & t'_b & 3 \\ C & K_{XC} & t'_c & 1 \\ E & K_{XE} & 0 & 0 \end{array} \tag{3.3}$$

indicating a neighbor A with tested bidirectional link, one neighbor C with untested bidirectional link, and an unverified entry for a node E. Further X explicitly revoked neighbor B, and hence can not send packets to it until time t'_b. At a time t_x according to TMM X, an entry with time-stamp $t' < t_x - \delta_n$ (where δ_n is a constant) is also considered as an unverified entry. For such an entry in $\mathcal{N}$ the time stamp and link status are set to 0. Basically it means that neighbors unheard for δ_n time are marked as unverified.

3.4.4 Authentication Record and Message Authentication Codes

TMMs recognize the structure of authentication records of the form

$$\mathbf{A} = [id_p \parallel h_r \parallel t \parallel ACK \parallel \mu], \tag{3.4}$$

where t is a time-stamp and h_r is a value provided by a node id_p (in future references we term it as the provider). The value ACK is a flag which is set to one to indicate an acknowledgment. The value μ is a MAC computed by the provider id_p.

Typically, the value h_r is a hash of a destination record of the form

$$\mathbf{D} = [id \parallel q \parallel m \parallel a \parallel \tau] \tag{3.5}$$

id is the identity of the destination, q is a sequence number, m is a metric (typically hop-count), τ is an absolute value of time, and a is an auxiliary value which provides additional information regarding the path to the destination with identity id. However, depending on the specific nature of the protocol the interpretation of the values may differ.

3.4.5 Protocol Parameters

TMMs recognize the following protocol specific parameters:

1. δ_n: maximum age of neighbor time-stamp;
2. Δ: validity period for destination records.
3. Q_x: current sequence number of TMM X
4. MAX: Maximum size of neighbor table
5. INF: Hop count considered as unreachable
6. Δ_W: Waiting period
7. ω: specifies the protocol in use

When the TMM is turned on for the first time these parameters are initialized to their default values. To securely load this information the key distribution center can provide the required information securely using the shared secret K_X. Additionally as we shall see in later chapters, not all parameters are used for all protocols.

3.4.6 MACs

TMMs verify/compute many types of MACs:

1. μ_r: MACs for time-stamped destination records;
2. μ_t: MACs for time-stamps;
3. μ_a: Acknowledgment MACs; and
4. μ_s: self-MAC

A MAC of the form μ_r is computed to securely convey the hash of destination record from one TMM to another. To convey a DR with hash h_r to a neighbor A, TMM X computes (at time t_x)

$$\mu_r(X, A) = h(h_r \parallel t_x \parallel 0 \parallel K_{XA}) \tag{3.6}$$

A time-stamp MAC of the form μ_t is computed by X for verification by B

$$\mu_t(X, B) = h(t_x \parallel t_x \parallel 0 \parallel K_{XB}). \tag{3.7}$$

An acknowledgment MAC μ_a to acknowledge a MAC μ_r is computed by A

$$\mu_a(A, X) = h(h_r \parallel t_a \parallel 1 \parallel K_{AX}). \tag{3.8}$$

An acknowledgment MAC μ_a to acknowledge a time-stamp MAC μ_t is computed by B as

$$\mu_a(B, X) = h(t_x \parallel t_a \parallel 1 \parallel K_{AX}). \tag{3.9}$$

The shared pairwise secrets used for computing MACs μ_r, μ_t and μ_a are typically stored in the neighbor table of the TMM.

A self-MAC is computed by a TMM X for verification by itself at a later time. For this purpose a secret known only to the TMM (self-generated inside the TMM) is used. Three types of self-MACs are recognized by TMMs, represented as μ_s, μ_{aux} and μ_{set}.

The self-MAC μ_s is computed by X using a secret S_X known only to X as a response to a valid authentication record submitted to the TMM with a MAC μ_r, under some conditions. For example, in response to an authentication record $\mathbf{A} = [A \parallel h_r \parallel t_a \parallel 0 \parallel \mu_r(A, X)]$ submitted to TMM X, the TMM may compute

$$\mu_s(X) = h(A \parallel h_r \parallel r \parallel S_X). \tag{3.10}$$

where r is a value which depicts the internal state of the TMM. The self-MAC is a memorandum issued by the TMM to itself, bound to a state r. In this particular scenario the memorandum states that "a value h_r was provided by A" when the TMM state was r. This self-MAC is accepted by the TMM only if the TMM is in state r. For self-MACs that should be accepted regardless of TMM state, the self-MAC is issued against a state $r = 0$. This value of r is also stored within a TMM.

Further details about r, and MACs μ_{aux} and μ_{set} are provided in later chapters.

3.5 TMM Functions

In the proposed approach TMMs are housed in MANET nodes. These TMMs are capable of establishing pairwise secrets with each other using strategies which demand low overhead for operations to be performed inside the trusted boundary of TMMs. These pair-

wise secrets are used for computing message authentication codes (MAC). Nodes communicate with their TMMs using fixed and well-defined interfaces - by writing into the input registers of the TMM and reading time-stamped MACs from the output registers of the TMM. Such MACs accompany MANET routing packets sent by nodes.

An important prerequisite for a trustworthy module to *warrant* trust is that the TCB functions executed inside the module are simple, and consequently, easily verifiable. Simple TCB functions can also be implemented as hardwired logic (software-free), thus rendering moot a wide range of attacks that attempt to modify software. It is also desirable that the modules consume as little power as possible, and consequently disseminate negligible heat, as such modules can be physically well shielded from deliberate and accidental intrusions. With these self-imposed limitations on TCB functions, aimed at improving the reliability of TMMs while simultaneously lowering their cost, we seek a set of simple TCB functions for MANETs.

Now we present few TMM interfaces that are exposed to the node housing the TMM. These functions perform generic actions, and hence can be used by a TMM irrespective of the employed protocol.

1. Initialize(): employed by all protocols; This function is used to securely load the protocol parameters to their default values.
2. SendTS() : employed by all protocols;
3. UpdateNeighborTable() : employed by all protocols;

3.5.1 SendTS()

This method is used to compute authenticated time-stamps (μ_t) for all the neighbors listed in the neighbor table $\mathcal{N}$ as shown in equation 3.7. Usually this function is invoked to authenticate periodic HELLO messages sent to all neighbors except for those whose status $l == 3$. The message included in these HELLO messages is the time at which the MACs (μ_t) are computed. The algorithm followed by SendTS() can be seen in Figure 3.5.1.

SendTS() { // create time-stamp MAC for every entry in $\mathcal{N}$
$\mu_t(1) \cdots \mu_t(MAX) = 0$;
$t = t_x$;
FOR $i = 0 to MAX$ // MAX is the maximum size of neighbor table
  $\mathbf{N} = \mathcal{N}(i) = [id_n \parallel K_{pw} \parallel t' \parallel l]$; // row i of $\mathcal{N}$
  IF $((id_n \neq 0) \wedge (l \neq 3))$
    $\mu_t(i) = h(t \parallel t \parallel 0 \parallel K_{pw})$;
RETURN $\mu_t(1) \cdots \mu_t(MAX), t$;
}

Figure 3.1

Function SendTS()

3.5.2 UpdateNeighborTable($\mathbf{A}$)

UpdateNeighborTable() is used to update the neighbor table ($\mathcal{N}$)that is internally stored in the TMM. It can either update existing records based on received packets, or can add/delete neighbor records. The function accepts an authentication record ($\mathbf{A}$) as input.

When the function is invoked to either add or delete neighbor records (identified by $\mu == 0$) the interface first tries to find out the index of the passed neighbor identity (id)

by using an internal function called $findindex()$. A neighbor record $(id \parallel K_{pw} \parallel 0 \parallel 0)$ is added to $\mathcal{N}$, if the requested neighbor is not already present in $\mathcal{N}$ ($findindex()$ returns 0).

However when a record for neighbor id does exists, the TMM treats this as a request to either revoke or remove the neighbor. The TMM would remove neighbor records that either: i) has a status $l \neq 3$, and is expired $(t' + \delta_n + \Delta < t_x)$, or ii) has status $l == 3$, and are expired $(t' < t_x)$. Additionally, the TMM would revoke the neighbor, by setting its status to 3, if: i) the neighbor was not heard for δ_n seconds; the TMM revokes the neighbor for the next Δ seconds, or ii) X requested to revoke the neighbor; the TMM revokes the neighbor for the next 2Δ seconds.

Every packet received by a node X is first submitted to UpdateNeighborTable(), where the corresponding neighbor record of the sender (id) is updated. The TMM X first authenticates the submitted packet by recomputing the value of μ. If verified, it computes an acknowledgment MAC μ_a for information (h_r) received in the packet. Later the TMM modifies the record for neighbor id by updating the status l and latest time-stamp t' fields.

Finally if the packet was received from a bidirectional neighbor ($l == 2$), and if it is neither an acknowledgment ($ACK \neq 1$) nor a time-stamp ($h_r \neq t$) the TMM X computes a self-MAC μ_s for the received value h_r as shown in equation 3.10.

A more detailed description of UpdateNeighborTable() can be found in Figure 3.5.2.

UpdateNeighborTable($\mathbf{A}$) {
//$\mathbf{A} = id \parallel h_r \parallel t \parallel ACK \parallel \mu$
$\mu_a = \mu_s = 0$;
$i = findindex(\mathcal{N}, id)$;
//$\mathcal{N}(i) = [id \parallel K_{pw} \parallel t' \parallel l]$ //stored neighbor record
IF ($\mu == 0$) //no MAC - add or delete neighbor
  IF ($i == 0$) //add neighbor
    $K_{pw} = h(K_X \parallel id) \oplus h_r$) //$h_r$ is MLS public value
    $\mathcal{N}(i) = [id \parallel K_{pw} \parallel 0 \parallel 0]$ //insert record
  ELSE //delete neighbor
    IF ($(t' + \delta_n + \Delta < t_x) \wedge (l \neq 3)$)
      $\mathcal{N}(i) = [0 \parallel 0 \parallel 0 \parallel 0]$; //delete record
    ELSE IF ($(t' + \delta_n < t_x) \wedge (l \neq 3)$)
      $\mathcal{N}(i) = [id \parallel 0 \parallel t' + \delta_n + \Delta \parallel 3]$;
    ELSE IF ($(t' < t_x) \wedge (l == 3)$)
      $\mathcal{N}(i) = [0 \parallel 0 \parallel 0 \parallel 0]$; //delete record
    ELSE
      $\mathcal{N}(i) = [id \parallel 0 \parallel t_x + 2\Delta \parallel 3]$;
ELSE //update neighbor record
  $\mathbf{N}' = \mathcal{N}(i) = [id' \parallel K'_{pw} \parallel t' \parallel l']$
  IF ($\mu == h(h_r \parallel t \parallel ACK \parallel K'_{pw})$
    $t_c = t_x$;
    $\mu_a = h(h_r \parallel t_c \parallel 1 \parallel K'_{pw})$; //compute ACK
    IF ($(t > t') \wedge (ACK == 1)$)
      $\mathcal{N}(i) = [id \parallel K'_{pw} \parallel t \parallel 2]$;
    ELSE IF $(t > t') \wedge (ACK == 0)$
      IF ($l' == 2$) //neighbor already listed as bidirectional
        $\mathcal{N}(i) = [id \parallel K'_{pw} \parallel t \parallel 2]$;
      ELSE
        $\mathcal{N}(i) = [id \parallel K'_{pw} \parallel t \parallel 1]$;
    IF $(ACK == 0) \wedge (h_r \neq t) \wedge (l' == 2)$
      $\mu_s = h(id \parallel h_r \parallel r \parallel S_X)$; //self-MAC
RETURN μ_a, μ_s, t_c
}

Figure 3.2

Function UpdateNeighborTable($\mathbf{A}$)

3.5.2.1 Usage of UpdateNeighborTable()

Consider a scenario where a node X enters a subnet and recognizes the presence of nodes A, B and C within its range. At this point, while node X recognizes its neighbors, the TMM of X does not. The TMM of X recognizes a node A as a neighbor only if a time-stamped and authenticated MAC (authenticated using secret K_{XA}) is provided to the TMM.

Node X uses the UpdateNeighborTable() function to add nodes A, B and C to its neighbor table with status $l = 0$. As the pairwise key for A, B and C is available after this step, now X can invoke SendTS() to obtain time-stamped MACs μ_t, which would authenticate X to these neighbors. On receipt of the HELLO packet, A submits the time-stamped MAC to its UpdateNeighborTable() resulting in the addition of X as a neighbor with status $l = 1$. For every packet submitted to UpdateNeighborTable(), the function returns the corresponding acknowledgment MAC μ_a. Hence A would acknowledge the received HELLO, and now X could list A as a bidirectional neighbor. Finally, an acknowledgment from X to A would list X as a bidirectional neighbor of A.

In the same way, a list of neighbors are maintained in the neighbor table of all TMMs, characterized by the link status and the latest time-stamp. Periodically, nodes may send supercilious HELLO packets to ensure that the TMMs of neighboring nodes recognize their presence. The MACs necessary to authenticate HELLO packets can be obtained by invoking SendTS().

Additionally, the function UpdateNeighborTable() allows a node X to revoke its neighbor (say A). X would revoke a neighbor A when:

- X is experiencing a weak link with neighbor A. Usually X is supposed to hear from each of its neighbors for every δ_n seconds. However when X is experiencing a weak link with A, it might not hear from A for more than δ_n seconds. In this instance X would revoke the neighbor A for the next Δ seconds.
- X identifies that A is misbehaving. Now X can revoke A for the next 2Δ seconds.

X would not send or receive packets from a node it revoked until the revocation timer (stored as t') expires.

3.5.3 Supplementary Functions

Apart from the above explained functions the TMM X has a few internal functions which can be invoked to perform specific reusable tasks. The function $checkbd(id)$ (as shown in Figure 3.5.3) is used to verify whether id is currently a bidirectional neighbor. The function verifies the status of id (whether $l == 2$) and also checks whether id was last heard recently enough ($t_x - t' < \delta_n$). If satisfied, the function returns 1 indicating that id is a verified bidirectional neighbor; else, it returns 0.

Additionally the function $computemacs(h_r, id_p, j)$ is used to compute verifiable MACs μ_r as shown in equation 3.6. When $j == 0$ the function returns MACs for every listed bidirectional neighbor in $\mathcal{N}$ except id_p. However when $j == 1$ it only computes MAC for id_p. Figure 3.5.3 details the pseudo code for this function. In order to verify the bidirectionality status of neighbors this function uses the above mentioned $checkbd()$ interface.

$checkbd(id)$ {
$i = findindex(\mathcal{N}, id)$;
$\mathbf{N} = \mathcal{N}(i) = [id_n \parallel K_{pw} \parallel t' \parallel l]$;
IF $((id == id_n) \wedge (l == 2) \wedge (t_x - t' < \delta_n))$
    RETURN 1;
ELSE
    RETURN 0;
}

Figure 3.3

Function $checkbd(id)$

$computemacs(h_r, id_p, j)$ {
IF $(j == 0)$ //compute macs for all bidirectional neighbors except id_p
    $\mu_r(1) \cdots \mu_r(MAX) = 0$; $t = t_x$
    FOR $i = 1 \cdots MAX$
      $\mathbf{N} = \mathcal{N}(i) = [id_n \parallel K_{pw} \parallel t' \parallel l]$;
      IF $((id_n \neq 0) \wedge (id_n \neq id_p) \wedge checkbd(id_n))$
        $\mu_r(i) = h(h_r \parallel t \parallel 0 \parallel K_{pw})$
    RETURN $\mu_r(1) \cdots \mu_r(MAX), t$;
ELSE //compute MAC only for id_p
    $i = findindex(\mathcal{N}, id_p)$;
    $\mathbf{N} = \mathcal{N}(i) = [id_n \parallel K_{pw} \parallel t' \parallel l]$;
    IF $((id_n == id_p) \wedge checkbd(id_n))$
      $\mu_r = h(h_r \parallel t \parallel 0 \parallel K_{pw})$
    RETURN μ_r, t;
}

Figure 3.4

Function $computemacs(h_r, id_p, j)$

CHAPTER 4

BASIC TCB FOR ON-DEMAND PROTOCOLS

In this chapter we propose a simple and efficient TCB for on-demand Protocols which can be leveraged to improve the performance of MANETs by i) providing assurances that reduce the scope of attacks that can be launched by attackers, and by ii) reducing the overhead required for leveraging the TCB. As explained earlier in the proposed approach simple TCB functions are executed inside trustworthy MANET modules (TMM) housed in every MANET node. We assume that only the TMMs are trusted: the rest of the node - all other hardware and software - are untrusted.

Every routing packet received by a node X provides information about a specific destination. Node X uses this accumulated information to carry out the routing process. In order to ensure security X should be prevented from illegally modifying or creating superfluous routing data. Ideally to satisfy this requirement routing data should be stored within trusted boundaries, and thereby preventing unwanted modifications. However, the TMMs housed in each node are severely resource constrained, and hence internally storing all the gathered routing data is not practically feasible.

Therefore in order to maintain the integrity of routing data every received routing packet is signed by a self-MAC $\mu_s = (h_r \parallel id \parallel r \parallel S_X)$. Whenever X decides to use the routing information (represented by h_r) it has to submit the corresponding μ_s that

authenticates it. In this fashion a TMM would prevent a node from illegally modifying received routing data. Hence a node caches all the received routing information (hash of which is represented as h_r) along with the appropriate MAC μ_s.

For the discussion provided in this chapter the value of the field r is always set to 0, as we assume that the TMM does not change its state. Further details about r are provided in subsequent chapters.

In this chapter we present an interface AtomicRelay() which, along with the other functions explained in Section 3.5, can be used to realize both AODV and DSR on-demand routing protocols. Both the protocols start their route establishment process by broadcasting a RREQ, to which a corresponding RREP is generated either by a destination, or by an intermediate node that has fresh enough route. The main difference between the two protocols is that in AODV a requesting node obtains the hop distance and next-hop information by the end of a successful route establishment; while in DSR the entire path to the destination is known.

In both the protocols prematurely expired paths are handled by RERR packets. The function AtomicRelay() identifies a RERR packet when the value of h_r is extended by the flag INV. A routing packet includes a destination record (DR) $\mathbf{D} = [id \parallel q \parallel m \parallel a \parallel \tau]$, and as explain in Section 3.4.4 the value of $h_r = h(\mathbf{D})$. While forwarding this DR the value of h_r is extended as: $h_r = h(h_r \parallel INV)$.

For the discussion provided in this chapter the packet represented by h_r is a RERR when $INV = 1$.

4.1 AtomicRelay$(\mathbf{D}, id_p, \mu_s, INV, \omega)$

TMM X exposes the function AtomicRelay$()$ so that the node could submit the routing information it wants to send, along with the corresponding MAC μ_s. The inputs accepted by this function are:

1. a DR $\mathbf{D} = [id \parallel q \parallel m \parallel a \parallel \tau]$
2. identity of the provider id_p
3. self-MAC μ_s
4. flag INV, ω

The flag ω indicates the underlined protocol ($\omega = DSR$ or $AODV$). A detailed algorithm for AtomicRelay$()$ is provided in Figure 4.1.

When $\mu_s == 0$ the TMM X interprets this as a request to create a DR about itself. Now the internally stored sequence number Q_x is incremented by 1, and later a DR for X is constructed. Later the TMM computes the value of h_r and invokes the function $computemacs(h_r, X, 0)$ to authenticate it to all the stored bidirectional neighbors.

If the function is invoked to relay a previously received information ($\mu_s \neq 0$), the TMM X first computes the value of h_r and verify it against the submitted self-MAC μ_s. When id_p (the provider of h_r) is still a bidirectional neighbor the TMM X builds a new DR by updating the values of m and a. When the implied protocol is DSR ($\omega == DSR$) the value of a is extended to include the identity of X. Finally the corresponding h_r of the newly constructed DR is authenticated to all the bidirectional neighbors of X except id_p.

However when id_p is no longer a bidirectional neighbor of X, the TMM sets $m = INF, a = 0$. The value of h_r is computed by setting the flag $INV = 1$, and later is authenticated to all the current bidirectional neighbors of X.

AtomicRelay($\mathbf{D}, id_p, \mu_s, INV, \omega$) {
//$\mathbf{D} = id \parallel q \parallel m \parallel a \parallel \tau$
IF ($\mu_s == 0$) //create a DR for relaying
$q_x = Q_x + +$; //Q_x is maintained in persistent storage
IF ($\omega == DSR$) $a = h(a \parallel X)$
$h_r = h(X \parallel q_x \parallel 0 \parallel a \parallel t_x + \Delta)$
$h_r = h(h_r \parallel 0)$
RETURN $computesmacs(h_r, X, 0)$; //compute MAC for all bidirectional neighbors
$h_r = h(h(\mathbf{D}) \parallel INV)$
IF ($\mu_s == h(h_r \parallel id_p \parallel 0 \parallel S_X)$) //relay a DR
IF ($checkbd(id_p)$)
IF ($m \neq INF$) $m = m + 1$;
IF ($\omega == DSR$) $a = h(a \parallel X)$;
$h_r = h(h(\mathbf{D}) \parallel INV)$;
ELSE //RERR creation - neighbor lost
$m = INF$; $a = 0$;
$h_r = h(h(\mathbf{D}) \parallel 1)$; //indicates that this is a RERR
RETURN $computesmacs(h_r, id_p, 0)$; //MACs for all neighbors except id_p
}

Figure 4.1

Function AtomicRelay($\mathbf{D}, id_p, \mu_s, INV, \omega$)

4.2 Realizing AODV and DSR

When TMM enabled nodes are used in a MANET subnet employing AODV/DSR the only difference is that along with plain routing packets, every node sends some additional values - a clock-tick value and some MACs. RREQ and RERR packets are accompanied by one MAC for every bidirectional neighbor. RREP and data packets are accompanied by one MAC for the next hop.

