Secure Sensor Network Routing: A Clean-Slate Approach*

Bryan Parno†
Carnegie Mellon University
parno@cmu.edu

Mark Luk
Carnegie Mellon University
mluk@ece.cmu.edu

Evan Gaustad
Carnegie Mellon University
egaustad@cmu.edu

Adrian Perrig
Carnegie Mellon University
adrian@ece.cmu.edu

ABSTRACT

The deployment of sensor networks in security- and safety-critical environments requires secure communication primitives. In this paper, we design, implement, and evaluate a new secure routing protocol for sensor networks. Our protocol requires no special hardware and provides message delivery even in an environment with active adversaries. We adopt a clean-slate approach and design a new sensor network routing protocol with security and efficiency as central design parameters. Our protocol is efficient yet highly resilient to active attacks. We demonstrate the performance of our algorithms with simulation results as well as an implementation on Telos sensor nodes.

1. INTRODUCTION

Sensor networks provide economically viable solutions for a wide variety of applications, including surveillance of critical infrastructure, safety monitoring, and many health-care applications. As sensor networks are increasingly deployed in such security- and safety-critical environments, the need for secure communication primitives is self-evident. Likewise, the development of secure primitives enables the use of sensor networks in new applications.

The central goal of this work is to ensure node-to-node message delivery, even if the sensor network is under active attack. In the presence of an attacker, it is an extremely challenging task to maintain correct routing information; the attacker could inject malicious routing information or alter legitimate routing setup/update messages. Even when route setup/update messages are authenticated, compromised sensor nodes can supply incorrect routing information of their own and cripple the routing infrastructure. As we discuss in Section 12, most proposals for sensor network routing protocols assume a trusted environment and cannot function under attacks [15] (two exceptions are INSENS, which routes only between nodes and a central basestation [4], and SIGF, which relies on nodes knowing their geographic location [33]).

In general, there are three main directions for designing secure routing protocols: *prevention*, *detection / recovery*, and *resilience*. The *prevention* approach seeks to harden the protocol against attacks, typically through cryptographic mechanisms that restrict participants' actions. The prevention approach is generally the most efficient and effective approach, but it only forestalls known attacks. *Detection* involves monitoring the real-time behavior of protocol participants. Once malicious behavior is detected, we resort to *recovery* techniques to eliminate malicious participants and to restore network order and functionality. The detection approach can guard against potentially unknown attacks, as long as we can distinguish anomalous behavior and correctly attribute it to a misbehaving entity. The *resilience* approach seeks to maintain a certain level of availability even in the face of (possibly unpredicted) attacks. Ideally, this approach should provide graceful performance degradation in the presence of compromised network participants, i.e., the availability of the network should degrade no faster than a rate approximately proportional to the percentage of compromised participants.

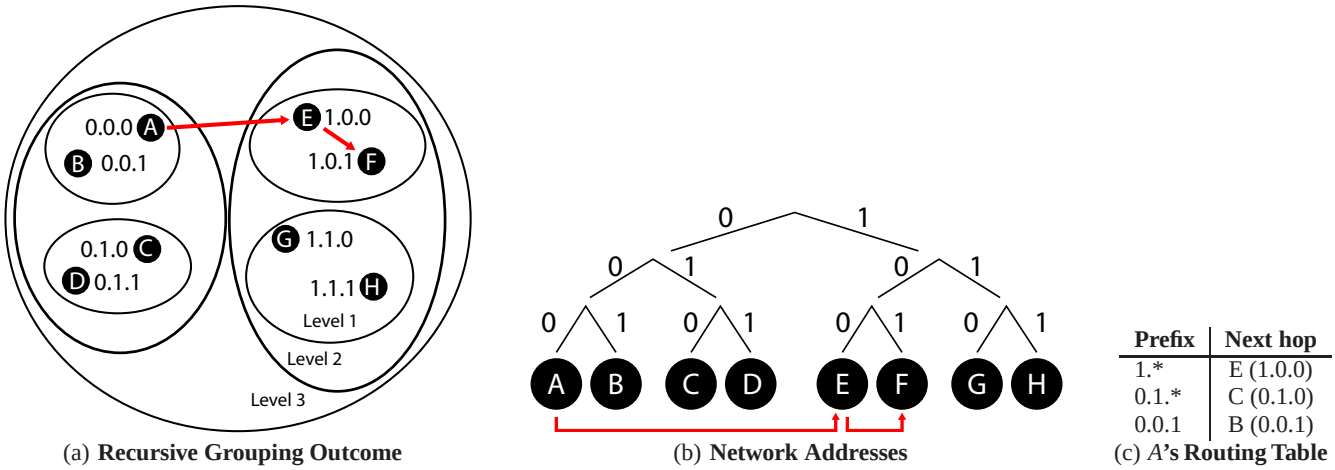
Previous secure routing protocols usually rely on a single approach. The majority of secure routing mechanisms focus exclusively on the prevention approach, since it is the most efficient and effective against known attacks; an example is the S-BGP protocol [16]. Many researchers propose detection and recovery mechanisms; for example, the watchdog and pathrater scheme attempts to identify malicious behavior in ad hoc networks [19], and secure traceroute mechanisms attempt to locate malicious nodes along a routing path in the Internet [23]. To provide resilience, many protocols (e.g., the INSENS protocol [4]) turn to multipath routing, hoping at least one of the paths will be unaffected by an attacker.