HELLO packet exchange and neighborhood establishment are performed as explained in Section 3.5.2.1 by using the functions UpdateNeighborTable() and SendTS().

A RREQ packet contains information about its creator (source). It would also list details about the required destination (like its identity and last known sequence number), which are unmodified while forwarding a RREQ. Let h_i represent the hash of such constant fields. To create a RREQ for destination D, node X invokes the function AtomicRelay() with $\mu_s = 0$, $a = h_i$, and $INV = 0$. The function would construct a DR about X (with $a = h_i$) and compute verifiable MACs to all the listed bidirectional neighbors of X. Note that the TMM increments the internally stored sequence number Q_x while creating such a DR. Therefore we ensure that the sequence number of a node is incremented whenever it requests an RREQ. This would prevent a bad node from sending un-wanted RREQs that will increase network overhead.

Now suppose A is a neighbor of X that has received the RREQ from X and does not have a path to D. Further B is a neighbor of A (but not a neighbor of X) that has a valid path for D (say B received this information previously from its neighbor P). Firstly A would invoke the function AtomicRelay(), and submit the received RREQ and the cor-

responding self-MAC (A would obtain the self-MAC when it first submits the received packet to its UpdateNeighborTable()). The TMM A would generate verifiable MACs that could be used for relaying the RREQ. Now when B receives the RREQ it would create a RREP by submitting the previously received routing information from P and the corresponding μ_s value to its AtomicRelay(). The TMM B would now increment the value of m and returns verifiable MACs (μ_r) to all the neighbors of B. However, since RREP is unicast, B would only include the MAC verifiable by A, and ignore the rest of the μ_r values returned by AtomicRelay().

It has to be noted that even though B created the RREP it actually is relaying the information it previously received from P. Node B could only do this if:

- the information has not expired yet,
- and P is still present as a bidirectional neighbor to B.

For instances where a destination (say D) needs to create a RREP, the function AtomicRelay() is invoked with $\mu_s = 0$, and $a = h_i$. The value h_i is a one-way function on all the constant values included in the RREP. The TMM D would create DR about itself and would authenticate it to all the listed bidirectional neighbors. As explained earlier, since RREP is a unicast, D would only include the MAC verifiable by the node to whom the RREP is forwarded to.

In addition to the process followed above, for DSR ($\omega == DSR$) the value of a is extended to include the node's identity while forwarding DRs. This ensures that a represents the cumulative hash of the source path included in DSR packets. For instance in the above example the source path included in the RREQ received by B is (X, A), and

the value of $a = h(h(h_i \parallel X) \parallel A)$, where h_i is hash of the immutable fields included by the node X that created this packet.

The value of h_r in all the above mentioned calculations is computed with the flag $INV = 0$, as the corresponding DR does not represent a RERR. Now suppose if B lost its connectivity to P, it (node B) is required to generate a RERR for destination D. B would do this by invoking AtomicRelay(), and resubmit the DR (it received from P), and the corresponding μ_s to authenticate this DR. TMM B determines that P is no longer a bidirectional neighbor, and will modify the DR by setting $m = INF, a = 0$. This modified DR is authenticated it to all the neighbors of B. Node A would forward this RERR by submitting the received DR to its AtomicRelay() with $INV = 1$.

4.3 Security Offered by Proposed TCB

In this section we investigate the security offered by the TMMs. We follow an informal approach, by explaining these assurances using natural language. We first establish assertion statements (as Lemmas), and later use these statements to prove the various security features offered by the TMMs.

4.3.1 Assertion Statements

Lemma 1. A node X can not obtain the secrets K_X and S_X that are internally stored inside the TMM X

Proof. The TMMs are assumed to be read-proof and write-proof. To justify the ability to read-proof we deliberately reduce the complexity of operations performed inside the boundary to ensure unconstrained physical shielding. □

Lemma 2. A node X can not directly change the internally stored neighbor table $\mathcal{N}$

Proof. The neighbor table $\mathcal{N}$ is stored internally in TMM X. As explained in **Lemma 1**, the TMMs are well shielded, and hence any data structure that is internally stored cannot be directly accessed by node X. □

Lemma 3. A node X can not modify the received destination record (DR), before submitting it to its TMM interface UpdateNeighborTable().

Proof. Each routing packet sent in the network includes a MAC μ_r that is used to authenticate the information (DR) conveyed in the packet. As shown in equation 3.6, μ_r is calculated over h_r (hash of the DR) using the shared secret K_{pw}. Additionally node X has to submit every received packet (h_r) to the function UpdateNeighborTable(), which verifies h_r by recomputing the value of μ_r.

Since node X does not know the value of K_{pw} (the secret used in computing μ_r), any changes to the received packet will be identified by the function UpdateNeighborTable(), and hence will not be accepted. Therefore node X can not modify the received packet before submitting it to the function UpdateNeighborTable(). □

Lemma 4. A node X can not modify a DR signed by its TMM before relaying it.

Proof. The proof for this statement is similar to the one presented for **Lemma 3**. A TMM signs a DR using the shared key K_{pw}. Any changes to this DR, by node X, would be identified by the TMM of the receiving node, and hence would be dropped. □

4.3.2 Assurances Offered by TMMs

Theorem 4.3.1. A node X can not make illegal changes to received routing information.

Proof. The routing information (DR) received in a packet is authenticated using the MAC μ_r. **Lemma 3** states that X can not modify received DRs before submitting them to its TMM. Further, the TMM has an interface that handles all the required changes that need to be made to the received DR. In our case the function AtomicRelay() makes these changes. Finally after making the changes, the TMM signs the modified DR, and return the corresponding MACs (μ_r) to the node X.

Lemma 4 states that a node can not modify DRs that are signed by its TMM. Hence node X is inhibited from introducing any illegal changes to the received DR. □

Theorem 4.3.2. The TMM X only accepts routing packets from bidirectional neighbors of node X.

Proof. Every packet received by X should be first submitted to the function UpdateNeighborTable(). This function is encoded in such a way that it only outputs the self-MAC μ_s when the packet was received from a bidirectional neighbor. Function like AtomicRelay(), which are used for further processing the received DR, requires μ_s as its input. They use it to authenticate the received DR.

Further, **Lemma 2** states that X can not modify the internally stored neighbor table $\mathcal{N}$. Hence routing packets are only accepted from bidirectional neighbors. □

Theorem 4.3.3. Sending a false RERR packet would penalize node X.

Proof. The DR required to create a RERR message can only be produced by calling the function AtomicRelay(). To obtain this DR node X has to submit a DR with finite height ($m < INF$), and whose provider (say A) is no longer listed with a status $l == 2$ in the neighbor table $\mathcal{N}$.

Node X could generate a fictitious RERR, by manually deleting the bidirectional link to A. X could do this by calling the function UpdateNeighborTable() as described in Section 3.5.2. However doing so would mean that X can not use A as a bidirectional neighbor for a period 2Δ. The risk of loosing a neighbor (and thereby links to other nodes through that neighbor) can be an effective deterrent against attacks based on creating superfluous RERR messages. □

Theorem 4.3.4. TMMs does not allow nodes to collude together.

Proof. **Lemma 1** states that a node can not expose the secrets that are stored within its TMM. Further **Theorem 4.3.1** proved that a node can not introduce illegal changes to received routing packets. Hence TMMs prevent nodes from colluding together. □

The above mentioned security goals, viz.,

1. ensuring that routing records cannot be modified illegally
2. ensuring that routing records will be accepted only from nodes with tested bidirectional links and will be relayed only to nodes with tested bidirectional links,

3. regulating RERR creation, and
4. addressing collusion based attacks

are achieved by performing trivial operations inside the TMM.

A TMM can only make a node to accept authenticated DRs, and ensure that changes to received DRs are made adhering to the protocol. However, the TMM cannot force the node to broadcast a packet as the communication interfaces are not included within trusted boundaries. Hence explicit provisions are required to discourage nodes from maliciously dropping packets. It is for this purpose we provide a node the ability to revoke a neighbor from its neighbor table $\mathcal{N}$. Nodes which identify selfish behavior by neighbors can simply remove the identified selfish neighbor. Therefore the selfish node would loose connectivity with these nodes for the next Δ or 2Δ seconds. Hence, unwarranted dropping of packets can result in a node being disconnected from the network as the neighbors of the node would remove the node from their neighbor list.

In other words, *active* attacks (involving illegal modifications to routing packets) are addressed by TMMs by ensuring that the rules governing the underlying routing protocol cannot be violated. Passive attacks involving selective or selfish participation are addressed by the nodes through their ability to eject nodes from their "logical neighborhood" - the neighborhood as seen by the TMM of the node.

4.3.3 Limitations

In DSR a node gets to know the entire path to the destination by examining the included source path. This additional information can be used to reduce the overhead created by

future RREQ packets. Even though the proposed scheme provides RREP creation by intermediate nodes, it does not support the usage of cached source paths in DSR.

Moreover, due to the physical limitations of a TMM accumulated routing data has to be stored outside the trusted boundaries. Hence even though a node cannot alter received information, it can hide the presence of a specific record. For instance in the example mentioned in Section 4.2 B could hide the presence of the valid DR for D (which it previously received from P), and forward the RREQ it received from A. The TMM B cannot identify this selfish behavior, as it has no information about the records currently stored at the node B. Node B would do this in order to prevent its presence in the final path between X and D, and thereby can skip forwarding unrelated traffic. This would allow B to preserve its resources from forwarding others packets.

Apart from hiding received DRs a node could also advertise stale routing information. For instance say an attacker X has a valid DR ($\mathbf{D} = [D \parallel q \parallel m \parallel a \parallel \tau]$), and the corresponding MAC μ_s for destination D, which was provided by neighbor A. Now when X receives a RERR from A, it would forward the RERR after invalidating the stored DR as $\mathbf{D}'$ by setting $m = INF$ and $a = 0$. However the TMM cannot remember this change of information since the DR $\mathbf{D}$ is not stored within trusted boundaries. Therefore when X later on receives a RREQ for D, it can continue to use the stale information it has for D ($\mathbf{D}$, μ_s). X can continue to use this stale DR until the expiry of its validity time.

In oder to prevent above mentioned malicious behaviors, a TMM should have the knowledge of the destination records currently stored at the node housing it. Hence before forwarding a DR for a destination (say D) a node has to present to its TMM the existing

DR for D. Enforcing such a rule would prevent nodes from hiding, or replaying stale DRs. Additionally, to support the usage of cached routes the TMM should also be able to verify the auxiliary information (source paths) included in the routing packet. In this regard we designed a data structure "Index Ordered Merkle Tree (IOMT)" which could be used to securely maintain the integrity of dynamic database of records that are stored in an untrusted location.

Every leaf of IOMT corresponds to a unique destination, and a parent node is computed as a one-way function of its children. Hence the root of the tree binds all the DRs stored as leaves, and is stored inside the TMM. Whenever a node desires to advertise a DR it would submit the corresponding leaf, and a set of "instructions" that map the leaf to the root. Since the value of the root is stored within trusted boundaries, the TMM can validate the submitted DR by mapping it to the root using the provided instructions. A detailed description of IOMTs is provided in the next chapter.

CHAPTER 5

INDEX ORDERED MERKLE TREE

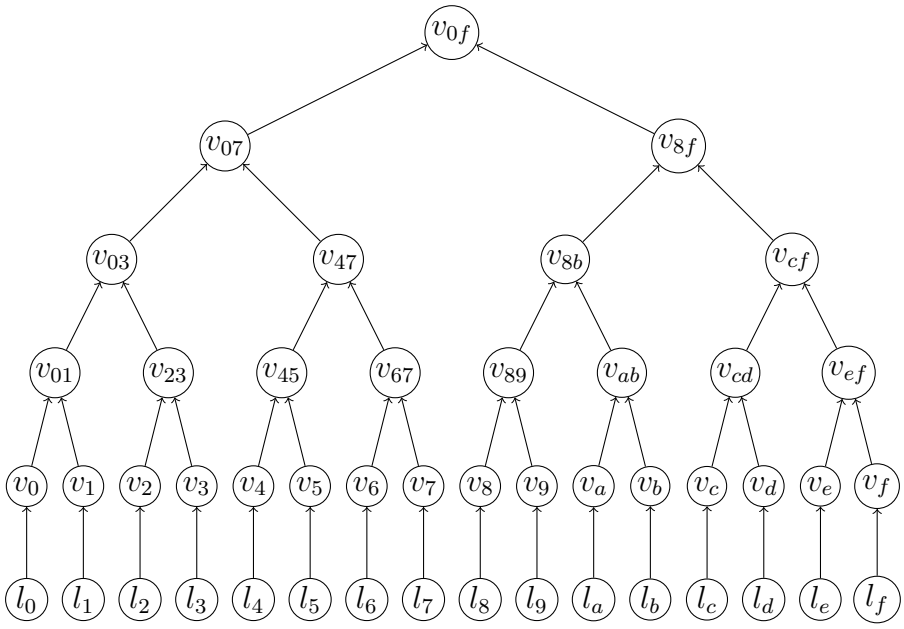

Figure 5.1

A Binary Merkle tree with 16 leaves.

A binary Merkle-tree [59] is a hash tree where a single value (root of the tree) can be used to authenticate multiple values (leaf nodes of the tree). A merkle tree is constructed using two functions both of which can be derived from a cryptographic hash function $h()$ (say SHA-1). A function $v_i = h_l(l_i)$ maps a leaf value l_i to an intermediate node v_i at

height 0; a function $z = h_e(x, y, f)$ maps two intermediate nodes at height $n - 1$ to their parent node at height n using the flag f.

Figure 5 depicts a merkle tree with $N = 16$ leaves and height $L = 4$. As mentioned above a verifier (that has access only to the root v_{0f}) can confirm the existence of a leaf l_i, when provided with the value of the leaf l_i, and a set of **v** "instruction" (which are used to re-compute the root).

For example to prove that l_4 is part of the tree, a prover provides $L = 4$ instructions $(v_5, 0), (v_{67}, 0), (v_{03}, 1)$ and $(v_{8f}, 0)$. Each instruction has two values with the first being a hash, and the second value is an "order-bit" which specifies how to use the included hash. A verifier first computes $v_4 = h_l(l_4)$, and later derives the following values $x = h(v_4 \parallel v_5)$, $x = h(x \parallel v_{67})$, $x = h(v_{03} \parallel x)$ and $x = h(x \parallel v_{8f})$. Note that the order-bit specifies the order of the concatenation. The final value is compared against the root stored in the verifier to confirm the value of the provided leaf l_4.

To change the value of a leaf (say from l_4 to l'_4) a prover submits the following information: i) the new value l'_4, ii) set of **v** instructions, iii) justification for the change. The verifier validates the provided justification, and if satisfied makes the necessary change, and computes the new root as demonstrated above.

5.1 Index Ordered Merkle Tree

In general, a leaf $l_i, 1 \leq i \leq N$ may contain a record with a record-index $i \in \mathcal{I}$ where $|\mathcal{I}| >> N$. For example, $\mathcal{I}$ could be the space of 128-bit IPv6 addresses or 48-bit MAC addresses, but N may be substantially smaller (for example, thousands).

For the cases with $N > |\mathcal{I}|$ each leaf can correspond to a particular ordered-index. For instance, i^{th} leaf can represent an entity with index i. Now in order to prove the presence of an entity with record-index i, a verifier asks for the i^{th} leaf, and the corresponding set of instructions. Similarly to change the i^{th} leaf, a prover submits the new value, set of instructions and justification for the change. After verifying the justification the verifier can update the i^{th} leaf and the final root.

Contrary to the above case, when $|\mathcal{I}| >> N$ only a small subset of values in $\mathcal{I}$ are actually associated with leaves $l_1 \cdots l_N$. Now when a verifier requests for a specific record (say with index i), the prover has to either provide the requested record, or prove that no such record currently exists. Without such a requirement a prover can add multiple records for index i, and can later replay obsolete information about i, which is undesirable.

In general a leaf of IOMT l_i is of the form (i, θ, i'), where i is record-index of the leaf. θ is pre-image resistant hash of the record stored for this index. The third value specifies that the tree does not have records whose indices are enclosed in (i, i').

A value x is enclosed by (i, i') if $i < x < i'$, or if $x < i' < i$, or if $i' < i < x$. If $i = i'$ all values are enclosed - implying that i is the only leaf in the tree. The set of all current leaves are thus virtually ordered based on an index. where each index points to the next available index. The highest index wraps around and points to the lowest index. This approach is used in NSEC records in DNSSEC [64] (domain name system security) for providing authenticated denial of queried records. An NSEC record of the form (`abc.example.com`, `abf.example.com`) proves that no record pertaining to a name `abe.example.com` exists. IOMT combines NSEC and Merkle hash trees.

The root of the IOMT is initially set to zero indicating that all the leaves are empty (initialized to $(0, 0, 0)$). The first inserted leaf points to itself. In order to add a leaf (C, θ_C), the prover has to demonstrate that a record with index C is not present as a leaf. For this it has to provide: i) a leaf that *includes* C (say (A, θ_A, E), ii) an empty leaf $(0, 0, 0)$, and iii) the respective **v** instructions to verify both the leaves. When C is inserted, the leaf of A is changed as (A, θ_A, C) so that it now points to C, and the empty leaf is set to (C, θ_C, E).

Similarly, when deleting a leaf with an index S, the prover needs to provide: i) the leaf of S (say (S, θ_S, T)), ii) the leaf that is pointing to S (say (P, θ_P, S)), and iii) the respective **v** instructions to verify both the leaves. To finish the deletion, the leaf for P is modified as (P, θ_P, T) to point to the leaf that was initially pointed by S, and the leaf of S is emptied to $(0, 0, 0)$.

As can be seen from the above examples, the process of either adding or deleting a leaf requires modifying two leaves simultaneously. Consider a scenario where two leaves (say l_1 and l_6) need to be modified concurrently. Note that the leaves l_1 and l_6 have a common parent (v_{07}) at height $n = 3$. To modify both leaves simultaneously, verifier is provided with

1. the old leaves l_1 and l_6;
2. $n - 1 = 2$ instructions $(v_0, 1)$ $(v_{23}, 0)$ to reach the left-child (v_{03}) of the common parent, starting from $v_1 = h(l_1)$;
3. $n - 1 = 2$ instructions $(v_7, 0)$ and $(v_{45}, 1)$ to reach the right-child v_{47} of the common parent, starting from $v_6 = h_l(l_6)$; the verifier can now compute the common parent $v_{07} = h(v_{03} \parallel v_{47})$.
4. $L - n = 1$ instruction(s) (in this case $(v_{8f}, 0)$) to reach the root from the common parent, where L is the height of the tree; and
5. the new leaves l'_1 and l'_6 (along with the justification for the changes).

After verifying that l_1 and l_6 are part of the tree, and the justification provided, the verifier can now compute a new root starting from values $v'_1 = h_l(l'_1)$ and $v'_6 = h_l(l'_6)$, using the same set of $2 \times (n-1) + L - n$ instructions.

5.2 IOMT Functions

Every routing packet received by a node contains information about a specific destination. A node stores this information for future routing purposes. Additionally apart from routing data, some protocols require nodes to store some auxiliary information. In some instances this auxiliary information reduces the routing overhead, while in other it is mandatory to make routing decisions. For instance, in DSR a node can cache source paths and later use them to respond to future RREQs. This reasonably reduces the overhead incurred by future route creations. On the other hand, in TORA a node is required to store heights of all its neighbors. This information is used to maintain multiple paths, and is required to determine whether a node has lost its last downstream link.

Hence apart from normal routing data in most cases a node is also required to store additional auxiliary information. For this purpose we propose to have two IOMTs: i) main IOMT to store routing data, and ii) auxiliary IOMT to store auxiliary data. The roots of these two trees are stored internally within the TMM.

Apart from the protocol parameters (as specified in Section 3.4.5) the TMM also stores the values: r and r_a, where r is the root of main IOMT, and r_a is the root of auxiliary IOMT. Moreover the leaf of a IOMT is represented as:

$$\mathbf{L} = [i \parallel \theta \parallel i'] \tag{5.1}$$

where i is identity represented by the leaf, θ is the information stored about i, and i' is the next available identity pointed by this leaf. $\theta = 0$ means that no information is stored regarding the identity i, and is termed as an *uninitiated* leaf.

Additionally, the TMM has an internal function $h_e()$ defined as:

$$h_e(x,(y,b)) = \begin{cases} x & \text{if } y = 0 \\ y & \text{if } x = 0 \\ h(x \parallel y) & \text{if } b = 0 \\ h(y \parallel x) & \text{if } b = 1 \end{cases} \tag{5.2}$$

This function is used to map two leaves x and y to their parent, and the value of $h_e(0,(0,b)) = 0$.

5.2.1 Merkle Tree Functions

In this section we present various internal TMM functions that are required to carry out different merkle tree operations. These functions are not exposed to the node, and can only be invoked by the TMM itself. Figure 5.2.1 outlines the steps followed in each of these functions.

The function $h_{ve}(x_1, \mathbf{v} = \{(y_1,b_1)\cdots(y_n,b_n)\})$ is used to map an internal node x_1 to a node at a higher level using $\mathbf{v}$ instructions. Further the function $mapleaf(\mathbf{L},\mathbf{v})$ is used to map the leaf $\mathbf{L}$ to the root, and returns the computed value of the root.

In Section 5.1 we saw that two leaves have to be modified either while inserting or deleting a node. For this purpose the TMM calls the function $mapleaves(\mathbf{L}_l,\mathbf{L}_r,\mathbf{v}_{lr})$. The function first maps the passed leaves to their corresponding intermediate nodes x_l and x_r

respectively by utilizing the function $h_l()$. The passed instruction set $\mathbf{v}_{lr} = \{\mathbf{v}'_l, \mathbf{v}'_r, \mathbf{v}'_p\}$, where $\mathbf{v}'_l$ and $\mathbf{v}'_r$ are the instructions that are used to map the leaves $\mathbf{L}_l$ and $\mathbf{L}_r$ to their common parent. In this regard $mapleaves()$ internally calls the function $h_{cp}(x_l, \mathbf{v}'_l, x_r, \mathbf{v}'_r)$.

Finally the common parent returned by $h_{cp}()$ is mapped to the root by applying the $\mathbf{v}'_p$ instructions.

5.2.2 AddDeleteLeaf$(\mathbf{L}_l, \mathbf{L}_r, id, \mathbf{v}_{lr}, tree)$

The function AddDeleteLeaf() is used to either add or delete a leaf from IOMT. In this regard the TMM utilizes an internal function $checkencloser((i, j), k)$ which verifies if k is enclosed by the values i and j. The pseudo code for the function $checkencloser()$, and AddDeleteLeaf() can be found in Figure 5.2.2.

The inputs to the function AddDeleteLeaf() are thus

1. id, identity that needs to be deleted/inserted.
2. two leaves $\mathbf{L}_l = (i, \theta_i, i')$ and $\mathbf{L}_r = (j, \theta_j, j')$;
3. a sequence of instructions $\mathbf{v}_{lr}$
4. a flag $tree$ that specifies the tree to be operated on

The proposed TMMs have two IOMTs, and the flag $tree = (MAIN$ or $AUX)$ is used to identify the tree that needs to be modified. Later the TMM tries to map the two leaves to the root using the function $mapleaves()$.

Now if the tree is empty ($root == 0$), the TMM adds the first leaf into the tree by modifying the leaf $\mathbf{L}'_l = (id, 0, id)$. However if the tree is non empty, and if the function is invoked to add an identity, the TMM identifies the empty leaf among $\mathbf{L}_l$ and $\mathbf{L}_r$. Assume that $\mathbf{L}_l$ is empty ($i == 0$), now the TMM performs the following steps:

Function $h_{ve}()$ to map an internal node x_0 to a node at a higher level
$h_{ve}(x_1, \{(y_1, b_1) \cdots (y_n, b_n)\})$ {
FOR $i = 1 \cdots n$
    $x_{i+1} = h_e(x_i, (y_i, b_i))$;
RETURN x_{i+1};
}

Function to map a leaf to the root
$r = mapleaf(\mathbf{L}, \mathbf{v})$ {
$x = h(\mathbf{L})$;
RETURN $h_{ve}(x, \mathbf{v})$;
}

Function to map two nodes to a common parent
$h_{cp}(x_l, \mathbf{v}'_l, x_r, \mathbf{v}'_r)$ {
IF $(x_l \neq x_r)$
    $\xi_l = h_{ve}(x_l, \mathbf{v}'_l)$;
    $\xi_r = h_{ve}(x_r, \mathbf{v}'_r)$;
    RETURN $h(\xi_l \parallel \xi_r)$;
ELSE //$x_l == x_r$
    RETURN x_l;
}

Function to map two leaves to the root
$r = mapleaves(\mathbf{L}_l, \mathbf{L}_r, \mathbf{v}_{lr})$ {
$x_l = h_l(\mathbf{L}_l)$;
$x_r = h_l(\mathbf{L}_r)$;
//$\mathbf{v}_{lr} = \{\mathbf{v}'_l, \mathbf{v}'_r, \mathbf{v}'_p\}$
$p = h_{cp}(x_l, \mathbf{v}'_l, x_r, \mathbf{v}'_r)$
RETURN $h_{ve}(p, \mathbf{v}'_p)$;
}

Figure 5.2

Merkle Tree Algorithms

1. verify that (j, j') encloses id (to ensure that no leaf for id exists currently);
2. create leaf $\mathbf{L}'_r = (j, \theta_j, id)$ to replace $\mathbf{L}_r$;
3. create leaf $\mathbf{L}'_l = (id, 0, j')$ to replace $\mathbf{L}_l$;
4. update root.

When the function is invoked to delete id, the TMM first identifies the leaf that's representing id. Suppose $\mathbf{L}_r = (j = id, \theta_j, j')$ should be replaced with $\mathbf{L}'_r = (0, 0, 0)$. Now the steps taken by the TMM are

1. verify $\theta_j = 0$ and $i' = id$;
2. create $\mathbf{L}'_r = (0, 0, 0)$ and $\mathbf{L}'_l = (i, \theta_i, j')$ to replace $\mathbf{L}_r$ and $\mathbf{L}_l$;
3. update root.

TMM ensures that it can delete only uninitiated leaves ($\theta = 0$).

Whenever a node X gets information about a new identity (say id) that is not already present in its IOMT, the function AddDeleteLeaf() is invoked to create a place-holder for id. AddDeleteLeaf() adds an uninitiated leaf $\mathbf{L}$ (whose $\theta = 0$) to the IOMT to represent id. The TMM needs to offer additional functions, that can be used by X, to later update $\mathbf{L}$ to represent the newly arrived information about id.

Similarly when the information stored for the identity id has expired, the TMM offers functions which would uninitiate the leaf representing id. Now X can invoke AddDeleteLeaf() to delete this uninitiated leaf.

Further details on these additional TMM functions are presented in subsequent chapters.

$checkencloser((i,j),k)$ {
RETURN $((i<k<j) \vee ((i>j) \wedge (k>i)) \vee ((i>j) \wedge (k<j)))$;
}

AddDeleteLeaf$(\mathbf{L}_l, \mathbf{L}_r, id, \mathbf{v}_{lr}, tree)$ {
//$\mathbf{L}_l = [i \parallel \theta_i \parallel i']$;
//$\mathbf{L}_r = [j \parallel \theta_j \parallel j']$;
IF $(tree == MAIN)$
  $root = r$;
ELSE
  $root = r_a$
$x = mapleaves(\mathbf{L}_l, \mathbf{L}_r, \mathbf{v}_{lr})$;
IF $(x == root)$
  IF $(root == 0)$ //inserting first leaf
    $\mathbf{L}'_l = [id \parallel 0 \parallel id]$
  ELSE IF $(i == 0) \vee (j == 0)$ //inserting a leaf
    IF $((i == 0) \wedge checkensloser(j, j', id))$
      $\mathbf{L}'_l = [id \parallel 0 \parallel j']$
      $\mathbf{L}'_r = [j \parallel \theta_j \parallel id]$
    ELSE IF $(j == 0) \wedge (checkensloser(i, i', id))$
      $\mathbf{L}'_l = [i \parallel \theta_i \parallel id]$;
      $\mathbf{L}'_r = [id \parallel 0 \parallel i']$;
  ELSE //deleting a leaf
    IF $(id == i) \wedge (\theta_i == 0) \wedge (j' == i)$
      $\mathbf{L}'_l = [0 \parallel 0 \parallel 0]$;
      $\mathbf{L}'_r = [j \parallel \theta_j \parallel i']$;
    ELSE IF $(id == j) \wedge (\theta_j == 0) \wedge (i' == j)$
      $\mathbf{L}'_l = [i \parallel \theta_i \parallel j']$;
      $\mathbf{L}'_r = [0 \parallel 0 \parallel 0]$;
$root = mapleaves(\mathbf{L}'_l, \mathbf{L}'_r, \mathbf{v}_{lr})$;
IF $(tree == MAIN)$
  $r = root$;
ELSE
  $r_a = root$;
}

Figure 5.3

Functions for Adding and Deleting Leaves

CHAPTER 6

SECURING ON-DEMAND PROTOCOLS USING IOMT

MANET nodes accumulate routing information that needs to be stored within them self. In this chapter we present the idea of using IOMTs to prevent a node from misrepresenting or hiding this accumulated information. As explained above, during routing process nodes gather two types of information: i) routing data, and ii) auxiliary data. Therefore we proposed the idea of having two trees, where the main IOMT can be used to store routing data, while the auxiliary IOMT handles auxiliary information.

In this chapter we propose the design of TMMs that can be used to secure AODV and DSR routing protocols. Hence the auxiliary information stored in a node corresponds to source paths received in DSR routing packets. In DSR every node appends it address before forwarding a routing packet. Therefore, a node is aware of the entire path taken by the received packet. This path information included in routing packets is termed as source paths ($\mathbf{S}$). Nodes can use this data to respond to future RREQs, and thereby considerably reducing the overhead incurred by those requests.

6.1 IOMT Design for TMM

In this section we provide more details about the main and auxiliary IOMTs that are used to store accumulated information.

6.1.1 Main IOMT

In general, every routing packet contains information about a specific destination. This information is extracted as a destination record (DR), and as shown in Equation 3.5 a DR for D is of the form: $\mathbf{D}_D = [D \parallel q \parallel m \parallel a \parallel \tau]$. These accumulated DRs are stored as leaves of main IOMT. Each leaf corresponds to a unique destination identity, and the leaf representing D is of the form:

$$\mathbf{L}_i = [D \parallel \theta_D \parallel D'] \tag{6.1}$$

where D' is the next available destination pointed by this leaf, and

$$\theta_D = h(h(\mathbf{D}_D) \parallel id_p) \tag{6.2}$$

where id_p is the neighbor that provided this DR.

6.1.2 Auxiliary IOMT

The auxiliary IOMT is used to hold gathered source paths. Consider a packet that has traversed through nodes A, B, C and reached a node X. Now the value a included in the DR received by X is of the form:

$$a = h(h(h(h_i \parallel A) \parallel B) \parallel C) \tag{6.3}$$

where h_i is hash of the protocol constants included in the packet by node A. Hence for DSR, the value a is a cumulative hash of the source path included in the packet. Therefore we design our auxiliary IOMT to hold this hash value, and each leaf corresponds to a unique value of a.

The leaf representing the source path received in the DR for D ($\mathbf{D}_D = [D \parallel q \parallel m \parallel a \parallel \tau]$) is of the form:

$$\mathbf{L} = [a \parallel (\theta = \tau) \parallel a'] \tag{6.4}$$

where a' is the cumulative hash that represents the next available source path. Additionally, the expiry period of the source path represented by a is the expiry time of the DR that delivered a (which is τ). Hence this value is stored in the leaf as θ.

6.2 Usage of Source Paths

Cached source paths can be used to respond to future RREQs. However RREPs generated by intermediate nodes using stored source paths cannot include the sequence number of the destination. In MANETs sequence numbers are used by nodes to identify latest information. Usually information associated with a greater sequence number are preferred, and hence every RREP generated in the network is associated with the latest sequence number of the destination. Contrary, intermediate nodes generating replies based on obtained source paths cannot provide this information.

Considering this limitation, and to make use of the optimizations provided in DSR, a node will unicast the received RREQ towards the destination, along the known source path. Unicasting an RREQ (rather than broadcast) would considerably reduce the bandwidth it consumes to reach the listed destination. On receiving this RREQ a destination would generate a RREP that includes its latest sequence number.

Now consider an example where node X has a source path $\mathbf{S} = (P, Q, R, S, T)$, and it receives a RREQ for destination Q. Ideally we want X to respond back with a RREP based

on the stored path $\mathbf{S}$. Since X can not provide the sequence number of Q, we propose that X unicast the RREQ along the path T, S, R, Q to reach the destination.

Therefore, X will unicast the RREQ to T, and within the RREQ includes two values: i) the path that can be used by T (which is (S, R, Q), and ii) the length of this path (in this case 3). Node T on receiving this RREQ is forced to either use the path specified by X, or one that has fewer hops (say T has a different source path $\mathbf{S}' = (P, Q, A, T)$, where the number of hops to reach Q is only 2). In this fashion the RREQ is unicast to destination Q, which on receiving the packet responds back with a RREP that has its latest sequence number.

However in scenarios where a node receives an RREQ unicast, and when the suggested path is no longer valid, the node broadcasts the RREQ. Say in the previous example T lost its connectivity to S, and therefore the path suggested by X is no longer valid at T. Now T would broadcast the RREQ it received from X.

Each leaf of the auxiliary IOMT represents a source path that details information about multiple nodes. For instance in the above example at node X, the leaf representing $\mathbf{S}$ provides information about nodes: $P, Q, R, S,$ and T. As stated above, these leaves are indexed based on the cumulative hash a, rather than the identities of the nodes included in the source path. Hence, it would be computationally impractical for a TMM to answer the following question: "is there a valid leaf that represents a source path, which provides information about the requested destination?". In order to securely answer this question, the TMM has to scan through all the leaves searching for the requested destination identity, for every received RREQ packet.

Therefore a feasible solution is to present a choice on the usage of source paths. In the previous example node X can also broadcast the RREQ (requesting for Q) without using the stored source path. Nevertheless, when a node decides to unicast a RREQ, every node down the line is forced to unicast it as long as the suggested source path is valid. As shown in the above case, node T receiving the RREQ unicast from X is bound to either use the source path listed in the packet, or one that has fewer hops. Hence, the initial trigger for a RREQ unicast is optional, but a node that initiates this process forces all the subsequent nodes to unicast the RREQ.

Finally, it is to be noted that a node would look into available source paths only when it *does not have a valid destination stored for the destination* (DR with $m < INF$). A node has to respond with a RREP to the received RREQ (irrespective of whether the RREQ was received either by a broadcast or unicast) if it has a valid height to the requested destination.

6.3 TMM Data Structures

Apart from the parameters defined in Section 3.4.5, the TMM also holds two values: i) r: root of main IOMT, and ii) r_a: root of auxiliary IOMT. The values of r and r_a are initially set to 0 (which signifies that they are empty), and their value keeps changing as information is added into the corresponding IOMT.

In Section 3.4.6 we defined that the self-MAC $\mu_s(X) = h(id_p \parallel h_r \parallel r \parallel S_X)$ is bound to the current state of the TMM. We define the state of the TMM based on the routing information currently stored at the node. Since the basic TCB, proposed for securing on-

demand protocols (design presented in chapter 4), does not contain any information about the currently stored routing data we assumed that the value of $r = 0$. However when designed based on IOMT, a TMM stores the root of the main tree that contains all the currently stored destination records (DRs). Hence, the value of $\mu_s(X)$ is computed over the root of the main IOMT r, and is considered valid until r is unchanged.

6.3.1 Self-MAC μ_{aux}

Along with the specifications provided in Section 3.4, the TMM issues a new self-MAC μ_{aux} which is utilized for securely unicasting RREQ packets. Its value is computed by TMM X as:

$$\mu_{aux}(X) = h(id \parallel id_n \parallel n \parallel a_m \parallel r_a \parallel S_X) \tag{6.5}$$

where id is identity of the destination, id_n is the node to whom RREQ should be unicast, a_m is a cumulative hash of the path suggested by X to id_n, and n is hop distance between id and id_n while using the path represented by a_m. Unlike μ_s, μ_{aux} is tied to the root of auxiliary IOMT r_a, and is valid until r_a is unchanged.

The INV flag was previously used to distinguish between RERR and normal routing packets. However in this chapter, when $INV = 1$ it represents that the received packet is a RREQ, and that it was unicast by the sender. Hence in the example shown in Section 6.2, when X decides to unicast the RREQ, it has to submit to its TMM the self-MAC $\mu_{aux}(X) = h(id = Q \parallel id_n = T \parallel n \parallel a_m \parallel r \parallel S_X)$, where a_m is cumulative hash of the source path that should be present at T. Now TMM X would construct a DR for Q as $\mathbf{D}_Q = [Q \parallel q \parallel m = n \parallel a = a_m \parallel \tau]$. Finally the value of h_r is computed as:

$h_r = h(h(\mathbf{D}_Q) \parallel INV = 1)$. Hence $INV = 1$ indicates that the DR is sent via unicast, and that the values of a and m received in the DR represent the suggested path and its length respectively.

Now suppose that T receives the RREQ unicast (sent by X), and that it does not have the requested path. To represent this state, the TMM T would compute $\mu_{aux}(T)$ with $id = Q, id_n = n = 0$, and $a = a_m$. Such a self-MAC indicates that T does not have a valid leaf for the requested value a_m in its auxiliary tree.

6.4 TMM Functions

Apart from the various functions listed in Sections 3.5 and 5.2, the TMM exposes the following interfaces:

1. Update(),
2. Maintenance(),
3. CreateDR(),
4. SendDR(),
5. CheckPath() and
6. SendInvDR()

The function Update() is used to process received DRs with $INV \neq 1$. It updates the leaves of the main and auxiliary trees based on the received DR. Maintenance() is invoked to handle maintenance activities like invalidating stored DRs and auxiliary values. The interface CreateDR() is used to create DRs either: i) for the node itself, or ii) from an uninitiated leaf ($\theta = 0$). Conversely, the function SendDR() is used to relay currently stored DRs. Self-MAC μ_{aux} is created using the function CheckPath(), and the process

of securely unicasting RREQ packets (as explained in Section 6.2) is carried out by the function RelayInvRR().

6.4.1 Update$(\mathbf{D}, \mu_s, id_{pw}, \mathbf{L}_i, \mathbf{v}_i, \mathbf{D}_o, id_{po}, \mathbf{L}_j, \mathbf{v}_j, \omega)$

This function is used by node X to update currently stored information based on the received DR. A detailed description of the algorithm can be found in the Figure 6.4.1.

The inputs accepted by this function are:

1. received DR $\mathbf{D} = [id_w \parallel q_w \parallel m_w \parallel a_w \parallel \tau_w]$
2. provider id_{pw} of the received DR, and the corresponding self-MAC μ_s
3. stored DR $\mathbf{D}_o$, and its provider id_{po}
4. leaf of the main tree $\mathbf{L}_i$ that represents $\mathbf{D}_o$, and the corresponding instruction $\mathbf{v}_i$ to map it to the root r.
5. leaf of the auxiliary tree $\mathbf{L}_j$ that represents a_w, and the corresponding instruction $\mathbf{v}_j$ to map it to the root r_a.
6. a flag ω that represents the underlined protocol.

The function first maps the received DR $\mathbf{D}$ to the self-MAC μ_s. This function only accepts DRs whose $INV == 0$, and hence computes the corresponding $h_r = h(h(\mathbf{D}) \parallel 0)$. Additionally, the function does not accepts DRs that have smaller sequence number. Later, the TMM maps the two leaves $\mathbf{L}_i$ and $\mathbf{L}_j$ to their corresponding roots; and also checks if they indeed represent information about the received destination (id_w) and auxiliary value (a_w) respectively.

When $\omega == DSR$, and the received DR has a finite height ($m_w < INF$), the TMM updates the leaf $\mathbf{L}_j$ that stores information about a_w.

Later the TMM updates the stored DR $\mathbf{D}_o$ based on the received information ($\mathbf{D}$) when:

1. received a greater sequence number,
2. received a equal or better height,
3. received DR from provider ($id_{pw} == id_{po}$),
4. has no information currently stored for id_pw (leaf $\mathbf{L}_i$ is uninitiated with $\theta_i == 0$).

Finally the TMM X updates the value of θ_i to represent the changes done to $\mathbf{D}_o$, and lastly would update the root r.

6.4.2 **Maintenance**$(\mathbf{D}, id_p, \mathbf{L}_i, \mathbf{v}_i, h_i)$

This function is used by a node X to maintain stale information. The pseudo code of the function can be found in Figure 6.4.2. It takes the following inputs:

1. stored DR $\mathbf{D}$
2. provider identity id_p
3. a leaf $\mathbf{L}_i$ and the set of instructions $\mathbf{v}_i$ that map it to the root.
4. a initial hash h_i

When the function is invoked to maintain stored DRs ($h_i == 0$), the TMM X will un-initialize the leaf (sets $\theta_i = 0$) if the DR has expired. Additionally, when the provider of the DR is no longer listed as a bi-directional neighbor, the TMM identifies it as a loss of link. In order to advertise this change the TMM creates a new DR

$$\mathbf{D}' = [id \parallel q \parallel m = INF \parallel a = 0 \parallel \tau] \tag{6.6}$$

and authenticate it to all the neighbors of X. Finally the stored DR is updated by setting $m = INF + 1, a = 0$, and $q = q + 1$. Later the TMM updates the leaf, and finally the root r to reflect these changes to the stored DR.

Update$(\mathbf{D}, \mu_s, id_{pw}, \mathbf{L}_i, \mathbf{v}_i, \mathbf{D}_o, id_{po}, \mathbf{L}_j, \mathbf{v}_j, \omega)$ {
//$\mathbf{D} = [id_w \parallel q_w \parallel m_w \parallel a_w \parallel \tau_w]$ //received record
//$\mathbf{D}_o = [id_o \parallel q_o \parallel m_o \parallel a_o \parallel \tau_o]$ // stored record
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$; //leaf for stored record
//$\mathbf{L}_j = [j \parallel \tau_j \parallel j']$; //leaf from aux tree
$h_r = h(h(\mathbf{D}) \parallel 0)$; //hash of received record (assumed that INV flag is 0)
$z = h(h_r \parallel id_{pw} \parallel r \parallel S_X)$; //verify self-mac
IF $((z \neq \mu_s) \vee (q_w < q))$ RETURN ERROR;
$x = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
$y = mapleaf(\mathbf{L}_j, \mathbf{v}_j)$;
$p = h(h(\mathbf{D}_o \parallel id_{po}))$ //verify θ_i
IF $((x \neq r) \vee (y \neq r_a) \vee ((\theta_i \neq 0) \wedge (p \neq \theta_i)) \vee (id_o \neq id_w) \vee (id_o \neq i) \vee (j \neq a_w))$
    RETURN ERROR;
IF $((m_w < INF) \wedge (\omega == DSR))$ //add aux value to aux tree
    $\mathbf{L}'_j = [j \parallel \tau_w \parallel j']$;
    $r_a = mapleaf(\mathbf{L}'_j, \mathbf{v}_j)$;
IF $((q_w > q_o) \vee (m_w \leq m_o - 1) \vee (id_{pw} == id_{po}) \vee (\theta_i == 0))$
    $q_o = q_w$; $m_o = INF$; $\tau_o = \tau_w$; $a_o = a_w$;
    IF $(m_w \neq INF) \quad m_o = m_w + 1$;
    $h_r = h(\mathbf{D}_o)$; $\theta_i = h(h_r \parallel id_{pw})$;
    $r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
}

Figure 6.1

Function Update()

However when a stored source path needs to be maintained, the value h_i is a cumulative hash of the stored path except the provider id_p. The corresponding leaf is un-initialized either when the path has expired ($\theta \leq t_x$), or provider id_p is no longer listed as a bi-directional neighbor. Finally the TMM X updates the root r_a to reflect these changes.