The central contribution of this work is to start from a clean slate and systematically design a general-purpose secure routing protocol that incorporates all three design principles. Our goal is to design a highly secure, highly available node-to-node sensor network routing protocol. Point-to-point routing is essential for many sensor network protocols, including Geographic Hash Tables (GHTs) [29] and certain key distribution schemes like PIKE [2].

In our routing protocol, we dynamically establish routing tables and network addresses for each node using techniques that prevent interference from an active attacker. We then apply resilient routing

†Bryan Parno is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense.
*This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, grants CNS-0347807 and CCF-0424422 from the National Science Foundation, and by a gift from Bosch. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, Bosch, CMU, NSF, the U.S. Government or any of its agencies.

CoNEXT 2006 Lisboa, Portugal



(a) **Recursive Grouping Outcome** (b) **Network Addresses**

Prefix	Next hop
1.*	E (1.0.0)
0.1.*	C (0.1.0)
0.0.1	B (0.0.1)

(c) ***A*'s Routing Table**

Figure 1: Protocol Overview. During the recursive grouping algorithm, smaller groups repeatedly merge to form larger groups. Figure 1(a) shows the resulting groups. The nodes are labeled with their IDs (*A*, *B*, etc.), as well as their resulting network addresses. Figure 1(b) illustrates how the network addresses grow as groups merge and add bits to differentiate themselves from their merge partner. The arrows in Figures 1(a) and 1(b) indicate how a message from node *A* to node *F* would be routed. Note that while the network addresses form a tree, we do not route along the tree. Figure 1(c) shows node *A*'s routing table, which maps progressively longer network address prefixes to next-hop neighbors.

techniques to transmit packets, while incorporating mechanisms to detect and eliminate malicious nodes. The memory overhead for our routing protocol is small: each sensor node stores one routing entry for each prefix of its address, representing the next-hop neighbor node to move towards the destination area. Most importantly, however, we can secure every step of our routing protocol against internal and external attacks and ensure high availability for packet forwarding.

2. ASSUMPTIONS

In this work, we assume the presence of a network authority (NA), with public key K_{NA} and private key K_{NA}^{-1}. Each sensor node must be preloaded with the network authority's public key K_{NA}. We assume each node receives a unique ID_α (drawn from a total order) along with a certificate $\{ID_\alpha\}_{K_{NA}^{-1}}$ digitally signed by NA. The NA uses a signature scheme that places the majority of the burden on the signer, so that verification can be extremely efficient. For example, verifying a Rabin signature only requires a single modular multiplication [20]. As an alternative, the network authority could construct a hash tree over all of the node IDs and preload each node with the root value and the intermediate values needed to authenticate its ID to other nodes. This would limit the verification cost to a few hashes, but it would come at the expense of additional communication overhead.

The NA also provides each node with a set of $\kappa = O(\log n)$ randomly chosen, verifiable challenge values $C_0, ..., C_\kappa$. These will be used to detect deviations during the route setup process. We can store the challenges compactly using a standard construction based on one-way hash chains [4, 35]. We provide each sensor node α with a one-way hash chain, $\Psi = \{(C_0, C_\kappa) : h^\kappa(C_\kappa) = C_0\}$, where $h^\kappa(y)$ signifies applying the hash function κ times starting with the value y. Each initial challenge C_0 is signed by the network authority (giving the node $\{C_0||ID_\alpha\}_{K_{NA}^{-1}}$) using the same signature scheme as before. To provide an authenticated challenge during round i, node α can release challenge C_i (calculated as $h^{\kappa-i}(C_\kappa)$), along with C_0, ID_α and $\{C_0||ID_\alpha\}_{K_{NA}^{-1}}$. Other nodes can verify the challenge by verifying $\{C_0||ID_\alpha\}_{K_{NA}^{-1}}$ and checking that $h^i(C_i) = C_0$.

During the address setup phase of our recursive grouping protocol, we assume that the nodes within a group use a reliable broadcast mechanism to communicate with one another. This mechanism could take the form of a simplistic flooding protocol, though in our implementation, we leverage the partial routing information possessed by each node (see Section 11 for details). As a result, a malicious node that drops messages (or an adversary that jams particular network links) will slow the address setup process but will not subvert it.

Finally, our protocol is designed for networks in which the nodes are primarily stationary, though we can tolerate occasional periods of mobility by rerunning our algorithm for establishing routing information.

3. PROTOCOL OVERVIEW

Our protocol assigns a network address to each node and establishes routing tables using a recursive grouping algorithm. For a given topology, the algorithm proceeds entirely deterministically, *preventing* attacks on routing information and limiting a subverted node's ability to perform malicious actions. When the recursive grouping algorithm terminates, each node has a unique network address, as well as a routing table that implicitly maps variable-length address prefixes to next-hop node neighbors. Figure 1(a) illustrates the resulting set of recursive groups; the nodes shown all belong to the same level 3 group, which is subdivided into level 2 subgroups that are in turn divided into level 1 subgroups. Each time two groups merge, they extend their network addresses with an additional bit as shown in Figure 1(b). While the resulting network addresses form a binary tree, our routing algorithm does not perform tree routing (i.e., it does not route along the tree structure, as evidenced by the fact that the internal "nodes" in the address tree do not correspond to actual sensor nodes), since tree routing is typically inefficient.