6.4.3 CreateDR$(\mathbf{L}_i, \mathbf{v}_i, \omega)$

The function CreateDR() is used by node X to create DRs about itself; and if requested, can also create DRs about destinations that have uninitiated leaves (X has no information stored for them). The algorithm for the function can be see in Figure 6.4.3. The function takes the following inputs:

1. a leaf $\mathbf{L}_i$ and the set of instructions $\mathbf{v}_i$ that map it to the root r.
2. flag ω that determines the protocol.

When invoked the TMM first increments the stored sequence number Q_x. Further it creates a DR about X, and authenticate it to all the listed bidirectional neighbors.

Additionally, when an uninitiated leaf $\mathbf{L}_i$ is passed as input, the function initializes the DR for the destination identity i. Later it updates the values of θ_i and r subsequently to reflect these changes. Finally the TMM X computes verifiable MACs, for all bidirectional neighbors, over the initialized DR.

6.4.4 SendDR$(\mathbf{L}_i, \mathbf{v}_i, \mathbf{D}, id_p, \omega)$

This function is used to relay stored DRs, and the algorithm followed can be seen in Figure 6.4.4. The inputs accepted by this function are:

1. stored DR $\mathbf{D}$

Maintenance($\mathbf{D}, id_p, \mathbf{L}_i, \mathbf{v}_i, h_i$) {
//$\mathbf{D} = [id \parallel q \parallel m \parallel a \parallel \tau]$ // stored record
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$;
$x = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
$p = h(h(\mathbf{D}) \parallel id_p)$;
IF $(h_i == 0)$ //maintenance of stored routing record
  IF $((x == r) \wedge (p == \theta_i) \wedge (id == i))$
    IF $(\tau \leq t_x)$ //expired record
      $\theta_i = 0$;
    ELSE IF $(checkbd(id_p) == 0)$ //invalid record
      $m = INF + 1$; $q = q + 1$; $a = 0$
      $\theta_i = h(h(\mathbf{D}) \parallel id_p)$;
      $\mathbf{D}' = [id \parallel q \parallel m = INF \parallel a = 0 \parallel \tau]$
      $h_r = h(h(\mathbf{D}') \parallel 0)$;
      $computemacs(h_r, X, 0)$;
    $r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
ELSE //maintenance of aux record
  IF $((x \neq r_a) \vee (i \neq h(h_i \parallel id_p)))$ RETURN ERROR;
  IF $((checkbd(id_p) == 0) \vee (\theta_i \leq t_x))$ //expired or invalid aux record
    $\theta_i = 0$;
    $r_a = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
}

Figure 6.2

Function Maintenance()

CreateDR($\mathbf{L}_i, \mathbf{v}_i, \omega$) {
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$;
$x = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
$Q_x + +$;
IF($\omega == DSR$) $\quad a = h(a \parallel X)$
$\mathbf{D} = [X \parallel Q_x \parallel n \parallel a \parallel t_x + \Delta]$;
$h_r = h(h(\mathbf{D}) \parallel 0)$;
$computemacs(h_r, X, 0)$;
IF $((x == r) \wedge (\theta_i == 0))$
$\quad \mathbf{D} = [i \parallel 0 \parallel INF \parallel 0 \parallel 0]$;
$\quad \theta_i = h(h(\mathbf{D}) \parallel id_p = 0)$;
$\quad r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
$\quad h_r = h(h(\mathbf{D}) \parallel INV = 0)$;
$\quad computemacs(h_r, X, 0)$;
}

Figure 6.3

Function CreateDR()

2. provider identity id_p
3. a leaf $\mathbf{L}_i$ and the set of instructions $\mathbf{v}_i$ that map it to the root.
4. flag ω that determines the protocol.

The TMM X first verifies the leaf $\mathbf{L}_i$ by checking if it indeed represents the values stored in DR $\mathbf{D}$, and later maps it to the root r. If satisfied, $\mathbf{D}$ is authenticated to all the bidirectional neighbors of X, and the value of h_r is computed by setting $INV = 0$. When the protocol is DSR ($\omega == DSR$) and DR has finite height, the stored auxiliary value is extended to include the identity of the node X. Additionally, when $m == (INF + 1)$, the submitted DR was recently modified due to a link loss. Hence in such scenarios, the stored sequence number q is decremented by 1 before sending the DR.

SendDR($\mathbf{L}_i, \mathbf{v}_i, \mathbf{D}, id_p, \omega$) {
//$\mathbf{D} = [id \parallel q \parallel m \parallel a \parallel \tau]$ //stored record
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$; //leaf for stored record
$x = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
$p = h(h(\mathbf{D} \parallel id_p))$ //verify θ_i
IF $((x \neq r) \vee (i \neq id) \vee (p \neq \theta_i)$ RETURN ERROR;
IF $((checkbd(id_p)) \wedge (t_x < \tau))$
  IF $((\omega == DSR) \wedge (m < INF))$
    $a = h(a \parallel X)$;
  $q' = q$;
  IF $(m == (INF + 1))$ $q' = q - 1$;
  $\mathbf{D}' = [id \parallel q' \parallel m \parallel a \parallel \tau]$
  $h_r = h(h(\mathbf{D}') \parallel 0)$;
  RETURN $computemacs(h_r, id_p, 0)$;
}

Figure 6.4

Function SendDR()

6.4.5 CheckPath($\mathbf{L}_i, \mathbf{v}_i, h_i, n, (id_1 \cdots id_n)$)

This function is used by node X to obtain the self-MAC μ_{aux} that is used while unicasting RREQ packets. It accepts the following inputs:

1. a leaf of the auxiliary tree $\mathbf{L}_i = [a \parallel \theta_a = \tau_a \parallel a']$, and the corresponding instructions $\mathbf{v}_i$
2. initial hash h_i
3. a set on n nodes $(id_1 \cdots id_n)$, where id_1 is the required destination, and id_n is the node that provided the auxiliary value a.

The leaf $\mathbf{L}_i$ represents a source path $\mathbf{S}$ whose cumulative hash is the value a. The value h_i is a cumulative hash that represents all the nodes in $\mathbf{S}$ until the destination id_1. For instance if $\mathbf{S} = (P, Q, R, S, T)$ and $id_1 = R$, the value of $h_i = h(h(h_p \parallel P) \parallel Q)$ (h_p is hash of the constants included by P), and the three nodes (R, S, T) are passed as inputs to the function CheckPath().

It can be noted that in the above example $a = h(h(h(h_i \parallel R) \parallel S) \parallel T)$. The TMM uses an internal function $extend(h_i, n, (id_1 \cdots id_n))$ to extend the value of h_i to include the n nodes $(id_1 \cdots id_n)$. The algorithm for $extend()$ and CheckPath() can be found in Figure 6.4.5.

The TMM X first maps the leaf $\mathbf{L}_i$ to the root r_a, using the $\mathbf{v}_i$ instructions. An uninitiated leaf $\mathbf{L}_i$ ($\theta_a == 0$) indicates that X does not have a source path identified by a, and hence creates the self-MAC as:

$$\mu_{aux} = h(id_1 \parallel 0 \parallel 0 \parallel a \parallel r_a \parallel S_X) \tag{6.7}$$

However when $\theta_a \neq 0$, the TMM X extends the value of h_i by the passed n nodes $(id_1 \cdots id_n)$ to verify that $\mathbf{L}_i$ indeed points to the indicated source path. Now the TMM verifies whether the leaf is still valid by checking: i) that the leaf has not yet expired, and ii) the provider id_n is still a bi-directional neighbor. If satisfied, the TMM X identifies that $\mathbf{L}_i$ points to a valid source path that includes the destination id_1. Hence node X can use this path information to unicast a RREQ to the provider id_n, and in this regard computes the self-MAC as

$$\mu_{aux} = h(id_1 \parallel id_n \parallel n-1 \parallel y \parallel r_a \parallel S_X) \tag{6.8}$$

where y is the cumulative hash of the source path that should be used by id_n to forward the RREQ. It can be easily observed that the length of this indicated source path is $n-1$.

$x = extend(h_i, n, (id_1 \cdots id_n))$ {
$x = h_i$;
FOR $i = 1 : n$
  $x = h(x \parallel id_i)$;
OUT x;
}

CheckPath$(\mathbf{L}_i, \mathbf{v}_i, h_i, n, (id_1 \cdots id_n))$ {
//$\mathbf{L}_i = [a \parallel \tau_a \parallel a']$
$x = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
IF $(x == r_a)$
  IF $(\tau_a == 0)$
    $\mu_{aux} = h(id_1 \parallel 0 \parallel 0 \parallel a \parallel r_a \parallel S_X)$;
  ELSE
    $y = extend(h_i, n, (id_1 \cdots id_n))$
    IF $((a == y) \wedge (\tau_a > t_x) \wedge (checkbd(id_n)))$
      $y = extend(h_i, n-1, (id_1 \cdots id_{n-1}))$;
      $\mu_{aux} = h(id_1 \parallel id_n \parallel n-1 \parallel y \parallel r_a \parallel S_X)$;
RETURN μ_{aux};
}

Figure 6.5

Function CheckPath()

6.4.6 **SendInvDR**$(\mathbf{L}_i, \mathbf{v}_i, \mathbf{D}_o, id_{po}, \mathbf{D}_w, INV, id_{pw}, \mu_s, \mathbf{X}, \mu_{aux})$

This function is used by node X when it receives a DR (say for a destination id) with either $m == INF$ or $INV == 1$, and even X has a DR with infinite height stored for id. This function is used to carry out the RREQ unicast process as explained in Section 6.2. The pseudo code followed by the function can be see in Figure 6.4.6.

The inputs accepted by this function are:

1. received DR $\mathbf{D} = [id_w \parallel q_w \parallel m_w \parallel a_w \parallel \tau_w]$
2. provider id_{pw} of the received DR, the corresponding self-MAC μ_s, and the flag INV
3. stored DR $\mathbf{D}_o$, and its provider id_{po}
4. leaf of the main tree $\mathbf{L}_i$ that represents $\mathbf{D}_o$, and the corresponding instructions $\mathbf{v}_i$ to map it to the root r.
5. self-MAC μ_{aux}, and the set $\mathbf{X} = [id \parallel id_n \parallel n \parallel a_m]$ that contains the values used in the computation of μ_{aux}.

The TMM X first verifies the received DR by using the input INV flag. It later maps the provided leaf to the root using the $\mathbf{v}_i$ instructions, and also checks whether the leaf indeed points to the DR $\mathbf{D}_o$. Finally the TMM verifies the self-MAC μ_{aux}, if it is not null ($id \neq 0$), by using the values passed in the set $\mathbf{X}$.

Additionally, this function should only be invoked by X when:

1. received an invalid DR for destination id_w
2. the stored DR for id_w has infinite height, or X has an un-initialized leaf for id_w

The TMM X ensures that these conditions are satisfied by verifying the received and stored DRs.

The TMM uses a flag $iflag$ which when set determines that the received DR should be unicast to id_n (the neighbor indicated in μ_{aux}). When $\mathbf{D}_w$ is received by unicast ($INV ==$

1) the TMM expects a non-NULL μ_{aux} which either specifies: i) a smaller path than m_w (sets $iflag$ to 1), or ii) that the suggested path a_w is not valid (sets $iflag$ to 0).

However when $INV == 0$ and $id \neq 0$, X is requesting to start the unicast process, and hence the TMM sets $iflag = 1$.

Finally if $iflag == 1$ the TMM creates a DR whose $m = n$, and $a = a_m$. TMM X unicast this new DR only to the node id_n. However when $iflag == 0$ the TMM broadcasts a DR, whose $m = INF$ and $a = 0$, to all the bidirectional neighbors.

6.5 Realizing AODV and DSR

In this section we detail how the above mentioned functions can be used by a node X to realize AODV and DSR routing protocols. Firstly, the function UpdateNeighborTable() can be used to maintain connectivity, and to obtain the self-MAC μ_s. MACs for periodic HELLO messages can be obtained from SendTS(). Every packet received by X is first submitted to UpdateNeighborTable().

Additionally, the function AddDeleteLeaf() can be used to create place holders for identities that were recently known to X. Suppose when X receives a DR about destination D for the first time, X would invoke AddDeleteLeaf() to create an un-initialized leaf for D in the main IOMT. This newly created leaf, along with the received DR, can be submitted to Update() for further processing. Similarly when X receives a new source path (whose cumulative hash is not present in auxiliary IOMT), X would invoke AddDeleteLeaf() to create an un-initialized leaf for this newly arrived information.

SendInvDR($\mathbf{L}_i, \mathbf{v}_i, \mathbf{D}_o, id_{po}, \mathbf{D}_w, INV, id_{pw}, \mu_s, \mathbf{X}, \mu_{aux}$) {
//$\mathbf{D}_w = [id_w \parallel q_w \parallel m_w \parallel a_w \parallel \tau_w]$ - received record
// if INV=1 in received record, a path a of length m is suggested to forward the record
//$\mathbf{D}_o = [id_o \parallel q_o \parallel m_o \parallel a_o \parallel \tau_o]$ - stored record
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$; // leaf for stored record
//$\mathbf{X} = [id \parallel id_n \parallel n \parallel a_m]$ //verified path to id from aux tree
IF $((id_o \neq id_w) \vee (i \neq id_o))$ RETURN ERROR;
$h_r = h(h(\mathbf{D}_w) \parallel INV)$
IF $(\mu_s \neq h(h_r \parallel id_{pw} \parallel r \parallel S_X))$ RETURN ERROR;
$x = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$; $p = h(h(\mathbf{D}_o) \parallel id_{po})$;
IF $((x \neq r) \vee (p \neq \theta_i) \vee ((INV == 0) \wedge (m_w < INF)) \vee ((\theta_i \neq 0) \wedge (m_o < INF)))$
RETURN ERROR;
IF $(id \neq 0)$ //self-MAC μ_{aux} is not NULL
IF $(id \neq id_o)$ RETURN ERROR;
IF $((\mu_{aux} \neq h(id \parallel id_n \parallel n \parallel a_m \parallel r_a \parallel S_X)) \vee (checkbd(id_n) == 0))$
RETURN ERROR;
$iflag = 0$;
IF $(INV == 1)$ //path suggested by received record
IF $((id == 0) \vee (n > m_w - 1))$ RETURN ERROR; //no path information provided
IF $((n == 0) \wedge (a_m == a_w))$ //suggested path is not usable
$iflag = 0$;
ELSE IF $(n \leq m_w - 1)$//suggested path or better path
$iflag = 1$;
ELSE RETURN ERROR
ELSE IF $(INV == 0) \wedge (id == id_o) \wedge (n < MAX))$
//previous hop did not, but X is suggesting a path
$iflag = 1$;
ELSE IF $(INV == 0) \wedge (id == 0))$ //broadcast DR
$iflag = 0$;
IF $(iflag)$
$\mathbf{D}' = [id \parallel q_w \parallel n \parallel a_m \parallel \tau_w]$;
$h_r = h(h(\mathbf{D}') \parallel 1)$;
RETURN $computemacs(h_r, id_n, 1)$ //compute only one MAC for id_n
ELSE
$\mathbf{D}' = [id \parallel q_w \parallel INF \parallel 0 \parallel \tau_w]$; $h_r = h(h(\mathbf{D}') \parallel 0)$;
RETURN $computemacs(h_r, id_p, 0)$; //compute MACs for all except provider;
}

Figure 6.6

Function SendInvDR()

When X desires to create a RREQ for a destination (say D) it invokes the function CreateDR(), and passes an un-initialized leaf for D. The function first increments the sequence number Q_x, and later creates a DR about X. Additionally, it also computes a DR for D with an infinite height ($m = INF$). Both these DRs are authenticated to all the bidirectional neighbors of X. Hence in our approach a RREQ packet contains two DRs: i) a DR about the source which includes its latest sequence number, and ii) a DR (with $m == INF$) about the destination. A node A receiving these two DRs would process them separately. While first DR updates the currently stored information about X, the second one is used to forward the received RREQ.

It can be observed that whenever X creates a RREQ its sequence number is incremented. Hence X is charged (by increasing its sequence number) for using the network. This would prevent a node from generating un-wanted RREQs aimed at creating extra traffic to launch denial of service attacks.

Suppose when A does not have a path for D, A would submit the received and stored DR for D to the function SendInvDR(). Since A does not have a path to D, the function would return a DR for D that has infinite height. Additionally, A can get the updated DR about X by calling the function SendDR(), which can be used to relay stored DRs. The RREQ forwarded by A would include these two DRs.

In the above process while relaying the received RREQ, node A is required to also submit the stored DR for D, while calling the function SendInvDR(). This would prevent A from selfishly hiding valid paths for requested destinations.

Now say when A has a path to D, it has to respond back with an RREP. Now A can submit the stored DR for D to the function SendDR(), which would authenticate it to all the bidirectional neighbors of A. Nevertheless, since this RREP should only be unicast back to X, A would only include the MAC μ_r that is verifiable by X, and ignore the rest of the MACs returned by SendDR().

For scenarios where a destination (say D) is required to respond back with an RREP, the function CreateDR() is invoked by passing NULL for all the parameters. This would make CreateDR() to create a DR only about the node.

Finally expired leaves in both main and auxiliary IOMTs can be cleared by calling the function Maintenance().

6.5.1 Creation of RERR

In both AODV and DSR when a node X loose a bidirectional link to a neighbor (say A), X creates a RERR packet for all the DRs provided by A. Lets assume X created a RERR for destination D. In this RERR packet the hop count for the unreachable destination is set to $m = INF$, and the sequence number q is unchanged. However in order to prevent routing loops, on-demand protocols like AODV and DSR require X to increment the sequence number ($q' = q + 1$) stored for D. This ensures that X would only accept RREPs (for D) that has a sequence number greater than q'.

The function Maintenance() is used to carry out the above logic. For the above example X would invoke Maintenance(), and pass the DR for D (say $\mathbf{D}_D$) as input. The TMM X would create a DR with $m = INF$, $a = 0$ and authenticate it to all the neighbors of X.

Further it would increment the sequence number stored in $\mathbf{D}_D$. The TMM X also sets $m = INF + 1$, which indicates that the DR is modified based on a recent link loss at node X. Such an indication is required when X desires to resend the RERR packet. In such a scenario (when X wants to resend the RERR), X would invoke SendDR() to relay $\mathbf{D}_D$. Now the TMM X can verify that $m == (INF + 1)$, and relay a RERR packet (with $m = INF$, and $q = q - 1$) after decrementing the stored sequence number for D.

6.5.2 RREQ Unicast for DSR

Accumulated source paths can be used in DSR to unicast RREQs towards the specified destination. For suppose in the above example say A has a source path (P, D, Q, R, S), and received a RREQ for D from X. Now when A does not have a path for D, it could use the stored source path to unicast the received RREQ towards D. For this purpose A would require to obtain a self-MAC $\mu_{aux} = [D \parallel S \parallel 3 \parallel a_m \parallel r \parallel S_A]$, where S is the node to whom the RREQ should be unicast, a_m represents the cumulative hash of the source path that could be used by S (which is (P, D, Q, R)), and 3 denotes that by using this path S is three hops away from D. A can acquire this certificate by calling the function CheckPath(), and by passing the auxiliary leaf that points to the stored source path (P, D, Q, R, S).

Now A would pass this certificate to the function SendInvDR(), along with the received and stored DR for D. SendInvDR() would authenticate the passed inputs, and compute a DR with $m = 3$, $a = a_m$, and $INV = 1$. By setting the flag $INV = 1$, the TMM A indicates to the receiver of this DR (in our case S) that it (S) should further unicast the

RREQ using a source path that has a path length of at most $m = 3$ hops, and that the suggested path's cumulative hash is a_m.

Now when S receives this RREQ unicast from A, it can either use the suggested path, or one which has fewer hops that 3 (say a source path (X, D, Y), where S is only 2 hops away from D). For the selected path, S would obtain the corresponding self-MAC μ_{aux} by calling the function CheckPath(). Later S would invoke SendInvDR() to further unicast the RREQ along the selected path.

However when S does not have a valid source path to further carry on the unicast process, it has to broadcast the received RREQ. Such a scenario would arise, say when S lost its connectivity with the neighbor R. In this regard, S first has to prove to its TMM that the suggested path (represented by $a = a_m$ in the received DR) is not valid, which means that S has to provide an un-initialized auxiliary leaf for a_m. S could obtain this leaf by calling the function AddDeleteLeaf(). However a leaf for a_m can not be added when S already has an auxiliary leaf representing a_m. This would prevent S from hiding the path suggested in the received DR.

Now S would pass this newly created un-initialized leaf to the function CheckPath(), which issues a self-MAC μ_{aux} that signifies that S does not have the suggested path a_m. Later X would submit μ_{aux} and the received DR to SendInvDR(), where the TMM verifies if the auxiliary values included in both μ_{aux} and the received DR are identical ($a_m == a$). Finally the TMM would return MACs which authenticate the received DR to all the bidirectional neighbors of S. By utilizing these MACs S can broadcast the received RREQ.

6.6 Analysis of IOMT design

In this section we analyze the security offered by TMMs when IOMTs are used to store acquired routing data. The design of the TMMs using IOMT is an extension to the basic TCB proposed in chapter 4. Hence all the security features mentioned in Section 4.3 are also offered when IOMTs are used along with TMMs. Additionally, employing IOMT guarantees that a node can not hide or replay stale information. Further, the proposed design also enables a node to securely use cached source paths to forward future RREQs. In this section we present informal proofs to these additional features.

6.6.1 Assertion Statements

Lemma 1. A modification to the information stored in a leaf, would change the root of the IOMT.

Proof. According to the property of a merkle tree, and therefore even for IOMT, the root of the tree can be used to map all the leaves of the tree. Hence, the root represents the state of the information currently stored in an IOMT. According to its design, every leaf of an IOMT stores information about a specific identity. Whenever this information is modified, even the root has to be updated to reflect these changes.

In the proposed TMM functions, whenever a function updates the information stored by a leaf, the TMM ensures that it also updates the root of the IOMT. □

Lemma 2. In a given IOMT, every leaf represents information about a unique identity.

Proof. A node X has to invoke the function AddDeleteLeaf() to add a leaf that represents an identity (say j) thats not already present in the IOMT. Before making this addition the

function requires an input leaf $\mathbf{L} = (i, \theta, i')$, where (i, i') encloses the newly added identity j. If a leaf for j already exists in the IOMT, it would be impossible for X to produce a leaf that encloses j. Hence the function AddDeleteLeaf() can only add a leaf that represents an identity thats not yet present in the IOMT.

Further as AddDeleteLeaf() is the only function that can be used to add leaves, we can claim that every leaf in an IOMT represents a unique identity. For the main IOMT every leaf represents a DR about a unique destination, while for auxiliary IOMT, every leaf represents a unique source path. □

Lemma 3. While processing a received routing packet about a destination D, the TMM also requires the internally stored DR for D.

Proof. The only two functions that can be used to process received DRs are Update(), and SendInvDR(). While Update() handles received DRs with finite height ($m < INF$) or those sent with $INV == 0$, the function SendInvDR() is used to process received DRs with infinite height ($m == INF$). Both these functions require the internally stored DR as their input, and hence the statement is proved. □

Lemma 4. The self-MAC μ_{aux} can be used by node X to indicate information about a stored source path.

Explanation: The value of $\mu_{aux} == h(id \parallel id_n \parallel n \parallel a_m \parallel r \parallel S_X)$. When $id_n = n = 0$, μ_{aux} indicates that X does not have a valid source path represented by a.

However when $((id_n \neq 0) \wedge (n \neq 0))$, μ_{aux} indicates that X has a valid source path which can be used to reach destination id. Further it conveys that the next node in this path

is neighbor id_n, which should have a source path as represented by a_m, and that by using this indicated path id_n is n hops away from destination id.

Node X can obtain this self-MAC by invoking the function CheckPath(). □

6.6.2 Assurances Offered by the TMMs

Theorem 6.6.1. A node X can not hide or replay stale DRs for a destination D.

Proof. **Lemma 3** states that in order to process a received routing packet, node X has to submit the stored DR it has for D. This would prevent X from hiding information about D, while processing routing packets regarding D.

According to **Lemma 1** whenever TMM X updates the stored DR for D, it would also update the root of the main IOMT. Further, **Lemma 2** states that X can only have a single DR for D currently stored in main IOMT. Therefore X can not replay a stale DR for D. □

Theorem 6.6.2. Suppose node X received hop counts $4, 3,$ and 2 subsequently from neighbors $A, B,$ and C respectively (say for a destination D). After processing all these DRs, in the order they are arrived at X, node X can not advertise a height based on the heights it received either from A or B.

Proof. **Lemma 2** states that X can have only a single leaf in its IOMT that represents destination D. Further according to **Lemma 3**, X has to submit the stored DR for D while processing routing packets about D. Since in this example the DRs received by X has finite heights, the function Update() has to be invoked to process these received DRs.

The function Update() is encoded in such a way that the stored DR is updated (more specifically the values of hop count (m), and provider id_p are changed) whenever X receives a smaller height. As the hop count advertised by C is the smallest, after processing all the three received DRs, the stored DR is updated based on the DR sent by C.

Again according to **Lemma 2** X can have only a single DR stored for D in its main IOMT. Hence the TMM X would update this DR whenever it receives a smaller height. Therefore at the end of processing all the three DRs, the TMM X can only advertise the stored DR for D, which is updated based on the hop count received from C. □

Theorem 6.6.3. For a RREQ packet that is being unicast towards a destination D, at every hop the source path used to forward this RREQ is decreasing in length. In absence of a required source path, the RREQ would be broadcast.

Proof. The function SendInvDR() is used to forward RREQs. When X receives a DR that has its flag $INV = 1$, X is required to submit a μ_{aux} whose $a_m == a$, where a is the auxiliary value received in the DR. When $((id_n \neq 0) \wedge (n \neq 0))$, the TMM X expects a path shorter than the one indicated in the DR. The TMM X would accept μ_{aux} only when these conditions are satisfied, and would forward the DR (that represents the RREQ packet) to id_n with $INV = 1$.

Additionally when $id_n = n = 0$, this indicates that the requested source path is no longer valid at X. Hence the TMM X, would broadcast the DR.

Therefore, whenever the RREQ is unicast the TMM ensures that it is being forwarded in the path whose length is decreasing. But when a valid source path is not available the RREQ is broadcast. □

CHAPTER 7

COLLISION RESISTANT TORA

Broadly the applications of MANETs can be divided into two types: i) those that provide connectivity between any two pairs of nodes, and ii) those that provide connectivity only to a few set of nodes. For example of the former application consider a group of ad hoc devices deployed in a hostile environment that require peer communication. Additionally, a classic example of the later application could be an ad hoc network that is established to extend the reach of a few base-stations. In such a scenario all the nodes in the MANET desire to establish a path to the available base-stations over multiple hops.

Routing protocols like AODV and DSR which establish routes on demand, and provide minimal latency are better choices for applications that require peer communications. However the proactive version of TORA presents itself as a good choice for applications that require the advertisement of only a few nodes.

7.1 Shortcomings of TORA

Consider a simple subnet topology depicted in Figure 7.1.

After the OPT from the destination ϕ is propagated the heights of the node will be: $(rl, 0, \phi)$, $(rl, 1, A)$, $(rl, 2, C)$, $(rl, 2, B)$, $(rl, 3, G)$, $(rl, 3, E)$, and $(rl, 3, F)$, where rl

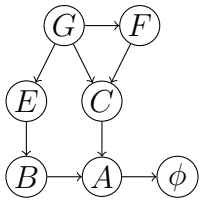

Figure 7.1

A Subnet running TORA

denotes the reference level and is $rl = (0, 0, 0)$. Assume that the link $C \rightarrow A$ goes down at time τ. The sequence of events that will transpire are then as follows:

a) C generates a new RL $rl' = (\tau, C, 0)$ by sending a UPD with height $(rl', 0, C)$; G takes no action as it still has downstream neighbors E and F;

b) F propagates the new RL by sending a UPD with height $(rl', -1, F)$; even after this link reversal G has a downstream neighbor E, hence no action is taken.

Clearly, TORA strives to reduce the number of control packets required to deal with changes in topology, and does this well. Where TORA does not do so well is in scenarios that result in partitioning of the subnet. To see this consider a scenario where A looses its link to destination ϕ at time τ. The sequence of events in TORA will be as follows:

a) $A \rightarrow$ UPD $(rl, 0, A)$ (A generates new RL $rl = (\tau, A, 0)$);

b) $B \rightarrow (rl, -1, B)$; $C \rightarrow (rl, -1, C)$ (B and C propagate new RL);

c) $E \rightarrow (rl, -2, E)$ and $F \rightarrow (rl, -2, F)$ (E and F propagate new RL);

d) $G \rightarrow (rl', 0, G)$ (G reflects RL as $rl' = (\tau, A, 1)$);

e) $F \rightarrow (rl', -1, F), C \rightarrow (rl', -1, C)$ and $E \rightarrow (rl', -1, E)$ (F, C and E propagate reflected RL);

f) $B \rightarrow (rl', -2, B)$ (B propagates reflected RL);

g) $A \rightarrow$ CLR (A detects partition);

h) $B \rightarrow$ CLR; $C \rightarrow$ CLR (propagating CLR in the isolated subnet);

i) $G \rightarrow$ CLR, $F \rightarrow$ CLR; $E \rightarrow$ CLR (propagating CLR in the isolated subnet);

Note that a UPD with a new RL has to reach all nodes in the partition. The RL is then reflected and returned to the originator of the RL, which detects a partition. Following this the originator of the RL creates a CLR packet which needs to be propagated throughout the partitioned subnet. If the average per-hop processing delay is Δ, and L is the longest path in the partitioned subnet, for a time of up to $3L\Delta$ several nodes in the partitioned subnet may possess non-NULL heights which are actually invalid. During this time data packets created by nodes in the partition may be relayed back and forth till the time the nodes realize that they do not actually have a valid height. Another disadvantage of TORA is the need for some mechanism for time-synchronization between nodes to correctly interpret the field τ in an RL.

Additionally, when a node sends a routing packet it is desirable to verify that 1) the packet has reached its neighbors and 2) they have processed the packet in adherence to the protocol. Using acknowledgments satisfies the first requirement, but does not guarantee the second. Making a node respond to every routing packet it receives, will appease both these requirements. This will also alleviate the need for sending explicit acknowledgments, thereby potentially saving network bandwidth. More importantly, by verifying the response, a sender can judge the legitimacy of the responder; if malicious intent is observed such nodes can be avoided from future routes.

TORA does not meet this requirement during its route maintenance. After losing its last downstream link, a node reverses its upstream links (by making its height as local maximum). An upstream neighbor that has a downstream link, even after the reversal, is not required to respond. Now the initiator of the route maintenance cannot differentiate this legitimate behavior from an illegitimate one, where the upstream neighbor simply ignores the received maintenance packet.

Collisions: Perhaps the most acute of TORA's shortcomings is its susceptibility to collisions. Once again consider a scenario above where A reverses its link to C by sending a UPD packet. Now assume that the packet was lost due to collision. At this point A assumes that C is its downstream neighbor, while C assumes that A is downstream, thereby creating a simple loop. More complex loops can also result due to collision [30]. That TORA by itself was not designed to address collisions is the reason that mandates a lower layer for this purpose. While TORA itself requires low overhead for control packets, when considered together with the lower layer (for sending/processing acknowledgments to/from every neighbor) TORA becomes far less appealing.

7.2 Collision Resistant TORA

Like the proactive version of TORA, CR-TORA is intended for application scenarios where all mobile nodes require to send data packet to a single (possibly mobile) destination. Similar to proactive TORA, CR-TORA employs CLR, UPD and OPT packets. CLR packets indicate NULL height of the sender; UPD and OPT packets indicate a non-NULL height.

The primary difference between CR-TORA and TORA is that CR-TORA, as the name implies, has some in-built features to handle collisions, and thereby eliminates the need for a lower layer like IMEP to address collisions. In doing so CR-TORA also lowers the control packet overhead and the settling time during network partitions, and eliminates the need for time-synchronization.

Some of the other salient differences are as follows:

i) CR-TORA does not use reference levels; the height of a node i is a single value δ_i which is typically the number of hops from the destination.

ii) Two neighbors i and j at the same height (or $\delta_i = \delta_j$) do *not* consider each other as upstream/downstream based on their identities. In CR-TORA a node i is downstream of j only if $\delta_i < \delta_j$.

Like TORA, data flows only downwards in CR-TORA.

7.2.1 CR-TORA: Principle of Operation

Recall that in TORA a node responds to link-failures or link-reversals by *generating* an RL, or *propagating* an RL, or *reflecting* an RL, or *clearing* an RL. In CR-TORA nodes respond by *creating* CLR, or *propagating* CLR, or *creating* UPD, or *propagating* UPD, or *retransmitting* CLR.

The basic principle of operation of CR-TORA is as follows. A node losing its last downstream link *creates* a CLR packet. If the receipt of a CLR packet leaves a node X with no downstream neighbor, X *propagates* the CLR right away. However, if X has other downstream neighbors it starts a timer which runs for a duration T_W. While the

timer is running X may continue to receive other control packets. If (during this time) other control packets cause X to lose its last downstream neighbor, X propagates a CLR right away. On the other hand, if X retains one or more downstream neighbors after the timer expires, X *creates* a UPD. Neighbors of X with NULL heights then *propagate* the UPD. Most often CLR packets are created when a node loses its last downstream neighbor. However there are other scenarios (which could result due to collisions) which would also mandate creation of a CLR packet, or retransmission of a CLR packet.

To facilitate comparison of TORA and CR-TORA we shall once again consider the same example subnet considered for TORA in Section 7.1. At this point we shall ignore collisions (a more detailed explanation follows in Section 7.2.2). After the OPT from the destination ϕ is propagated the heights of the nodes will be $(0, \phi)$, $(1, A)$, $(2, B)$, $(2, C)$, $(3, E)$, $(3, G)$, and $(3, F)$, and subnet is depicted in Figure 7.2.1. Note that In CR-TORA links between nodes that have same height are undirected (unlike TORA).

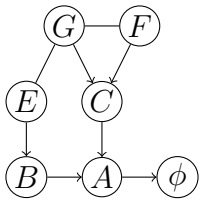

Figure 7.2

Subnet Running CR-TORA

Assume that as earlier, the link $C \rightarrow A$ goes down. The typical sequence of events in CR-TORA will be as follows:

a) $C \rightarrow$ CLR (C *creates* CLR);

b) $F \rightarrow$ CLR; $G \rightarrow$ CLR; (F and G *propagate* CLR); E waits for a time T_W ;

c) (No clear from B) $E \rightarrow$ UPD, $\delta_E = 3$ (E *creates* a UPD);

d) $G \rightarrow$ UPD, $\delta_G = 4$ (G *propagates* UPD);

e) $F \rightarrow$ UPD, $\delta_F = 5$; $C \rightarrow$ UPD, $\delta_C = 5$ (F and C *propagate* UPD).

In a scenario where A loses its link to destination ϕ (causing a subnet partition) the sequence of events that transpire in in CR-TORA will be:

a) $A \rightarrow$ CLR (A creates CLR);

b) $B \rightarrow$ CLR; $C \rightarrow$ CLR (B and C propagate CLR);

c) $E \rightarrow$ CLR; $F \rightarrow$ CLR; $G \rightarrow$ CLR (E, F and G propagate CLR).

In general, TORA generates lower number of control packets compared to CR-TORA when there is no network partition. However, this does not necessarily mean that CR-TORA has a higher overhead under such scenarios as we have ignored the overhead for IMEP in TORA. Even if IMEP overhead is ignored, TORA still generates a substantially higher number of control packets compared to CR-TORA when partitioning occurs in the subnet.

7.2.2 The CR-TORA Protocol

In the rest of this section we shall take a more in-depth look at the CR-TORA protocol.

7.2.2.1 CLR Event Identifiers (CEI):

In CR-TORA a CLR is created in response to a specific event - like a broken link. We shall see soon that there are two other specific events that lead to creation of a CLR. A node creating a CLR assigns a unique CLR event identifier (CEI) to the event.

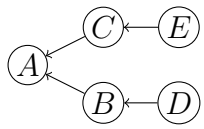

Figure 7.3

Sample Network Running CR-TORA

For example, if the link $B \rightarrow A$ goes down, B will set its height to NULL, and create a CLR with CEI α. Node D, which on receipt of the CLR-α, has lost its only downstream neighbor sets its height to NULL and simply *propagates* CLR-α. On the other hand, if the node A goes down, two different CEIs will be created: one by B, say CLR-α, due to the loss of the link $B \rightarrow A$, and one by C, say CLR-β, due to the loss of the link $C \rightarrow A$.

Unlike UPD reference heights in TORA, CR-TORA CEIs are *not* tied to time - or time synchronization is not necessary in CR-TORA. The only requirement is that no two CEIs should be the same. A simple strategy to accomplish this is choose the CEI by concatenating the identity of the creator with a sequence number maintained by the creator. In this case a CLR created by a node C will have a CEI $C \parallel q_c$. The next CLR created by C will have a CEI $C \parallel (q_c + 1)$. In the rest of this paper we shall simply employ a lower case Greek letters to represent CEIs.

7.2.2.2 CLR-List and UPD-List:

Every node maintains a CLR-list and UPD-list, both being a list of CEIs. Both lists are emptied from time-to-time, under different circumstances. The CLR-list $\mathcal{C}_i$ of node i is a list of CEIs made known to i, through CLR packets received by i. Both CLR and UPD packets broadcast by a node i will include its CLR-list[1] $\mathcal{C}_i$:

i) a CLR from i is of the form $[i, CLR, \mathcal{C}_i]$;

ii) a UPD from i is of the form $[i, UPD, \delta_i, \mathcal{C}_i]$.

When a node i with CLR-list $\mathcal{C}_i$ receives a packet $[j, CLR, \mathcal{C}_j]$, it adds all CEIs in $\mathcal{C}_j$ that were not already in its CLR-list $\mathcal{C}_i$ to its CLR-list. Thus, after the CLR is received $\mathcal{C}_i = \mathcal{C}_i \cup \mathcal{C}_j$.

A UPD is *created* by node i only after its T_W-timer fires. The timer is started when i receives a CLR, and if i still has at least one downstream neighbor. If i has at least one downstream neighbor left even after the timer fires, i creates a UPD which includes $\mathcal{C}_i$. As soon as the UPD is sent, i i) creates a UPD-list $\mathcal{U}_i = \mathcal{C}_i$; and ii) empties CLR-list $\mathcal{C}_i$.

A node *propagating* a UPD merely empties its CLR-list - it does *not* create a UPD-list. A node will propagate a UPD only if it had a NULL height before it received the UPD. A node i at a NULL height receiving a UPD $[j, UPD, \delta_j, \mathcal{C}_j]$ verifies if $\mathcal{C}_i \subset \mathcal{C}_j$ (in other words, i checks if all CEIs known to i are "addressed" by the UPD by j). Only if $\mathcal{C}_i \subset \mathcal{C}_j$, i sets its height to $\delta_i = \delta_j + 1$, propagates UPD $[i, UPD, \delta_i, \mathcal{C}_i]$, and empties its CLR-list $\mathcal{C}_i$.

[1] In our simulations the average number of CEIs that accompany a CLR or a UPD packet was found to be less than 2; thus the size of UPD and CLR packets in both TORA and CR-TORA are comparable.

On the other hand, if $\mathcal{C}_i \not\subset \mathcal{C}_j$ node i *retransmit* CLR $[i, CLR, \mathcal{C}_i]$.

7.2.2.3 Other CLR Creation Scenarios:

When a node i with a non-NULL height receives a UPD $[j, UPD, \delta_j, \mathcal{C}_j]$. Assume that j was i's last downstream neighbor, *and* the height δ_j announced by j is such that $\delta_j \geq \delta_i$ (or this UPD causes i to lose its last downstream neighbor); Now i *creates* a CLR (with a new CEI). Such a scenario arises when the CLR sent by j is lost due to collision, and i creates an UPD assuming j as its downstream neighbor; which (UPD generated by i) is then propagated by j.

When a node i with a non-NULL height receives a CLR $[j, CLR, \mathcal{C}_j]$, which leads to the loss its last downstream neighbor, i checks if any of the CEIs in $\mathcal{C}_j$ are also present in i's UPD-list $\mathcal{U}_i$. If not (or $\mathcal{C}_j \cap \mathcal{U}_i = \emptyset$) i propagates CLR $\mathcal{C}_i$. On the other hand, if $\mathcal{C}_j \cap \mathcal{U}_i \neq \emptyset$, then i *creates* a new CEI. The new CEI is added to i's CLR-list $\mathcal{C}_i$ before i broadcasts CLR $\mathcal{C}_i$. When a node creates a new CEI, or when it propagates a CLR, its UPD-list is emptied.

Thus, while a UPD is created by a node i only under one condition (after the T_W timer fires, if i retains a downstream neighbor), a CLR with a new CEI is created by a node i under three conditions:

i) i looses its last downstream link;

ii) i, which had a downstream neighbor j, receives a CLR which renders it with no downstream neighbor, *and* at least one of the CEIs that accompany the CLR packet by j is included in the UPD-list of i. Such a scenario can occur if i had prematurely created a

UPD assuming that j is still downstream (the CLR sent by j earlier may have been delayed or lost due to collision).

iii) When i receives a UPD packet from its last "downstream" neighbor j, node i finds that $\delta_j \geq \delta_i$ (or i's belief that j was downstream is recognized to be wrong).

7.2.3 The CR-TORA Algorithm

In CR-TORA the state of a node i is defined by i) latest OPT sequence number ϕ_q; ii) its height - δ_i; iii) a neighbor-table with list of neighbors and their height (we shall represent the height of a neighbor j as $H(j)$); iv) a CLR-list $\mathcal{C}_i$, and v) a UPD-list $\mathcal{U}_i$.