Instead, our routing design is based on area hierarchies as described by Kleinrock and Kamoun [18], and it guarantees that the size of the routing table, the size of the network addresses, and the

length of routing paths (as measured in logical hops through the space of network addresses) will be bounded by $\lceil \log_2 n \rceil$, where n is the number of nodes in the network.

Our protocol incorporates several techniques (described in Section 7) for *detecting* malicious behavior. We actively detect attempts by a single node to acquire multiple identities in the network. We also use a Grouping Verification Tree (GVT) to detect deviations during the recursive grouping protocol. When a malicious node is detected, we use a Honeybee technique to eliminate it from the network, allowing the network to *recover* from intrusions.

Once the routing information has been established, each sensor can route a packet by forwarding it to the entry in its routing table with the longest matching prefix (Figure 1(c) provides a sample routing table for node A). The arrows in Figures 1(a) and 1(b) illustrate the path taken when node A sends a packet to node F. We add *resiliency* to the routing process by providing the sender with a limited degree of control over the path taken by its packets. This allows a sender to route around both natural and malicious problems in the network and to choose paths based on additional metrics, such as energy efficiency.

4. ADDRESS AND ROUTING SETUP

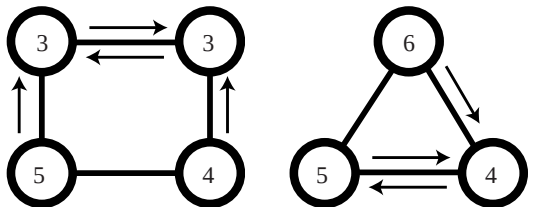
We begin this section with an overview of the recursive grouping algorithm that we use to establish addressing and routing information. Then we describe the necessary initialization steps and the algorithm itself. Finally, we discuss techniques to handle both the removal and the addition of nodes.

4.1 Setup Overview

Our recursive grouping algorithm assigns a unique network address to each node in the network, while also populating each node's routing table. Initially, every sensor node comprises its own group. Then, the algorithm repeatedly merges groups of nodes into larger groups. A group G initiates a merge by sending a merge proposal to the smallest neighboring group G'. Choosing the smallest neighbor encourages groups of similar size to merge, thus keeping the resulting network address space dense and compact (we analyze the efficiency of the recursive grouping algorithm in Section 8.2). If G' also proposes to G, then the two groups merge to form a new group. After each merge, the nodes in group G each add a bit to their network addresses to differentiate themselves from the nodes in group G', and also add an entry in their routing tables indicating a neighboring node that is a path to any node in group G'. The nodes in G' make similar changes. This process continues until the entire network converges into a single group.

At each stage of the grouping algorithm, the groups are assigned IDs that, along with their sizes, can be authenticated using a Grouping Verification Tree (GVT). We discuss the use of the GVT to perform authentication and detect deviation from the grouping algorithm in Section 7.1.

Outcome. At the completion of the grouping protocol, each node will have a unique network address, a routing table that maps variable-length address prefixes to next-hop neighbors, and a merge table that will be used to secure each step of the algorithm. The network address will be of the form $\mathcal{R} = R_{r-1}||...||R_0$, where each entry $R_i \in \{0,1\}$ and r indicates the number of times that node (or the group containing that node) has merged with another group. Each node will also have a routing table $\mathcal{T}[i]$ where $0 \leq i < r$. Each entry $\mathcal{T}[i]$ in a node's routing table maps an address prefix to a next-hop neighbor that leads toward a node with a network address matching the prefix. Finally, each node will have a merge table $\mathcal{M}[i]$ where $0 \leq i < r$ that records the ID and size of each group it merges with.



(a) Two groups of size three agree to merge and refuse merge proposals from the larger groups.

(b) The group of size four and the group of size five now agree to merge.

Figure 2: Recursive Grouping. Each group is labeled with its size, and lines connect two groups that can communicate with each other. The arrows indicate a merge proposal from one group to another. The figures show two intermediate steps in the recursive grouping algorithm. After one additional merge (not shown), the entire network will consist of a single group.

4.2 Initialization

When routing establishment is initiated, each node constitutes its own group, and the group's ID corresponds to the node's unique ID. Before beginning the recursive grouping described below, the nodes perform a secure neighbor discovery protocol. Each node α broadcasts $(ID_\alpha, \{ID_\alpha\}_{K_{NA}^{-1}})$ (i.e., its ID and the accompanying certificate from the network authority) to its neighbors. The neighbors verify the signature and add node α to their neighbor lists. An external adversary cannot inject a manufactured ID, since it will be unable to produce a proper signature to match it. A compromised node will be unable to alter its ID without invalidating its signature.

We bound the time allocated for the secure neighbor discovery protocol and require every node to announce its ID during this period.[1] After the discovery protocol concludes, nodes in the network will no longer accept new neighbors, preventing an adversary from attempting to insert malicious nodes at a later point. We also use a replication detection algorithm (described in Section 7.2) to detect an attacker who replays certificates from legitimate nodes.

4.3 Recursive Grouping

Our recursive grouping algorithm proceeds in an asynchronous, distributed fashion. A group G first collects information about the current size of each neighboring group. It then proposes to merge with the smallest neighbor G' and waits for a response. If that group does not wish to merge with G, then G considers the merge a failure, and restarts its merge process by redetermining its smallest neighbor. Favoring the smallest neighbor makes it more likely that groups of similar size will merge, and hence the size of a group will approximately double during each round. We analyze this effect more precisely in Section 8.2. If G' also proposes to merge with G, the two groups merge to form a new group γ (Figure 2 illustrates this process). Below, we describe the protocol in detail.

Determining Neighboring Group Information. Consider a group G at an intermediate stage of the grouping process. Some portion of the nodes in group G have neighboring nodes not in the group. We refer to these nodes as *edge nodes*. If all of a node's neighbors belong to its group, then we refer to it as an *interior node*. The edge nodes are responsible for communicating with neighboring groups and forwarding the results of merge agreements to the inte-

[1] If the nodes are loosely synchronized, the neighbor discovery period can have a realtime bound. Otherwise, an authenticated broadcast from the network authority can signal the end of the protocol.

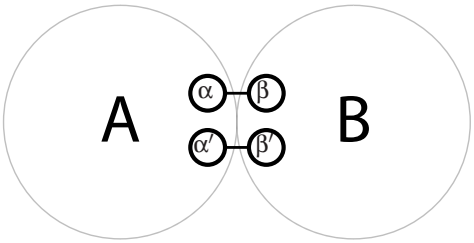


Figure 3: Merging Groups. In the figure, α, α' and β, β' are edge nodes for groups *A* and *B*, respectively. Nodes α and β and nodes α' and β' are neighbors. Both β and β' will decide that group *B* should merge with group *A*, and each will forward this decision to the rest of group *B*. This will only result in a single broadcast within *B*, since internal nodes will suppress duplicates.

rior nodes (see Figure 3). Since different edge nodes will hear from different neighboring groups, they must communicate with each other to decide on a merge target (i.e., the group to which to send a merge proposal). To facilitate coordination within the group, every node in the group tracks the current size of the group ($|G|$) and also maintains a list of the neighboring group IDs ($\Gamma = \{g_1, g_2, ..., g_k\}$).

To initiate a merge, each edge node floods its group with the size and ID of each neighboring group it borders. After an edge node receives $|\Gamma|$ distinct announcements, it has heard about all of the neighboring groups.

Exchanging Merge Proposals. Once the edge nodes know about all of the neighboring groups, they each independently compute the neighboring group G' with the smallest size[2] (with a tie broken in favor of the group with the smallest ID), and any edge node that borders G' will broadcast a merge proposal to that group. If the merge target refuses to merge (by announcing that it has already selected another group), then the edge nodes propagate the refusal to the rest of the group and the edge nodes once again compute the smallest neighboring group. If G' also selects G as a merge target, then the two groups will merge.

Merging. If groups G and G' agree to merge, they will form a new group γ. As a result of this merge, all of the nodes in γ must compute the new group ID_γ, record information about the group they merged with, add an entry to their network addresses, and update their routing tables and neighboring group lists.

First, all nodes in the group can independently compute the new group ID as:

$$ID_\gamma = \begin{cases} h(ID_G, |G|, ID_{G'}, |G'|) & ID_G < ID_{G'} \\ h(ID_{G'}, |G'|, ID_G, |G|) & ID_G > ID_{G'} \end{cases} \quad (1)$$

The new group ID incorporates the IDs and sizes of both of the old groups. The method for computing the new group ID is designed to avoid collisions in the group ID space. Each node in group G also updates its merge table such that $\mathcal{M}[i] = (G', |G'|)$. This information will be used later to authenticate the merge process.

In addition, every node that belonged to group G must add an entry to its network address to differentiate itself from the nodes that belonged to G' (and vice versa). Each node in group G updates its network address such that

$$R_i = \begin{cases} 0 & ID_G < ID_{G'} \\ 1 & ID_G > ID_{G'} \end{cases} \quad (2)$$

where i represents the number of times that particular node has merged. Nodes in G' use an analogous equation but with G and G' transposed.

As a result of the merge, each node in G (and in G') adds an entry to its routing table. For a node in G, the new entry indicates the ID of the node that informed it about G'. In other words, if node α learned about G' from node κ, then α adds an entry to its routing table such that $\mathcal{T}[i] = \kappa$. Node κ will serve as node α's next-hop neighbor when it needs to forward a packet to a node in group G'.

Finally, each node in G will merge its neighbor list Γ with the neighbor list Γ' provided by G' to obtain the neighbor list $\hat{\Gamma}$ for the new group. Thus, each node will always have an updated list of the groups that neighbor its own group.

Post-Merge. After the nodes in the newly formed group γ have computed the updates described above, the edge nodes in γ broadcast ID_γ, $|\gamma|$ and $\hat{\Gamma}$ to the neighboring groups, and once again begin the process of determining the smallest neighboring group. This grouping process continues until all of the nodes in the network have merged into a single group.