In CR-TORA every node reacts to the following five *triggers*: i) loss of last downstream link; ii) a CLR is received; iii) a UPD is received; iv) timer fires; and v) a OPT is received. The rules that govern CR-TORA under each trigger is summarized by the pseudo code as shown in Figure 7.4. In the following, DN represents "has downstream neighbor" (and !DN represents "no downstream neighbor"). We shall use the notation δ_{i_-} to represent the height of a node i *before* it received a control packet. After the packet is processed, the height of the node i may change, and is represented as δ_{i_+}.

In CR-TORA an OPT packet relayed by a node i is of the form $[i, OPT, \delta_i, \phi_q]$, where ϕ_q is the OPT sequence number chosen by the destination. A node i receiving an OPT from a neighbor j, $[j, OPT, \delta_j, \phi_q]$, with a fresh sequence number, node i i) sets its height to $\delta_i = \delta_j + 1$ and retransmits the OPT; and ii) empties its CLR-list and UPD-list. As mentioned earlier, the CLR-list is emptied when a node creates or propagates an UPD, and

1. Node i looses last downstream link:

```
   CREATE CLR
```

2. Node i receives $[j, CLR, \mathcal{C}_j]$:

```
IF (δ_i_ == NULL) return;
IF ((DN)&(timer-not-running))
   start-timer;
IF (!DN)
   IF (C_j ∩ U_i == ∅) PROPOGATE CLR
   ELSE CREATE CLR
```

3. Timer fires:

```
IF (DN) CREATE UPD
```

4. i receives $[j, UPD, \delta_j, \mathcal{C}_j]$:

```
IF (δ_i_ = NULL)
   IF (C_i ⊂ C_j) PROPOGATE UPD
   ELSE RETX CLR
ELSE IF (!DN)
   CREATE CLR
```

5: i receives OPT $[j, OPT, \delta_j, \phi_q{}']$:

```
IF (φ_q' > φ_q)
   φ_q = φ_q'
   H(j) = δ_j
   δ_i = δ_j + 1
   flush C_i, U_i
   SEND [i, OPT, δ_i, φ_q]
IF (φ_q' = φ_q)
   H(j) = δ_j
```

Figure 7.4

CR-TORA Algorithm

the UPD list is emptied either when it creates or propagates a CLR. The CLR-list and the UPD-lists of a node i are also flushed clear when a i creates a new CLR.

7.3 Formal Proof for Loop-Free Property of CR-TORA

The main idea of CR-TORA is influenced by the half-reversal technique proposed in [75]. After the initial OPT exchange, the entire network can be seen as a directed acyclic graph (DAG), where every node points to its downstream neighbors. Hence the entire topology of the network can be viewed as a DAG.

Route maintenance in CR-TORA is carried out using CLR and UPD packets. When a node X announces a CLR its height is set to $NULL$, and hence has all the links that originate or point to X can be removed from the DAG. Removing links from a DAG would still retain its acyclic property. Additionally, X will only accepts a UPD that answers all the CEIs included in the CLR packet sent by X. In other words X would accept a UPD created by A, only when this UPD is generated in response to the CEIs sent by X. This shows that the height announced by A is not dependent on that of X, as A created this UPD after receiving the CLR from X.

If the UPD includes all the CEIs sent by X, it (node X) would accept the UPD, and updates it height based on the height advertised by A. Now a link from X to A is added to the DAG, and this still does not violate the acyclic property, since X previously had no links in the DAG, and that the links that originate from A are not dependent on X. Hence the network topology still resembles a DAG even after executing the route maintenance phase.

Below we present a formal proof that emphasizes the loop-freedom in CR-TORA.

Theorem 7.3.1. The network that runs CR-TORA is loop-free.

Proof based on Contradiction: Let there be a set of m nodes $(N_1, N_2, \cdots, N_m)$ that form a loop. Without loss of generality lets assume that the downstream neighbor of N_1 is N_2 (represented as $downNeigh(N_1) = N_2$), $downNeigh(N_2) = N_3$, and that $downNeigh(N_m) = N_1$.

Further it can be seen that the sequence numbers stored at $(N_1, N_2, \cdots, N_m)$ are identical, as a node would update the internally stored sequence number when it receives a greater one.

Now since $downNeigh(N_1) = N_2$, the height of N_1 (represented as $Height(N_1)$) is

$$Height(N_1) = Height(N_2) + 1$$

and that of N_2 is

$$Height(N_2) = Height(N_3) + 1$$

and hence

$$Height(N_1) = Height(N_3) + 2$$

Expanding this equation we get

$$Height(N_1) = Height(N_m) + (m - 1) \tag{7.1}$$

Further since $downNeigh(N_m) = N_1$

$$Height(N_m) = Height(N_1) + 1 \tag{7.2}$$

The equations 7.1 and 7.2, can only be satisfied when $m = 0$. This means that the size of the set of nodes that form a loop in CR-TORA is 0, and hence CR-TORA is loop-free. □

7.4 Simulations

Simulations were carried out to evaluate the performance of TORA and CR-TORA using random realizations of subnet topologies with mobile nodes.

7.4.1 Simulation Environment

The simulating environment generates $N = 150$ randomly placed nodes in square region with edges of size 500 meters. The range of each node was assumed to be R meters (simulations were performed for $R = 55, 60, 65$). The destination is randomly chosen. Every node periodically attempts to send a data packet to the destination - once every T_D seconds on an average (simulations were performed for $T_D = 1, 2, 3$ seconds). However, a node sends a data packet only if it has a non-NULL height. All simulation runs were performed for a network time of 100 seconds.

Mobility: To model mobility, at random instances of time some nodes were moved by random distances in X and Y directions (distance uniformly distributed between ± 15 meters); some nodes were turned off; some of the nodes that are currently off were turned on and relocated at a random position. We simulated two mobility models. In the model M-I with lower mobility, on an average, during every second, i) $M_m = 7.5\%$ of the total number of nodes were moved; ii) $M_o = 3.75$ % of the nodes were turned off; and 50 % of nodes that are currently off are turned on. For the higher mobility model (Model M-II)

the parameters were $M_m = 15\%$ and $M_o = 7.5\%$. In applying the mobility model, the only difference between the destination and the other nodes is that the destination is never turned off.

MAC Layer: The channel bit rate was assumed to be 2Mb/s. Nodes employ p-persistent CSMA with $p \approx 1/20$. The carrier sense delay was assumed to be $\tau_{cs} = 1\mu sec$ sec. In other words, two nodes within the range of each other may not sense each other's transmission if they begin their transmissions within $\tau_{cs} = 1\mu sec$ of each other[2]. If a node senses that the channel becomes available at a time t, it begins its transmission at a time $t + x\tau_{cs}$ where x is random, and uniformly distributed between 1 and 20. A packet is received successfully by a node (without collision) only if not more than one of the receiver's neighbors (the sender of the packet) was transmitting during the entire duration of the packet. Most collisions at a node occur due to the "hidden station problem" - overlapping transmissions from neighbors of a node who are not within each other's range.

Control, Data and HELLO Packets: For both TORA and CR-TORA the destination sends OPT packets once every $T_{opt} = 5$ seconds. The duration of control packets were assumed to be 0.25 $msec$ (about 64 bytes). The HELLO packets were $0.0625msec$ long (about 16 bytes). IMEP ACK packets (only for TORA) were also assumed to be of $0.0625msec$ duration. For both protocols we assumed a random processing delay in each node, uniformly distributed between 1 and $5msec$.

[2] $1\mu sec$ is a conservative estimate given that the propagation delay for the maximum distance of 65 m is less than $0.25\mu sec$.

In TORA IMEP attempts to encapsulate multiple control/ACK packets into one IMEP packet. To offer a collision-free environment for TORA, IMEP sends ACKs for every control packet received. If a node has not heard an ACK from one or more of its neighbors within a time $T_{ack} = 15$ ms, it retransmits the control packet and explicitly indicates the identities of nodes which had not acknowledged the previous transmission. Only such nodes will need to send an ACK for the retransmitted packet. We limited the number of retransmissions to 2.

Data packets were of duration 1 $msec$ (about 256 bytes). The maximum duration of IMEP packets was set at $1.0625msec$ (272 bytes) to permit an ACK for a data packet to be sent along with the data packet. Only CSMA (no RTS/CTS handshake is used) was used even for data packet transmissions due to the relatively small size of packets.

In both TORA and CR-TORA a node A with multiple downstream neighbors chooses the one with the least height as the next hop to forward a data packet. If the data packet transmitted by A to a neighbor B is deemed unsuccessful, then A sends the data packet to its next downstream neighbor (if available). In both protocols a node A sending a data packet to a neighbor B attempts to overhear the retransmission of the data packet within a duration T_{ack}, failing which the data packet is retransmitted. The number of retransmissions are limited to 2 before the data transmission is deemed unsuccessful.

For maintaining dynamic list of neighbors every node tries to break silence once every T_s seconds. If a node A has not had the need to transmit a control or data packet (or ACK in TORA) in the last T_s seconds, A sends a HELLO packet to notify its presence to its neighbors. If a node A has not heard a transmission from a neighbor B for more than $2T_s$

seconds, the neighbor B is removed from A's neighbor table (if B happened to be A's last downstream link, link-failure route maintenance activities are triggered).

7.4.2 Results

Many simulation runs were performed; each run was for a network time of 100 seconds. The simulations were instantiated by the destination, by sending an OPT packet. Some of the parameters that were measured by the simulations were

a) $N_{tot} = N_{ctrl} + N_{opt} + N_{ack} + N_{dat}$: total number of packets: which is the sum total numbers of control packets, OPT packets, ACK packets, and data packets.

b) N_{tx}: total number of transmissions; for CR-TORA $N_{tot} = N_{tx}$; for TORA $N_{tx} < N_{tot}$ as packets queued for transmission can be aggregated by the IMEP layer.

c) n_{dat}: number of data packets instantiated by all nodes; note that $n_{dat} << N_{dat}$ as each of the unique n_{dat} data packets will need multiple transmissions/retransmissions over multiple hops.

d) n_{suc}: Total number of data packets reaching the destination;

e) t_{lat}: average latency for data packets.

f) n_{ev}: Total number of last-downstream-link-loss events

Table 7.1 gives a detailed comparison of TORA and CR-TORA for one specific choice of parameters: viz, $T_{opt} = 5$, $R = 60$, and $T_D = 1$ second for two mobility models M-I and M-II.

Ideally, in the span of 100 seconds, each node should have created one data packet - or $100N = 15000$ data packets should have been created. However, as data packets

Table 7.1

TORA (T) vs CR-TORA (CR-T) for two mobility models M-I and M-II.

		N_{tot}/N_{tx}	n_{dat}	n_{suc}	N_{dat}	t_{lat}	n_{ev}
M-I	T	624,869/446,861	13,615	7071	312,374	0.09	198
M-I	CR-T	231,169	13,229	8742	182,296	0.05	259
M-II	T	866,082/632,243	13,261	6232	459,083	0.11	397
M-II	CR-T	254,318	12,917	7621	195,461	0.052	512

are sent by a node only when they have a non-NULL height, the actual number of created data packets is less than 15,000. Ultimately the intent is to increase the number of packets received by the destination, while lowering the cost. In the TORA subnet for the mobility model M-I, the destination receives 7071 packets, compared to 8742 in the CR-TORA subnet. One simple measure of the "cost" is the total number of transmissions (by all nodes together). In the TORA subnet 624,869 TORA packets are aggregated into 446,861 IMEP transmissions compared to 231,169 packets (and the same number of transmissions as there is no aggregating lower layer) in CR-TORA. Furthermore, the average latency in TORA is 0.09 seconds while it is only 0.05 seconds in CR-TORA. Thus, CR-TORA outperforms TORA in every conceivable respect. CR-TORA results in a 15% higher throughput with a 40% reduction in latency, for 40% of the cost.

Data Packets: It is interesting to note that slightly more data packets were *created* in the TORA subnet ($n_{dat} = 13615$) compared to $n_{dat} = 13229$ for CR-TORA. The reason for this is TORA's long settling time during subnet partitions, during which nodes possess a non-NULL height while they do not actually have a physical path. This is also one reason for the substantially higher number of data packet transmissions N_{dat} in TORA compared to CR-TORA (about 1.7 times higher).

While both TORA and CR-TORA do not strive to determine the shortest path, the average path length for TORA is 9.9 compared to 7.1 for CR-TORA. This is in part due to the fact that a node does not consider a same height neighbor as downstream. Typically, CR-TORA heights reflect the actual number of hops between the node and the destination. Lower path lengths obviously lead to in lower latency. and lower data traffic. Furthermore the substantially higher N_{dat} in TORA also results in more packets being queued, leading to increased latency.

Route Maintenance Overhead: In both TORA and CR-TORA a series of route maintenance steps are triggered by an event where a node loses its last downstream neighbor. The number of such events n_{ev} were also measured during the simulations. While both TORA and CR-TORA were simulated for the exact same network topology (and mobility) the value n_{ev} was lower in TORA (198) compared to CR-TORA (259). As CR-TORA does not consider a neighbor of the same height as downstream the loss of the last downstream neighbor occurs more often in CR-TORA. However, even while more maintenance activities are triggered, CR-TORA mandates lower number of maintenance packets. Apart from data packets, the TORA subnet invoked $624869 - 312374 = 312,495$ "other" packets (HELLO, OPT, control packets like UPD and CLR, and IMEP ACK). In CR-TORA the number of "other packets" is substantially lower at $48,873$.

Effect of Mobility: As can be seen from Table 1 CR-TORA out-performs TORA by similar margins even for the scenario with increased mobility (M-II instead of M-I). While increased mobility results in increased overhead and lower throughput in both TORA and CR-TORA, the increase in TORA overhead ($866082 - 624869 = 241,213$) is 10 times

greater than that of CR-TORA ($254318 - 231,169 = 23,149$). This significant increase in traffic also affects the latency t_{lat} of TORA which is increased by 0.02sec (and only by $0.002sec$ in CR-TORA).

Waiting Time T_W: One new parameter introduced by CR-TORA is the waiting time T_W. Recall that if a node receiving a CLR packet has other downstream nodes it waits for a duration T_W before it sends a UPD. We experimented with various waiting times ranging from 10 to 20 $msec$. Our simulations show that a waiting time of 15 $msec$ maximized throughput (fraction of data packets that reached the destination). However the performance of CR-TORA is not very sensitive to this parameter. Between 10 to 20 $msec$ the worst and best case scenarios varied only by less than 5%.

7.4.3 Effect of Network Density

In our simulations network density is controlled by adjusting the value of the range R. For the three of choices of $R = 55, 60, 65$ meters, the total number of nodes connected to the destination (averaged over many random realizations) are 94, 128 and 141 (63%, 85%, and 94%) respectively. As can be seen from Figure 7.5(a) CR-TORA offers greater throughput in all three cases.

Figure 7.5(b) depicts the maintenance traffic generated for different network densities. An increase in network density (and hence connectivity) will lead to lower number of maintenance events n_{ev} leading to lower maintenance traffic. However increased network density can also lead to higher probability of collisions, and consequently an increase in traffic due to retransmissions. The total number of control packets N_{ctrl} reduced with

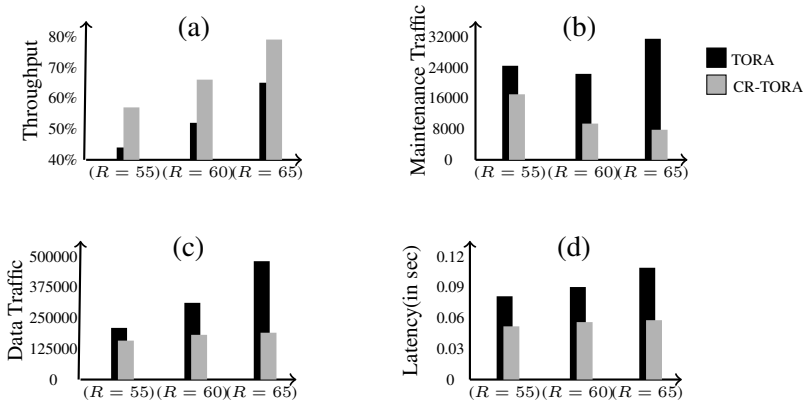

Figure 7.5

Comparison of TORA and CR-TORA for different network densities

increased network density for CR-TORA; on the other hand, in TORA N_{ctrl} increases for high network densities due to IMEP retransmissions due to collisions.

Increased connectivity resulted in an increase in the total number of data packets created (n_{dat}) in both TORA and CR-TORA, and consequently led to greater data traffic (N_{dat}). As can be seen from Figure 7.5(c) the increase in N_{dat} with network density is substantially higher for TORA, due to the higher number of collisions.

If we ignore collisions, one would expect lower latency with higher R due to a reduction in the number of hops. However increase in collisions with network density will create more retransmissions and thus increase the latency. As can be seen in Figure 7.5(d), in CR-TORA the two opposing effects almost balance out each other, causing only a very small increase in latency with increasing network density. On the other hand, due to the substantially higher number of collisions in TORA, the latency in TORA increases substantially with network density.

Recall that the TORA subnet generates more "useless" data packets (which will ultimately be undeliverable) as nodes possess invalid non-NULL heights for long durations under scenarios involving network partitions. We expect this phenomenon to be more evident for low density networks where more network partitions occur. As expected, for $R = 55$, the total number of data packets originated by the TORA and CR-TORA subnet are $n_{dat} = 13282$ and $n_{dat} = 11588$ respectively. The total number of data packets delivered are $n_{suc} = 5835$ for TORA and $n_{suc} = 6612$ for CR-TORA.

The facts that i) the goal of TORA is an important one for many application scenarios and that ii) there are compelling reasons to eliminate the need for an expensive lower layer were the motivations for the CR-TORA protocol. CR-TORA has in-built features that address collisions, and thereby eliminates the need for IMEP. Apart from eliminating the need for a lower layer the other beneficial properties of CR-TORA are i) lowering settling time during network partitions (leading to less instances of "useless" data traffic); ii) lower latency (primarily attributable to the fact that same height neighbors are not considered as downstream) due to shorter path lengths; and iii) lack of the need for time-synchronization.

Simulations show that, compared to TORA, CR-TORA reduces the number of total number of transmissions by 60%; results in a 15% increase in throughput; and a 40% reduction in latency. CR-TORA is thus a promising protocol for many practical MANET/sensor networks.

CHAPTER 8

SECURING CR-TORA

In this chapter we propose to use TMMs to secure CR-TORA. CR-TORA has the same design goals as TORA, which is to provide connectivity to a limited set of destinations. Like TORA it provides multiple paths to each available destination. This is achieved by storing the heights reported by each neighbor for each destination. Hence apart from its own height, a node is also required to store the heights of its neighbors.

Further in CR-TORA, for each destination, a node has to maintain a few separate fields like: i) the CLR and UPD lists, ii) a field to hold the waiting period, iii) current count of available downstream neighbors.

The TMMs designed to secure CR-TORA must address the above mentioned characteristics.

8.1 IOMT Design

For securing CR-TORA a node X requires to maintain two IOMTs: i) main IOMT that stores the height of X for each available destination, and ii) an auxiliary IOMT that stores the heights reported by X's neighbors for each destination.

8.1.1 Main IOMT

This tree is used to store the heights of a node for each available destination. Apart from the height metric m, CR-TORA also requires a node X to store the following for each destination:

- CLR-list
- UPD-list
- waiting period
- number of available downstream neighbors

Hence a destination record (DR) stored for a destination in CR-TORA is represented as:

$$\mathbf{D_c} = [\mathbf{D} \parallel d \parallel u \parallel T] \tag{8.1}$$

where $\mathbf{D} = [id \parallel q \parallel m \parallel a \parallel \tau]$. In the above equation:

- a and u are the cumulative hash of the current CLR and UPD lists stored for destination id respectively. The cumulative hashes are computed as explained in Section 6.1.2
- d is the number of available downstream neighbors
- T is used to hold the current waiting period

Even though X has to store additional information for each destination, a DR broadcast for destination id should only include the current CLR-list of id. Hence while broadcasting a packet about id it is sufficient to include the values stored in $\mathbf{D} = [id \parallel q \parallel m \parallel a \parallel \tau]$. Therefore while $\mathbf{D}_c$ represents the stored DR, $\mathbf{D}$ represents the DR broadcast by a node.

Every leaf of the main IOMT is used to store information about a unique destination, and is of the form:

$$\mathbf{L} = (id \parallel \theta_{id} \parallel id') \tag{8.2}$$

where id is the identity of the destination this leaf represents, and id' is the next available destination. The value $\theta_{id} = h(\mathbf{D}_c)$ is hash of the DR stored for destination id.

In CR-TORA the height of X for a particular destination (say id) is based on the heights reported by several neighbors of X. This deviates from protocols like AODV and DSR, where the height of a node is based on the DR received from a single neighbor (stored as the provider of the DR (id_p)). Hence in CR-TORA, the value of θ does not include any provider's identity. However, X stores all the reported neighbors heights in a separate auxiliary tree.

8.1.2 Auxiliary IOMT

This tree is used to store the reported heights of the neighbors for each destination. Every leaf of the auxiliary tree represents a unique tuple (destination identity (say id), neighbor identity (say id_n)). The information stored for each tuple is:

$$\mathbf{d} = [id \parallel id_n \parallel m] \tag{8.3}$$

where m is the height advertised by id_n.

The leaf of the auxiliary tree is of the form:

$$\mathbf{L} = (i \parallel \theta_i \parallel i') \tag{8.4}$$

where $i = h(id \parallel id_n)$, i' is the next available tuple, and $\theta_i = h(\mathbf{d})$.

8.2 TMM Data Structures

Apart from the parameters defined in Section 3.4.5, the TMM also holds two values: i) r: root of main IOMT, and ii) r_a: root of auxiliary IOMT. The value Δ_W, mentioned in Section 3.4.5, defines a constant period by which a node has to wait before creating an UPD packet.

The self-MAC μ_s is bound to the root of the main IOMT r, and is valid until r is unchanged. Additionally we introduce a new self-MAC μ_{set} which is computed as:

$$\mu_{set} = h(x \parallel y \parallel z \parallel Opt \parallel S_X) \tag{8.5}$$

where x, y and z are cumulative hashes of three different sets (say $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ respectively), and opt is a flag that specifies the operations performed on sets $\mathbf{S}_1$ and $\mathbf{S}_2$. The various operations supported are: i) Union, ii) Intersection, and iii) Set Minus. A self-MAC μ_{set} indicates that when an operation Opt is performed between the sets represented by x and y, the result is a set represented by z.

The cumulative hash of the elements represented in the set can be found using the function $extend()$ (as shown in Figure 6.4.5). For example let set $\mathcal{S} = (c1, c2, c3, c4, c5)$. Now the cumulative hash of the set $\mathcal{S}$ can be found by making the function call $extend(c1, 4, (c2, c3, c4, c5))$.

The value of the flag INV which is used for computing the value of h_r while sending a DR, is assumes to be 0 in this chapter. Finally, the TMM has an internal function $rand()$ that returns unique CEIs.

8.3 TMM Functions

Apart from the functions UpdateNeighborTable() and AddDeleteLeaf(), the proposed TMM design exposes the following functions:

1. SetOps(),
2. SendCRT(),
3. LossOfLink(),
4. UpdateUPD() and
5. UpdateCLR()

The function SetOps() is used to carry out set operations, and is responsible for issuing the self-MAC μ_{set}. The function SendCRT() is used to either create a DR about the node, or send stored DRs. Stale information is removed from the tree using the function LossOfLink(). A stored DR for a destination is updated based on the received UPD/OPT and CLR packets using the UpdateUPD() and UpdateCLR() functions respectively.

8.3.1 SetOps$(n, \{x_1 \cdots x_n\}, m, \{y_1 \cdots y_m\}, Opt)$

This function is used to execute set operations, and the algorithm followed is shown in Figure 8.3.1. The inputs taken by this function are:

1. a set $\{x_1 \cdots x_n\}$ of size n
2. a set $\{y_1 \cdots y_m\}$ of size m
3. Opt, that mentions the operation that need to be performed on the input sets

The function performs the specified operation, and computes the cumulative hash z of the resulting set. Finally it issues a self-MAC μ_{set}.

SetOps$(n, \{x_1 \cdots x_n\}, m, \{y_1 \cdots y_m\}, Opt)$ {
IF $Opt == UNION$ //Union
$\{z_1 \cdots z_r\} = \{x_1 \cdots x_n\} \cup \{y_1 \cdots y_m\}$;
ELSE IF $Opt == INT$
$\{z_1 \cdots z_r\} = \{x_1 \cdots x_n\} \cap \{y_1 \cdots y_m\}$;
ELSE IF $Opt == SETM$
$\{z_1 \cdots z_r\} = \{x_1 \cdots x_n\} \setminus \{y_1 \cdots y_m\}$;
$x = extend(x_1, n-1, \{x_2 \cdots x_n\})$;
$y = extend(y_1, m-1, \{y_2 \cdots y_m\})$;
$z = extend(z_1, r-1, \{z_2 \cdots z_r\})$;
RETURN $\mu_{set} = h(x \parallel y \parallel z \parallel Opt \parallel S_X)$;
}

Figure 8.1

Function SetOps()

8.3.2 LossOfLink$(\mathbf{d}, \mathbf{L}_j, \mathbf{v}_j, \mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i)$

This function is used by node X to maintain stale information, and the algorithm followed by this function can be seen in Figure 8.3.2. It takes the following inputs:

1. stored neighbor height $\mathbf{d}$
2. auxiliary leaf $\mathbf{L}_j$ that represents $\mathbf{d}$, and the set of instructions $\mathbf{v}_j$ that map it to the root r_a
3. stored DR $\mathbf{D_c}$
4. leaf $\mathbf{L}_i$ that represents $\mathbf{D_c}$, and the set of instructions $\mathbf{v}_i$ that map it to the root r

The TMM X first maps the two leaves to their corresponding roots, and checks whether they indeed represent the provided information.

When the stored DR is expired ($\tau \leq t_x$) the TMM will un-initialize the leaf $\mathbf{L}_i$, and thereby erasing the information stored in the expired DR.

The TMM will un-initialize the auxiliary leaf, if the corresponding leaf for the same destination has its $\theta_i == 0$.

Additionally the leaf $\mathbf{L}_j$ is also un-initialized, when the neighbor represented in $\mathbf{d}$ is no longer a bidirectional neighbor. Further, if the lost neighbor was previously downstream, the value of d is decremented. Finally, if this results in the loss of the last downstream link ($d == 0$) the TMM initializes a to a newly created CEI by calling the function $rand()$. Lastly the leaf and the root r are updated to reflect these changes.

8.3.3 SendCRT($\mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i$)

This function is used by node X to send stored DRs, and also to create DR about X itself. The pseudo code can be seen in Figure 8.3.3. The function takes the following inputs:

1. stored DR $\mathbf{D_c}$
2. leaf $\mathbf{L}_i$ that represents $\mathbf{D_c}$, and the set of instructions $\mathbf{v}_i$ that map it to the root r

The TMM first checks if the function is invoked to create a DR about itself ($id == X$). If so, the TMM X increments the internal sequence number, create a DR about itself, and finally authenticate it to all the bidirectional neighbors of X.

However if the function is invoked to relay stored DRs, the TMM first maps the leaf to the root r, and later verifies if the leaf holds information about $\mathbf{D_c}$. If the stored DR has an infinite height the TMM authenticates it to all the bidirectional neighbors right away. Nevertheless, if $m < INF$ the TMM only sends the DR if the corresponding waiting period has expired ($T \leq t_x$). For this case, the TMM copies the value of u into a (copies the UPD-list into CLR-list). Finally TMM X updates the leaf and the root r subsequently.

LossOfLink($\mathbf{d}, \mathbf{L}_j, \mathbf{v}_j, \mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i$) {
//$\mathbf{D_c} = [id \parallel q \parallel \tau \parallel m \parallel a \parallel INV \parallel n \parallel u \parallel T]$;
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$;
//$\mathbf{d} = [id_o \parallel id_{po} \parallel m_o]$;
//$\mathbf{L}_j = [j \parallel \theta_j \parallel j']$;
IF $(r \neq mapleaf(\mathbf{L}_i, \mathbf{v}_i)) \vee (r_a \neq mapleaf(\mathbf{L}_j, \mathbf{v}_j))$ RETURN ERROR;
IF $((\theta_i \neq h(\mathbf{D}_c) \vee (\theta_j \neq h(\mathbf{d})))$ RETURN ERROR;
IF $(id \neq id_o)$ RETURN ERROR;
//house keeping without external input
IF $(\tau \leq t_x)$ //prepare main leaf for removal
    $\mathbf{L}_i = (i \parallel 0 \parallel i')$; $r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
    RETURN;
IF $(\theta_i == 0)$ //prepare aux leaf for removal
    $\mathbf{L}_j = (j \parallel 0 \parallel j')$; $r_a = mapleaf(\mathbf{L}_j, \mathbf{v}_j)$;
    RETURN;
//update without external input
IF $(checkbd(id_{po}) == 0)$ // only possible reason is loss of link to id_{po}
    $\theta_j = 0$; $r_a = mapleaf(\mathbf{L}_j, \mathbf{v}_j)$;
    IF $(m_o < m)$
        $d = d - 1$;
        IF $(d < 1)$
            $m = INF$;
            $T = 0$; $u = 0$;
            $a = rand()$; create new CEI
        $\theta_i = h(\mathbf{D_c})$; $r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
}

Figure 8.2

Function LossOfLink()

SendCRT($\mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i$) {
//$\mathbf{D_c} = [id \parallel q \parallel \tau \parallel m \parallel a \parallel INV \parallel n \parallel u \parallel T]$;
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$;
IF $(r \neq mapleaf(\mathbf{L}_i, \mathbf{v}_i)) \vee (id \neq i) \vee (h(\mathbf{D_c}) \neq \theta_i))$ RETURN ERROR;
$h_r = h(h(id \parallel q \parallel \tau \parallel m \parallel a) \parallel 0)$;
IF $(T < t_x)$
  RETURN $computemacs(h_r, X, 0)$;
  IF $(a \neq 0)$
    $u = a$; $a = 0$;
    $\theta_i = h(\mathbf{D}_c)$; $r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
}

Figure 8.3

Function SendCRT()

8.3.4 Macro INPUT-VALIDATE

For the sake of simplicity we divided the function that updates stored DRs into UpdateUPD() and UpdateCLR(). Function UpdateUPD() is invoked when a UPD packet is received, while UpdateCLR() is called when a CLR packet is received. However the inputs accepted, and the initial input validations performed by these two functions are identical. Hence we defined a macro INPUT-VALIDATEthat performs these identical operations, and it is loaded in both the function UpdateUPD() and UpdateCLR(). The operations carried out by INPUT-VALIDATEcan be seen in Figure 8.3.4.

The inputs accepted by both the function UpdateUPD() and UpdateCLR() are:

1. the received DR $\mathbf{D}_w$, and its provider id_{pw}
2. self-MAC μ_s
3. stored neighbor height $\mathbf{d}$
4. auxiliary leaf $\mathbf{L}_j$ that represents $\mathbf{d}$, and the set of instructions $\mathbf{v}_j$ that map it to the root r_a

5. stored DR $\mathbf{D_c}$
6. leaf $\mathbf{L}_i$ that represents $\mathbf{D_c}$, and the set of instructions $\mathbf{v}_i$ that map it to the root r
7. $\mathbf{S} = [x \parallel y \parallel z \parallel Opt \parallel \mu_{set}]$

The TMM first validates the self-MACs μ_{set} and μ_s. Later the TMM maps the two leaves to their corresponding roots, and also verifies if they indeed point to the information provided.

Further, if either of θ_i or θ_j are set to 0 the TMM initializes them to their default values. Finally before updating the received height of the neighbor, the TMM stores the currently stored height into a temporary variable. This is done to enable further checks on the status (either upstream or downstream) of the neighbor, before receiving the DR $\mathbf{D}_w$.

8.3.5 UpdateUPD$(\mathbf{D}_w, id_{pw}, \mu_s, \mathbf{d}, \mathbf{L}_j, \mathbf{v}_j, \mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i, \mathbf{S})$

This function is invoked by node X to update stored DRs when a DR with finite height ($m_w < INF$) is received. The pseudo code followed by the function can be seen in Figure 8.3.5.

TMM X first loads the macro INPUT-VALIDATEto perform input validations. Later the TMM performs two checks. First, the TMM checks if both the received (m_w) and stored (m) heights are finite. If so, the TMM increments d if $m_w < m$, and if the sender was previously not counted as downstream. Additionally, if $m_w \geq m$ and the provider was previously counted as downstream the value of d is decremented. Now if the updates d is equal to 0, it means that X lost its last downstream link due to link reversal. Hence X would clear its stored height, and create a new CEI as mentioned in Section 7.2.2.3.

INPUT-VALIDATE($\mathbf{D}_w, id_{pw}, \mu_s, \mathbf{d}, \mathbf{L}_j, \mathbf{v}_j, \mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i, \mathbf{S}$) {
//$\mathbf{D}_w = [id_w \parallel q_w \parallel m_w \parallel a_w \parallel \tau_w]$;
//$\mathbf{d} = [id_o \parallel id_{po} \parallel m_o]$;
//$\mathbf{D_c} = [id \parallel q \parallel m \parallel a \parallel \tau \parallel d \parallel u \parallel T]$;
//$\mathbf{L}_j = [j \parallel \theta_j \parallel j']$;
//$\mathbf{L}_i = [i \parallel \theta_i \parallel i']$;
//$\mathbf{S} = [x \parallel y \parallel z \parallel Opt \parallel \mu_{set}]$
//verify all inputs
IF $((\mu_{set} \neq 0) \wedge (\mu_{set} \neq h(x \parallel y \parallel z \parallel Opt \parallel S_X))$ RETURN ERROR;
IF $(\mu_s \neq (h(h(\mathbf{D}_w) \parallel 0) \parallel id_{pw} \parallel r \parallel S_X))$ RETURN ERROR;
IF $(r \neq mapleaf(\mathbf{L}_i, \mathbf{v}_i)) \vee (r_a \neq mapleaf(\mathbf{L}_j, \mathbf{v}_j))$ RETURN ERROR;
IF $((\theta_i \neq h(\mathbf{D}_c) \vee (\theta_j \neq h(\mathbf{d})))$ RETURN ERROR;
IF $(((id_w \neq id) \wedge (id_{pw} \neq id_{po})) \vee (id \neq id_o) \vee (id \neq i) \vee (q_w < q))$ RETURN ERROR;
IF $(\theta_j == 0)$
$id_o = id_w; id_{po} = id_{pw}; m_o = INF]$
IF $(\theta_i == 0)$
$id = id_w; q = 0; \tau = 0; m = INF; a = 0; INV = 0; d = 0; u = 0; T = 0;$
$m'_o = m_o; m_o = m_w;$
IF $(q_w > q)$
$q = q_w; a = 0; \tau = \tau_w; u = 0; T = 0; m = INF; d = 0;$
}

Figure 8.4

Macro INPUT-VALIDATE

The second check performed by the TMM is to verify if X received a finite height when the stored height for the destination is infinite. According to CR-TORA, X only accepts this UPD if all the CEIs stored in its CLR-list are included in the received packet. In other words, performing set minus between the stored and received CEIs should return NULL. The TMM uses the values included in the self-MAC μ_{set} to evaluate this condition. If satisfied, the received height is accepted by the TMM, else the stored height is unchanged.

Finally the TMM X updates the leaf and root subsequently.

UpdateUPD($\mathbf{D}_w, id_{pw}, \mu_s, \mathbf{d}, \mathbf{L}_j, \mathbf{v}_j, \mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i, \mathbf{S}$) {
INPUT-VALIDATE//load the macro
IF $((m_w < INF) \wedge (m < INF))$ //UPD
  IF $((m_w < m) \wedge (m'_o > m - 1))$
    $d = d + 1$;
  ELSE IF $((m_w \geq m) \wedge (m'_o < m))$
    $d = d - 1$;
    IF $(d < 1)$ //last DN lost due to link reversal
      $a = rand(); u = 0; T = 0; m = INF; d = 0$; $m_o = INF$; //create new clear
ELSE IF $((m_w < INF) \wedge (m == INF))$ //UPD
  IF $((a == 0) \vee ((\mu_{set}) \wedge (Opt = SETM) \wedge (x == a) \wedge (y == a_w) \wedge (z == 0)))$
    //accept UPD as all CEIs addressed
    $m = m_w + 1; d = 1$;
  ELSE
    $m_o = INF; T = 0$;
$\theta_j = h(\mathbf{d})$; $r_a = mapleaf(\mathbf{L}_j, \mathbf{v}_j)$;
$\theta_i = h(\mathbf{D_c}); r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
}

Figure 8.5

Function UpdateUPD()

8.3.6 UpdateCLR($\mathbf{D}_w, id_{pw}, \mu_s, \mathbf{d}, \mathbf{L}_j, \mathbf{v}_j, \mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i, \mathbf{S}$)

This function is used by node X to update the stored DRs based on received CLR packets ($m_w == INF$). The algorithm followed can be seen in Figure 8.3.6.

Like UpdateUPD() the function first loads INPUT-VALIDATEto perform input validations.

The function only executes when the stored height (m) is finite. When both the stored and received heights are infinite, its usually the case when a node is hearing back the CLR it sent, and no action is needed.

Hence when $m < INF$, the TMM first checks if the provider of the CLR was previously listed as downstream. If yes the value of d is decremented. Now if the updated $d > 0$, node X has a valid height even after processing the CLR. Hence the TMM sets the waiting time T (if its not already set), and adds the received CEIs to its existing CLR-list. The TMM uses the values included in $\mathbf{S}$ to carry out this operation.

However if d becomes 0, the TMM has lost its last downstream link, and would clear the stored height. Now the TMM first checks if the received CEIs intersect with the stored UPD-list. In such a case the TMM would create a new CEI as explained in Section 7.2.2.3. If not the TMM adds the received CEIs to its existing CLR-list.

8.4 Realizing CR-TORA

In this section we explain how the above mentioned functions can be used to realize CR-TORA protocol. Firstly the function UpdateNeighborTable() can be used to maintain connectivity and to create self-MAC μ_s. The function SendTS() is used to create periodic HELLO packets. The function AddDeleteLeaf() can be used to create place holders when

UpdateCLR($\mathbf{D}_w, id_{pw}, \mu_s, \mathbf{d}, \mathbf{L}_j, \mathbf{v}_j, \mathbf{D_c}, \mathbf{L}_i, \mathbf{v}_i, \mathbf{S}$) {
INPUT-VALIDATE// loading the macro
IF $((m_w == INF) \wedge (m < INF))$ //CLR
  IF $((m'_o < m) \wedge (d > 1))$
    $d = d - 1$;
    IF $(T \leq t_x)$ //timer has not been set
      $T = t_x + \Delta_W$; //start timer
    IF $((\mu_{set} == 0) \wedge (a == 0))$ $a = a_w$;
    ELSE IF $((\mu_{set}) \wedge (Opt = UNION) \wedge (x == a) \wedge (y == a_w))$ $a = z$;
    ELSE RETURN ERROR;
  ELSE IF $((m'_o < m) \wedge (d == 1))$ //last DN will be lost
    $m_o = 0$; $T = 0$; //ready to send CLR
    IF $(u \neq 0)$
      IF $((\mu_{set} == 0) \vee (Opt \neq INT) \vee (x \neq u) \vee (y \neq a_w))$ RETURN ERROR;
      IF $(z \neq 0)$
        $a = rand(); u = 0; T = 0$ //create new CEI
      ELSE IF $(z == 0)$
        $a = a_w; u = 0; T = 0$ //copy received CEI
    ELSE IF $((\mu_{set} == 0) \wedge (a == 0))$
      $a = a_w$;
    ELSE IF $(\mu_{set} \wedge (Opt = UNION) \wedge (x == a) \wedge (y == a_w))$
      $a = z$;
    ELSE RETURN ERROR;
$\theta_j = h(\mathbf{d}); r_a = mapleaf(\mathbf{L}_j, \mathbf{v}_j)$;
$\theta_i = h(\mathbf{D_c}); r = mapleaf(\mathbf{L}_i, \mathbf{v}_i)$;
}

Figure 8.6

Function UpdateCLR()

information about new identities arrive. For example when node X receives a DR about destination D from a neighbor A for the first time, it invokes AddDeleteLeaf() to create an uninitiated leaf in the main tree for identity D. Additionally, X calls AddDeleteLeaf() one more time to create an uninitiated leaf for $h(D \parallel A)$ in the auxiliary tree. These uninitiated leafs are later submitted to either UpdateUPD() or UpdateCLR() for further processing.

When destination D desires to create an OPT packet, it calls the function SendCRT() which returns a DR about D that includes its latest sequence number. Every node on receiving the OPT packet submits the received and saved DR for D to the function UpdateUPD().

Further maintenance of stale information is carried out by invoking the function LossOfLink(). Say X looses its connectivity to a neighbor A, through which it received a DR for D. Now X invokes its LossOfLink() to un-initialize the auxiliary leaf for $h(D \parallel A)$. Further the function also updates the downstream count d to reflect this loss. If d is updated to 0, the function LossOfLink() clears the heights stored for D. The updated DR can later be submitted to SendCRT(), which authenticates it to all the bidirectional neighbors of X.

The functions UpdateUPD() and UpdateCLR() are used to update stored DRs based on the received packets. Changes caused due to OPT and UPD packets are handled by invoking UpdateUPD(), while those introduced by received CLR packets are made by calling UpdateCLR().

The function SetOps() is used to perform the various set operations required for CR-TORA. As explained earlier the UPD-list and CLR-list are represented by their cumulative

hashes in the stored DR. For example consider that a node X receives a CLR that wipes out its last downstream link for destination D. Now according to CR-TORA protocol, X would clear its stored height for D, and have to update its stored CLR-list.

For instance, if any of the CEIs included in X's UPD-list are listed in the received CLR packet, X has to create a new CEI. In order to achieve this functionality node X makes the call SetOps$(n, \{x_1 \cdots x_n\}, m, \{y_1 \cdots y_m\}, INT)$, where $\{x_1 \cdots x_n\}$ represents the stored UPD-list and $\{y_1 \cdots y_n\}$ represents the received CEIs. The function would return a self-MAC μ_{set} that specifies the result of performing an intersection on the inputs sets. Now along with the received and stored DRs for D, X also submits this self-MAC μ_{set} to the function UpdateCLR(). In this fashion the function SetOps() can be used to carry out required set operations.

Finally the function SendCRT() can be used by a node X to relay stored DRs. If the DR has infinite height the function relays it immediately. However, the condition $T \leq t_x$ must be satisfied while relaying DRs with finite heights. This ensures that a particular DR cannot be sent until the expiry of its waiting period.

8.5 Security Offered by the TMMs

In this section we analyze how the proposed approach secures CR-TORA. The TMMs designed in this chapter are an extension to the basic TCB presented in Chapter 4. Hence the assurances offered by the basic TCB can also be extended to the TMMs used to secure CR-TORA. These TMMs use IOMTs to store acquired routing information to thwart the attacks listed in Section 4.3.3.