4.4 Network Maintenance

Node Death. During a sensor network's lifetime, some nodes will inevitably fail due to battery exhaustion or other mechanical problems. To adapt to sensor death, nodes must update their routing tables, but the network addresses can remain unchanged. In general, the resilient routing techniques described in Section 5.2 will allow nodes to route around dead companions. However, we also include a provision allowing nodes to bootstrap replacement information from their neighbors.

If node α decides that its neighbor β_j has died, it determines the index i of β_j's position in α's routing table (β_j may appear as a next hop more than once, but the same protocol applies). Then, α will broadcast a request to its neighbors, asking for a node other than β_j that will route to group i. Each neighbor will respond unless it too has no valid entries remaining, in which case it begins a similar inquiry. If a neighbor β_k replies with node β_m, then if β_m is one of α's neighbors, it will set $\mathcal{T}[i] = \beta_m$; otherwise, α sets $\mathcal{T}[i] = \beta_k$. Thus, the network routing topology will adapt as nodes die out.

Node Addition. Many applications require the ability to add nodes to the network after the initial deployment. Since such additions occur relatively infrequently, we propose periodically rerunning the recursive grouping algorithm. If the nodes are loosely synchronized, then time could be divided into epochs, and the network would run the recursive grouping algorithm at the beginning of each epoch. Otherwise, the network authority could use an authenticated broadcast to indicate the need to rerun the recursive grouping algorithm. Rerunning the algorithm would allow newly added nodes to acquire network addresses and enable the network to systematically adapt to the presence of dead nodes and to the current power levels of previously deployed nodes.

5. FORWARDING

In this section, we describe how nodes use the routing information established during the recursive grouping algorithm to forward packets. We also present a simple modification that adds resilience to the forwarding process by allowing the sender control over the forwarding path. Finally, we add an important optimization for providing efficient forwarding.

[2]All edge nodes will arrive at the same conclusion, since they all have the same information about the sizes and IDs of the neighboring groups.

5.1 Basic Forwarding

In the simplest case, our scheme uses area style forwarding [18] based on the secure network addresses and routing tables established by the recursive grouping algorithm. Each node's network address reflects its logical position in the network, and the routing table entries tell a node how to forward a packet to a group containing the destination address.

When the grouping algorithm concludes, the network will have merged into a single group, as Figure 1(a) illustrates. In the figure, all of the nodes in the network belong to the same level 3 group. Within the level 3 group, the nodes are divided into two level 2 groups (a 0-group and a 1-group, distinguished by the most significant entry of their addresses), each of which is further divided into two level 1 groups. Within the level 1 groups, each node has a level 0 identifier that distinguishes it from the other node in the same level 1 group. Thus, a node's full network address is the concatenation of its level identifiers, sorted in reverse order by level. For instance, node 0.0.1 has a level 0 identifier of 1, and an identifier of 0 for levels 1 and 2.

Before describing forwarding in detail, we first provide some intuition. When a node with network address N receives a message destined for address D, it finds the most significant digit between D and N that differs and sends the message towards the other group at the corresponding level. For example, if $D = 0.1.0$ and $N = 0.0.1$, then the fact that D and N share the same identifier (0) in the most significant bit of their addresses implies that they are in the same level 3 group. However, the next bit reveals that within that group, D resides in the level 2 group with a 1 identifier, while N resides in the level 2 group with a 0 identifier. Thus, N will forward the packet towards the level 2 group with a 1 identifier. Note that multiple physical hops may be required to reach that group. The first node in the level 2 group with a 0 identifier that receives this message will start matching bits at subsequent levels.

In terms of the information established by the recursive grouping algorithm, suppose a node with address $\mathcal{R} = R_{r-1}||...||R_0$ wishes to send a message to a node with address $\mathcal{R}' = R'_{r-1}||...||R'_0$. The sender begins by comparing the most significant digits in the two addresses (i.e., it checks R_{r-1} against R'_{r-1}). If these digits match, it proceeds to the next most significant digit, until it finds a digit such that $R_i \neq R'_i$ (this must happen, since the sender is presumably not sending messages to itself). The sender consults its routing table entry for position i and forwards the message to the next hop neighbor recorded in $\mathcal{T}[i]$. This ensures that the message will eventually reach a node that will match on address digits $R'_{r-1},...,R'_i$.

5.2 Resilient Forwarding

To achieve high availability of message delivery, we leverage multi-path forwarding. We extend the routing tables established during the recursive grouping algorithm to include multiple next-hop nodes at each stage. We then modify the forwarding algorithm to allow a sender to loosely select amongst the possible paths to a destination. Thus, the sender can route packets around problem areas (whether caused by malice or malfunction) in the network.

To add resiliency to our protocol, we can take advantage of the natural redundancy present in the recursive grouping algorithm to add additional options to each node's routing table. As described in Section 4.3, when two groups merge, each node creates one entry in its routing table to store the ID of the neighbor from which it heard about the merge target. In practice, a group is likely to have multiple edge nodes that all flood the group with information about the same neighboring group (e.g., in Figure 3, both β and β' will inform nodes in group B about group A). In this case, each node may hear about the neighboring group from multiple neighboring nodes. Instead of only storing the first such neighboring node as the next hop to reach that group, the node will remember three entries, L, M and R for each routing table entry $\mathcal{T}[j]$, with each L, M and R representing a different neighboring node that informed the node of the j^{th} neighboring group that it merged with. These additional entries will be used by the sender to select a path through the network.

We use the redundant routing table entries to route around both natural and malicious problems in the network. When a node sends a packet, by default, each node along the path will forward it to the M entry in the appropriate routing table entry. However, a sender can also choose to include a direction string $\Delta = \delta_0||\delta_1||...||\delta_k, \delta_i \in \{L,M,R\}$, $0 \leq i \leq k$. If the sender's address differs from the recipient's address in λ least significant digits, then the path will consist of λ logical hops, but the sender should set $k > \lambda$, since each logical hop may require multiple physical hops.

When an intermediate node receives the packet, it will forward the packet using the appropriate entry in its routing table as dictated by the direction string. For example, if the sender includes the direction string $\Delta = LMLR$, then the first hop along the path will forward the packet using its L entry, the second hop will forward the packet using the M entry and so on. By selecting a random direction string, the sender can easily choose a new path for its packets. It may decide to choose a new path when it fails to receive an acknowledgement from the recipient, or if it finds the latency of a particular path excessive. A node may also choose to use a secure traceroute protocol, such as the one proposed by Padmanabhan and Simon [23], to isolate the faulty node and potentially use the Honeybee technique described in Section 7.3 to eliminate it. As we show in Section 10.1, a randomly chosen direction vector will select a path that is largely disjoint from the first path, making it likely that the sender can successfully route around problem areas.

5.3 Choosing Nearby Edge Nodes

We can also use the redundancy of the recursive grouping algorithm to enhance the efficiency of our forwarding algorithm by having nodes select entries in their routing tables based on a distance metric. In other words, given the choice between two next-hop neighbors, a node will enter the one closest to the target group in its routing table entry.

As described in Section 4.3, when an internal node hears about a neighboring group via an edge node, it assigns the first neighbor from which it heard the announcement to its routing table entry for that group. However, as mentioned earlier, a group is likely to have several edge nodes bordering the same neighboring group. In that case, we can improve routing efficiency by having each internal node choose a neighbor that will route towards the closest edge node, rather than the edge node that it hears from first. If a node selects only the first edge node it hears from, then the edge node that announces the neighboring group first could become a hotspot in the network, since all of the internal nodes will use that one edge node to forward packets to the neighboring group. Furthermore, by forwarding to the closest edge node, we shorten path lengths, since we quickly transport the packet to a node that knows more about the next hop than any of the nodes in the current group. From a security standpoint, choosing edge nodes based on distance prevents a malicious edge node from rushing its announcement so as to become the sole link between the two groups.

To measure distance, we use a standard construction based on one-way hash chains [10]. We provide each sensor node with a collection of one-way hash chains (in addition to those used to generate challenges), $\Psi = \{(\nu,\rho) : h^{\lambda}(\rho) = \nu\}$, where $h^x(y)$ signifies applying the hash function x times starting with the value y, and

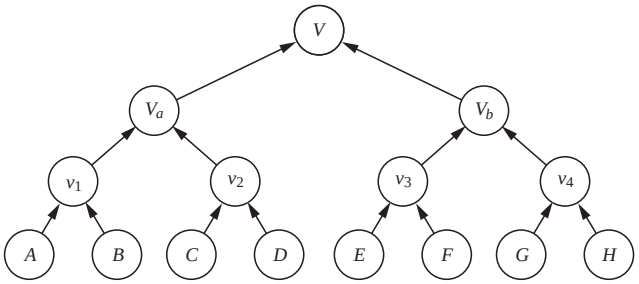


Figure 4: Hash Tree Construction. A hash tree is constructed by hashing the roots of two subtrees to form the root of a new subtree. In our protocol, the leaf values will be node IDs, and the internal nodes will represent group IDs created during the recursive grouping algorithm, as described in Section 4.

λ is the length of the hash chain. Each final value ν is signed by the network authority. When a node α determines its role as an edge node, it releases one of its hash chain pairs (ν, ρ) (and the signature $\{\nu, ID_\alpha\}_{K_{NA}^{-1}}$) to the edge nodes in the neighboring group. Those edge nodes announce the group's presence by forwarding the node's ID, along with the signature, ν, and $h(\rho)$ to their neighbors in the group. At each hop, a node verifies the signature on ν and forwards ν, the signature, and the result of applying the hash function once again. A node that receives a pair (ν, ρ'), calculates its distance by applying h to ρ' until it arrives at ν. If this requires k applications, then the node is at a distance of $d = \lambda - k$. The use of a one-way function prevents an adversary from artificially decreasing the distance to an edge node, since the hash chain can only be traversed in the forward direction.

6. BACKGROUND: HASH TREES

In this section, we review the construction of hash trees, with an emphasis on the two authentication properties used to secure our protocol: authentication of any leaf value given the root value, and authentication of a root value using a challenge-response protocol.

Hash Trees (a.k.a. Merkle hash trees [21]) provide an efficient mechanism for performing two authentication operations: *VerifyLeaf* and *VerifyTree*. *VerifyLeaf* authenticates a particular value θ among a sequence of values $A, B, \ldots$, based on a single authentic root value V. *VerifyTree* uses the leaves in the hash tree to verify the tree itself.