8.5.1 Assertion Statements

Lemma 1. A node X can not illegally modify information stored in an IOMT.

Proof. The designed TMMs have two IOMTs: i) main IOMT to store the height of X for each available destination, and ii) an auxiliary IOMT to store the heights of each neighbor for every destination. The roots of both the trees (r and r_a) are securely stored within the TMM. According to the property of a merkle tree, the root binds all the values that are stored in its leaves. Hence it is impossible for the node to change the information currently binded by the roots r and r_a. □

8.5.2 Assurances Offered by the TMMs

Theorem 6.6.1. A node X can not hide available downstream links for a destination D.

Proof. As shown in Section 8.1.1 the destination record (DR) stored for D is of the form:

$$\mathbf{D_c} = [\mathbf{D} \parallel d \parallel u \parallel T] \tag{8.6}$$

where d is the number of available downstream links. Whenever X submits a received routing packet (either to the function UpdateUPD() or UpdateCLR()) for further processing, the value of d is updated based on the newly arrived data. Additionally the value of d is also updated by LossOfLink(), whenever X looses a neighbor. Therefore at any given instant the value d represents the number of downstream neighbors available for D.

Further, **Lemma 1** states that node X can not introduce illegal modifications to the information stored in an IOMT. Hence X can not hide available downstream links to a destination D. □

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH

This chapter summarizes the contributions of this dissertation to secure MANET routing protocols, and outlines some possible extensions to this research.

The primary motivation for the research was that proactive security approaches - which attempt to ensure that nodes will not be able to misbehave - have many compelling advantages over reactive approaches which attempt to detect and react to inconsistencies. The specific advantages of proactive (as opposed to reactive) approaches are as follows:

Overhead: Proactive approaches mandate lower bandwidth overhead as the overhead for carrying over authentication is eliminated.

Efficiency: Improved efficiency results both from reduction in bandwidth overhead and the enhanced scope of assurances provided. Proactive approaches can be as efficient as the original protocols which simply assume that nodes will not misbehave. Reactive secure extensions to such protocols had to deliberately turn-off many efficiency-enhancing features of such protocols due to the fact that providing assurances for such complex features will mandate very high overhead to make them useful.

Universality: Reactive approaches are intricately tied to the exact nature of the protocol. Proactive approaches with higher level goals are not. It is for this reason that the approach proposed in this dissertation mandates very little changes to the original pro-

tocols. For all protocols the packet output by a node conforms to the original protocol specification. The only difference is that each packet is accompanied by one or more message authentication codes and a time stamp.

9.1 Contributions

While some researchers have investigated strategies to realize secure routing protocols through trustworthy computing, such approaches have been superficial. Such approaches make unjustifiable assumptions regarding the components of node that are trusted. At the very essence of trustworthy computing is the compelling need to reduce the trusted computing base to the extent feasible. Only a well trusted TCB can be meaningfully amplified to realize the desired assurances.

This dissertation makes several contributions in the area of securing MANETs using trustworthy computing. We provide a detailed description of trusted MANET modules (TMM), their functionality, and the interfaces exposed to access such functionality. In the approach based on TMMs, TMMs create routing data, send authenticated routing data to TMMs of neighboring nodes, receive authenticated data from TMMs of neighboring nodes. From the perspective of two TMMs the nodes that house them are untrusted middlemen who facilitate exchanges between the TMMs and store routing information.

One contribution of this dissertation was the trivial AtomicRelay() function to provide several assurances not provided by current secure routing protocols. The AtomicRelay() function simply needs to accept routing data from neighbors and relay them to neighbors while enforcing some simple rules (for example, incrementing the hop-count field by one).

Simply accepting and relaying data poses no significant challenge even for severely resource limited modules. The main challenge stems from the need to store routing data, as relayed routing information may be a function of several pieces of routing data received in the past.

Perhaps the most important contribution of this dissertation is the index ordered merkle tree (IOMT), which makes it possible for a TMM to virtually store the routing data. More specifically, while it is believed that a merkle tree is sufficient to virtually store dynamic data, we demonstrated that using a merkle tree opens up replay attacks due to the inability to confirm negative queries. This necessary ability is provided by using an IOMT instead of a merkle tree.

With the ability to store routing data using IOMTs our research proceeded to identify specific functionality required of TMMs to secure various routing protocols like AODV, DSR and CR-TORA.

As mentioned in our hypothesis, the tasks performed within the proposed TMMs are deliberately restricted to cryptographic hash functions and logical operations. Moreover, when TMMs are utilized to provide secure routing, only few MACs need to be appended to existing routing packets. Hence, the overhead required for realizing the proposed TCB is considerably reduced.

Another important contribution of this dissertation is a novel protocol - collision resistant TORA, developed to address several weaknesses of TORA. In particular, TORA is susceptible to collisions, has complex rules for maintaining the height of a node (for which reason no secure extension of TORA has been proposed in the literature to carry

over authentication in TORA), and demands high overhead when network partitions occur. CR-TORA was designed to simultaneously address all of TORA's weaknesses.

9.2 Scope for Future Work

Two avenues of future work include

1. investigating possible improvements to the TMM interfaces and functionality,
2. investigating the utility of IOMT in other application settings

In regard to the first avenue, one important addition can be identifying TMM functionality required for link-state based protocols. Note that all protocols that have been addressed by this dissertation are based on distance vector routing. Another possibility is to make necessary modifications to the interfaces and TMM functionality to enable the next specification of trusted platform modules (TPM) to support additional functionality. Currently, an extension of TPM specification is underway which focuses on mobile platforms [76, 77]. However, thus far the additional features in mobile TPMs are restricted to those features/restrictions demanded by mobile phone network operators and content distributors. The developed TMM functionality could be a useful addition to TPMs in mobile devices as many mobile devices of the future are expected to take part in ad hoc routing.

One of the primary differences between IOMT and a merkle tree is that in the merkle tree each leaf is independent. In an IOMT we deliberately introduce inter-dependencies between leaves to permit querying. More specifically, by ordering by a unique index we can facilitate negative queries. However many other types of inter-dependencies may be

imposed for performing different types of queries. Some of the ongoing work investigating such extensions of IOMT include [78] and [79].

9.3 Publications

The following publications resulted from the research performed towards the dissertation:

1. V. Thotakura, M. Ramkumar, Collision Resistant Temporally Ordered Routing Algorithm, International Journal of Information Sciences and Computer Engineering, pp 1-8, Vol 1, No 1, 2010.
2. V. Thotakura, M. Ramkumar, "Leveraging a Minimal Trusted Computing Base for Securing On-Demand MANET Routing Protocols", Proceedings: 9th International Information and Telecommunication Technologies Symposium (I2TS'2010), Brazil, Dec 2010.
3. V. Thotakura, M. Ramkumar, "Minimal TCB for MANET Nodes", Proceedings: 6th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, Niagara Falls, Oct 2010.
4. V. Thotakura, M. Ramkumar, Securing AODV and DSR Using Trustworthy Computing, Submitted to IEEE Transactions on Dependable and Secure Computing (TDSC)
5. V. Thotakura, M. Ramkumar, Design of a Trusted Module for Securing CR-TORA, Submitted to the IEEE Transactions on Information Forensics and Security (TIFS).

REFERENCES

[1] P.Johanson , D. Maltz, "Dynamic source routing in ad hoc wireless networks," *Mobile Computing,* Kluwer Publishing Company, 1996, ch.5 , pp. 153-181

[2] Charles E. Perkins, Elizabeth M. Belding-Royer, and Ian Chakeres, "Ad hoc On Demand Distance Vector (AODV) Routing," *IETF Internet draft, draft-perkins-manetaodvbis-00.txt,* Oct 2003

[3] V. Park and S. Corson, "Temporally Ordered Routing Algorithm (TORA) Version 1 Functional Specification", *IETF MANET, Internet Draft (work in progress)*, 2001.

[4] C Perkins, P Bhagvat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," ACM SIGCOMM Symposium on Communication, Architectures and Applications, 1994.

[5] P. Jacquet, P. Mühlethaler, T. Clausen, A. Laouiti, A. Qayyum, L. Viennot, "Optimized link state routing protocol for ad hoc networks," Proceedings of the 5th IEEE Multi Topic Conference (INMIC 2001)," 2001.

[6] Vincent D. Park and M. Scott Corson , "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks", *Proceedings of IEEE INFOCOM ' 97 ,* Kobe, Japan (April 1997)

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

[8] V. Thotakura, M. Ramkumar Collision Resistant Temporally Ordered Routing Algorithm, International Journal of Information Sciences and Computer Engineering, pp 1-8, Vol 1, No 1, 2010.

[9] V. Thotakura, M. Ramkumar, "Leveraging a Minimal Trusted Computing Base for Securing On-Demand MANET Routing Protocols", Proceedings: 9th International Information and Telecommunication Technologies Symposium (I2TS'2010), Brazil, Dec 2010.

[10] V. Thotakura, M. Ramkumar, "Minimal TCB for MANET Nodes", Proceedings: 6th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications, Niagara Falls, Oct 2010.

[11] V. Thotakura, M. Ramkumar, Securing AODV and DSR Using Trustworthy Computing, Submitted to IEEE Transactions on Dependable and Secure Computing (TDSC)

[12] V. Thotakura, M. Ramkumar, Design of a Trusted Module for Securing CR-TORA, Submitted to the IEEE Transactions on Information Forensics and Security (TIFS).

[13] URL: http://laptop.org/laptop/

[14] P. Papadimitratos, Z. Haas, "Secure Link State Routing for Mobile Ad Hoc Networks," in Proceedings of the IEEE CS Workshop on Security and Assurance in Ad hoc Networks, 2003.

[15] G. Pei, M. Gerla, T. Chen, "Fisheye State Routing in Mobile Ad Hoc Networks," ICDCS Workshop on Wireless Networks and Mobile Computing, D71-D78, 2000.

[16] Y-C Hu, A. Perrig, D.B. Johnson, "Packet Leashes: A Defense against Wormhole Attacks in Wireless Ad Hoc Networks," Rice University Department of Computer Science Technical Report TR01-384, Dec 2001.

[17] P Papadimitratos, Z. J.Haas, "Secure Routing for Mobile Ad Hoc Networks," Proceedings of the SCS Communication Networks and Distributed Systems Modeling and Simulation Conference(CNDS 2002), San Antonio, Texas ,2002

[18] Y-C Hu ,A Perrig,. D B.Johnson, "Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks," Journal of Wireless Networks, **11**, pp 11–28, 2005.

[19] J. Kim, G. Tsudik, "SRDP: Securing Route Discovery in DSR," IEEE Mobiquitous'05, July 2005.

[20] M G Zapata, N.Asokan ,"Securing Ad Hoc Routing Protocols," Proceedings of the ACM workshop on Wireless security, Atlanta, Georgia, September 2002.

[21] X. Du, Y. Wang, J. Ge, Y. Wang, "A Method for Security Enhancemens in AODV Protocol," Proceedings of the 17th International Conference on Advanced Information Networking and Applications, AINA03, Xian, China.

[22] T Wan, E kranakis P.C. Van Oorschot,"Securing the Destination Sequenced Distance Vector Routing Protocol," Proceedings of the 6th International Conference on Information and Computer Security, Malaga, Spain, October 2004.

[23] Y-C Hu, D B. Johnson, A Perrig, "SEAD: Secure Efficient Distance Vector Routing for Mobile Wireless Ad Hoc Networks," Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications, 2002.

[24] M. Burmester, T. Van Le, M. Weir, "Tracing Byzantine Faults in Ad Hoc Networks," Proceedings of Communication, Network, and Information Security (CNIS), NY, Dec 2003.

[25] TCG Specification: Architecture Overview, Specification Revision 1.4, 2nd August 2007.

[26] S.W. Smith, S. Weingart, "Building a High-Performance Programmable Secure Coprocessor," IBM Technical Report RC21102, Feb 1998.

[27] M. Ramkumar, "The Subset Keys and Identity Tickets (SKIT) Key Distribution Scheme," IEEE Transactions on Information Forensics and Security, **5**(1), pp 39–51, March 2010.

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

[29] M.G.Zapata, N.Asokan, "Securing Ad hoc routing protocols," WISE-02, Atlanta, Georgia, 2002.

[30] E. Weiss, G. R. Hiertz, B. Xu, "Performance Analysis of Temporally Ordered Routing Algorithm based on IEEE 802.11a," IEEE 61st Vehicular Technology Conference (VTC), May 2005.

[31] B. Awerbuch, D. Holmer, C. Nita-Rotaru, H. Rubens, "An On-Demand Secure Routing Protocol Resilient to Byzantine Failures," ACM Workshop onWireless Security (WiSe–02), September 2002.

[32] A. Perrig, R. Canetti, D. Song, D. Tygar, "Efficient and Secure Source Authentication for Multicast," in Network and Distributed System Security Symposium, NDSS '01, Feb. 2001.

[33] K.A. Sivakumar, M. Ramkumar, "On the Effect of Oneway Links on Route Discovery in DSR," Proceedings of the IEEE International Conference on Computing, Communication and Networks, ICCCN-2006, Arlington, VA, October 2006.

[34] K.A. Sivakumar, M. Ramkumar, "An Efficient Secure Route Discovery Protocol for DSR," to be presented in IEEE Globecom 2007, Washington, DC, Nov 2007.

[35] A. A. Pirzada, A. Datta, and C. McDonald, "Trustworthy Routing with the TORA Protocol," *Proceedings of the AusCERT Asia Pacific Information Technology Security Conference*, 2004.

[36] Tingyao Jiang, Qinghua Li, Youlin Ruan, "Secure Dynamic Source Routing Protocol," cit,pp.528-533, Fourth International Conference on Computer and Information Technology (CIT'04), 2004

[37] Jihye Kim , Gene Tsudik, "SRDP: Securing Route Discovery in DSR", *Proceedings of the The Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, p.247-260, July 17-21, 2005

[38] K.A. Sivakumar, M. Ramkumar, "Safeguarding Mutable Fields in the AODV Route Discovery Process," the Sixteenth IEEE ICCCN-07, Honolulu, HI, Aug 2007.

[39] K.A. Sivakumar, M. Ramkumar, "Improving the Resilience of Ariadne," IEEE SPAWN 2008, Newport Beach, CA, June 2008.

[40] S. Bucheggar. J-Y Le Boudec, "Nodes bearing grudges: Towards routing security, fairness, and robustness in mobile ad hoc networks," 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, Spain,2002.

[41] S Marti, T. J. Giuli, Kevin Lai and Mary Baker,"Mitigating routing misbehavior in mobile ad hoc networks," Proceedings of the 6th Annual International Conference on Mobile Computing and Networking, Boston, 2000.

[42] J.Marshall, V.Thakur, A.Yasinsac,"Identifying flaws in the secure routing protocol," Proceedings of the 2003 IEEE International Performance, Computing, and Communications Conference, 2003.

[43] A. A. Pirzada, C. McDonald, "Trusted Route Discovery with TORA Protocol," Proceedings of the Second Annual Conference on Networks and Services Research (CNSR), 2004.

[44] S. Buchegger, J.-Y. Le Boudec, "Performance Analysis of the CONFIDANT Protocol," Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking and computing, Lausanne, Switzerland, 2002.

[45] P. Michiardi, R. Molva, "CORE: A COllaborative REputation mechanism to enforce node cooperation in Mobile Ad Hoc Networks," Communication and Multimedia Security Conference, Portoroz, Slovenia, Sep 2002.

[46] P. Dewan, P. Dasgupta, A. Bhattacharya, "On Using Reputations in Ad hoc Networks to Counter Malicious Nodes," QoS and Dynamic Systems, Newport Beach, USA, 2004.

[47] S. Ganeriwal, M. B. Srivastava, "Reputation-based framework for high integrity sensor networks," Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks, Washington, DC, 2004.

[48] J. Liu, V. Issarny, "Enhanced Reputation Mechanism for Mobile Ad Hoc Networks," Proceedings of the Second International Conference on Trust Management (iTrust'2004), March 2004.

[49] M. Jarrett and P. Ward, "Trusted Computing for Protecting Ad-hoc Routing," Proceedings of the 4th Annual Communication Networks and Services Research Conference, IEEE Computer Society, May 2006.

[50] J-H. Song, V. Wong, V. Leung, "Secure Routing with Tamper Resistant Module for Mobile Ad Hoc Networks," ACM SIGMOBILE Mobile Computing and Communications Review, vol. 7, no. 3, ACM Press, New York, Jul. 2003.

[51] L. Buttyan, J. Hubaux, "Nuglets: a Virtual Currency to Stimulate Cooperation in Self-Organized Mobile Ad Hoc Networks," Technical Report DSC/2001/001, Department of Communication Systems, Swiss Federal Institute of Technology, Jan. 2001.

[52] J-P. Hubaux, L Buttyan, S. Capkun, "Quest for Security in Mobile Ad Hoc Networks," Proceedings of the ACM MOBIHOC 2001.

[53] K. Sivakumar, M. Ramkumar, "APALLS: A Secure Routing Protocol," manuscript under preperation.

[54] B. Gaines, M. Ramkumar, "A Framework for Dual Agent MANET Routing Protocols," IEEE Globecom 2008.

[55] M. Liu, Q. Meng, Y. He and D. Cheng, "The Ad Hoc Network Security Research Based on Trusted Computing and Semantic Remote Attestation," Proceedings: 2nd International Workshop on Educational Technology and Computer Science, Wuhan, China, March. 2010, pp. 724-727.

[56] M. Dejun, R. Shuai, Z. Degang and Z. Tao, 2008. Application of trusted computing to the adhoc networks security. Asia J. Inform. Technol., pp 370-373, Vol 7, No 8, 2008.

[57] Joo-Han Song, Vincent Wong, Victor Leung, and Yoji Kawamoto. 2003. Secure routing with tamper resistant module for mobile Ad hoc networks. SIGMOBILE Mob. Comput. Commun. Rev. 7, 3 (July 2003), pp. 48-49.

[58] Jarrett, M.; Ward, P.; , "Trusted computing for protecting ad-hoc routing," Communication Networks and Services Research Conference, 2006. CNSR 2006. Proceedings of the 4th Annual , vol., no., pp.8 pp.-68, 24-25 May 2006

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

[60] V.L. Chee and W.C. Yau, "Security Analysis of TORA Routing Protocol", *ICCSA 2007, LNCS 4706, Part II,* pp. 975986, 2007.

[61] A.A. Pirzada and C. Donald. "Secure Routing with the AODV Protocol," *IEEE*, 2005.

[62] L. Li, C. Chigan. "Token Routing: A Power Efficient Method for Securing AODV Routing Protocol," *IEEE*, 2006.

[63] Y. Hu and A. Perrig, "A Survey of Secure Wireless Ad Hoc Routing," IEEE Security and Privacy Magazine, vol. 2, no. 3, May-June 2004, pp. 28-39.

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

[65] A. Broder, M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," Internet Mathematics, vol. 1, no. 4, pp 485–509, 2005.

[66] M. Ramkumar, "Trustworthy Computing Under Resource Constraints with the DOWN Policy," *IEEE Transactions on Dependable and Secure Computing*.

[67] Patroklos Argyroudis and Donal O'Mahony, "Secure Routing for Mobile Ad hoc Networks," IEEE Communications Surveys and Tutorials, vol. 7, no. 3, pp 2-21, 2005.

[68] M. Ramkumar "An efficient broadcast authentication scheme for ad hoc routing protocols," IEEE ICC 2006, Istanbul, Turkey, Jun 2006.

[69] M. Ramkumar, "Broadcast Encryption Using Probabilistic Key Distribution and Applications," Journal of Computers, Vol 1 (3), June 2006.

[70] M. Scott Corson, S. Papademetriou, Philip Papadopoulos, Vincent D. Park, and gmir Qayyum. An Intemet MANET Encapsulation Protocol (IMEP) Specification. Interact-Draft, draft-ieff-manet-imep-spee- 01.txt, August 1998. Work in progress.

[71] T. Clausen, C. Dearlove, and J. Dean, "MANET Neighborhood Discovery Protocol (NHDP) draft-ietf-manet-nhdp-07," *IETF MANET, Internet Draft*, 2008.

[72] H. Ehsan and Z. Uzmi. "A Performance Comparison of Ad-Hoc Wireless Network Routing Protocols," *IEEE*, 2004.

[73] J. Cano and P. Manzoni. "A Performance Comparison of Energy Consumption for Mobile Ad-Hoc Network Routing Protocols," *IEEE*, 2000.

[74] E.M. Royer and C.-K. Toh, "A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks," *IEEE Personal Communications,* pp. 46-54, 1999.

[75] E. Gafni and D. Bertsekas, Distributed algorithms for generating loop-free routes in networks with frequently changing topology, *IEEE Trans. Commun.* (January 1981).

[76] Mooseop Kim, Hongil Ju, Youngsae Kim, Jiman Park, and Youngsoo Park , "Design and implementation of mobile trusted module for trusted mobile computing," *IEEE Transactions on Consumer Electronics*, vol.56, no.1, pp.134-140, February 2010

[77] J. Ekberg and M.Kylanpaa, "Mobile Trusted Module (MTM) - an introduction," *Report of Nokia Research Center*, 2007.

[78] A. Velagapalli, M. Ramkumar, "An Efficient Trusted Computing Base (TCB) for a SCADA System Monitor," submitted to the ACM/IEEE Second International Conference on Cyber-Physical Systems (ICCS 2011), Chicago, IL.

[79] S. Mohanty, M. Ramkuamr, “Ordered Merkle Trees for Trusted File Storage,” submitted to the 1st International Conference on Cloud Computing (CLOSER 2011), Noordwijkerhout, Netherlands.