To construct the hash tree, we place the values $A, B, \ldots$ at the leaf nodes of a binary tree, as Figure 4 shows. (For simplicity we assume a balanced binary tree.) The derivation of a parent node m_p from its left and right child nodes m_l and m_r is

$$m_p = h(m_l \,||\, m_r) \tag{3}$$

where $||$ denotes concatenation and h is a cryptographic hash function providing weak collision resistance [20]. We compute the levels of the tree recursively from the leaf nodes up to the root node. Figure 4 shows this construction over the eight values $A, B, \ldots, H$, e.g., $v_1 = h(A||B)$, $V_a = h(v_1||v_2)$, and $V = h(V_a||V_b)$.

The *VerifyLeaf* operation can use the root value of the tree to authenticate any of the leaf nodes in the tree. To authenticate a leaf value θ the sender discloses both θ and all the sibling nodes of the internal nodes on the path from θ to the root node. The receiver can then use these nodes to recompute the values on the path up to the root, and if the recomputed root value matches the known root value, the value θ is guaranteed to be authentic. For example, to authenticate value C in Figure 4 (given the authentic root V), the values D, v_1 and V_b are required to verify the equality:

$$V = h\Big(h\big(v_1 \,||\, h(\, C \,||\, D\,)\big) \,||\, V_b \Big) \tag{4}$$

In contrast, the *VerifyTree* operation uses authenticated leaf nodes and a challenge-response protocol to probabilistically validate the hash tree construction. Suppose a challenger wishes to verify a responder's hash tree. The responder commits to the tree by sending the root value and the number of leaf values in the tree to the challenger (assuming the responder is committing to the tree shown in Figure 4, the responder would provide $(V, 8)$). The challenger identifies a random subset of leaf values (either by picking from a known set of leaf values or by generating a random path through the tree from the root down to a leaf) and asks the responder to prove that those values exist in the tree. For each leaf selected, the responder provides the authenticated leaf value along with the intermediate values leading to the root of the tree. In the example shown in Figure 4, if the challenger chooses C and E, then the responder will provide two chains of values: (D, v_1, V_b) and (F, v_4, V_a). The challenger authenticates the leaf values by recomputing the appropriate hashes and checking that they produce a consistent tree. In the example, the challenger would compute Equation 4 to verify C and then compute an analogous equation to verify E. If all of the leaf values are verified successfully, and if the intermediate hash values are all consistent and lead to the root value that the responder committed to originally, then the challenger accepts the hash tree as legitimate. This procedure provides a strong, though probabilistic, guarantee of the correctness of the hash tree. Przydatek, Song and Perrig use a similar approach for secure information aggregation [27].

7. DETECTION AND RECOVERY

To enhance the security of our routing protocol, we add additional measures to detect and recover from malicious behavior. We describe the Grouping Verification Tree (GVT) used to detect attempts to deviate from the recursive grouping algorithm. We also employ a duplicate detection scheme to prevent nodes from claiming multiple identities or attempting to group promiscuously. Finally, we introduce a Honeybee technique for removing malicious nodes from the network.

7.1 Detecting Grouping Deviations

To prevent an adversary from tampering with the recursive grouping algorithm, we use a Grouping Verification Tree (GVT) to detect deviations or inconsistencies. The GVT construction is based on the hash trees described in Section 6, and hence the GVTs provide us with similar authentication properties. At each stage of the grouping algorithm, a group can use its GVT to authenticate its group's size and ID to other nodes (and similarly verify the size and IDs of neighboring groups).

The GVT for a particular group can be thought of as a hash tree constructed over the nodes in the group. Each node's ID is a leaf value, the group's ID is the root value and the intermediate group IDs (from previous merges) are the internal nodes of the hash tree. For example, consider the hash tree shown in Figure 4. Viewing the hash tree as a GVT, the leaf values $(A, B, \ldots)$ represent the IDs for 8 sensor nodes, while the root value V is the ID for the entire group. When nodes A and B merged, they formed a group with an ID of v_1, and similarly, when they merged with group v_2 (containing nodes C and D), the combined group of four nodes had an ID V_a.

Below, we explain the GVT mechanism in greater detail. Then we show how two groups that are about to merge can verify each others' size and ID using the GVT. They can also use the GVT to

Algorithm 1 : GVT Verification During a Merge. Assuming groups G and G' are about to merge, the steps below illustrate how G can verify the GVT for G'. G' would perform an analogous procedure to verify G's GVT.

1: G' announces its ID and size $(ID_{G'}, |G'|)$ to G
2: Group G chooses one of its nodes as a challenger C
3: C selects challenge C_k and broadcasts it to nodes in G
4: Nodes in G verify C_k is a correct challenge and edge nodes forward C_k to G'
5: Based on C_k, group G' chooses a responder node
6: Responder sends its certificate and merge table to G
7: Nodes in G perform the *VerifyTree* operation to authenticate the GVT for G'

verify each other's neighbor list. The final GVT can authenticate any node's network address.

GVT Formation. During the recursive grouping algorithm, each node maintains a GVT for the group that it belongs to. The group ID is calculated such that it represents the root of the group's GVT, with the IDs of the nodes in the group at the leaves (note the similarities in the calculation of the intermediate hash tree values in Equation 3 and the calculation of the group IDs in Equation 1). Each node also has a merge table $\mathcal{M}$ that records the ID and size of each group it has merged with. In the example in Figure 4, node A will have three entries in its merge table: $\mathcal{M}[0] = (B,1)$, $\mathcal{M}[1] = (v_2,2)$, and $\mathcal{M}[2] = (V_b,4)$. These values are used for the authentication operations described below.

GVT Verification During a Merge. When two groups, G and G', decide to merge, they use the *VerifyTree* operation described in Section 6 to authenticate each other's GVTs. Since the GVTs incorporate both the group IDs and the group sizes, this implicitly authenticates the ID and size claimed by each group (hence preventing a malicious node from lying about its group's size in order to influence the grouping process). Algorithm 1 summarizes the important steps in the verification procedure.

To use the *VerifyTree* operation, group G will generate a challenge for its merge target G' that will randomly select a leaf node (i.e., an individual sensor in group G') in the GVT. The leaf node will respond to the challenge on behalf of G', by providing its own node ID (and accompanying certificate) and the intermediate values in the GVT.

To prevent a malicious node from influencing the choice of the random challenge, group G selects a challenger node C based on the group ID, ID_G. To determine the challenger, each node in the group computes $F(ID_G)$, where F is a cryptographic hash function with an even distribution over its range. The node in group G with a network address that is a prefix of $F(ID_G)$ will be the group's challenger. As we show in Section 8.2, the network addresses in a group are unique, and no network address is a prefix for any other network address, so the challenging node will be unique.[3]

The challenger node then broadcasts its current challenge value C_k to the group,[4] along with $\{k, \{C_0||ID_C\}_{K_{NA}^{-1}}\}$, the authentication information necessary to validate the challenge (discussed in Section 2). Each node in group G can verify that the challenge was generated appropriately by the correct node. The edge nodes bordering group G' will forward the challenge value C_k to it.

Group G' will use the challenge C_k to choose a responder in a manner similar to the selection of the challenger. Each node in the group will compute:

$$r = F(ID_G, |G|, ID_{G'}, |G'|, C_k) \quad (5)$$

The node in the group with a network addresses that is a prefix of r will be the responder for G'. The responder sends its node ID, certificate, and merge table to the challenging group G. Since the merge table contains all of the intermediate values of the GVT (i.e., the internal nodes of the hash tree) and the certificate serves to authenticate the responder's ID, each node in group G can use the *VerifyTree* operation to authenticate the intermediate values that lead to the current size and ID claimed by group G'. If verification fails, then the challenging group aborts the merge and begins a new round by selecting its next smallest neighbor.

Neighbor Verification. We can extend the challenge-response protocol to allow both groups to verify the neighbor lists provided. The merge target G' forwards group G's challenge C_k to each of its neighboring groups. These neighbors select a responder using a modified version of Equation 5, such that neighbor x calculates:

$$r = F(ID_G, ID_{G'}, ID_x, C_k) \quad (6)$$

The chosen responder sends its node ID, certificate, and merge table to G' which forwards the response to G. Thus, G can verify the existence of G''s neighbors and vice versa. Each edge node in the combined group γ will then know how many neighbors it should expect to hear from the next time the group wishes to initiate a merge by computing the smallest neighboring group.

Network Address Verification. Eventually, all of the nodes in the network will fall under a single GVT, and thus they will all know the root value V. Thus, we can use the *VerifyLeaf* operation to authenticate any node's network address. To authenticate its network address to another node, node A can provide its merge table, which contains the intermediate values in the GVT. This allows the verifier to recompute the necessary hashes and verify that the final result is V.

7.2 Duplicate Detection

We introduce one additional detection step to prevent a malicious node from claiming multiple IDs (e.g., by replaying legitimate certificates from other nodes) or trying to merge with several groups simultaneously (in order to obtain multiple network addresses). After the recursive grouping algorithm concludes, each node announces its node ID and its network address to its neighbors. We use these pairs of values to run a replication detection algorithm [24] that allows legitimate nodes to detect a node ID that claims multiple network addresses or vice versa. The malicious node can be revoked using evidence from the replication detection algorithm or via the Honeybee technique described below.

7.3 Eliminating Malicious Nodes

If a legitimate node detects malicious behavior using any of the techniques described above, it uses the Honeybee recovery mechanism. Essentially, the legitimate node broadcasts a packet implicating the malicious node, and the other legitimate nodes revoke both nodes involved. By revoking both nodes, we limit the potential damage of a slander attack (i.e., a subverted node that claims a legitimate node is malicious), since a malicious node can only revoke a single legitimate node before being revoked itself.

More specifically, to remove a malicious node from the network, a legitimate node initiates a full network flood of a special Honeybee packet, with the malicious node's ID, its own ID, and a signa-

[3]The challenger is also guaranteed to exist, since after each merge, some nodes will have network addresses that start with 0 and some that start with 1. Combined with the uniqueness result, this implies that we can always find a specific challenger with a network address that is a prefix of $F(ID_G)$.

[4]Where k is the number of times the challenger has merged.

ture.[5] When another node receives this packet, it will revoke the malicious node (i.e., cease to communicate with, or on behalf of, that node). Like its namesake, however, this technique also requires the detecting node to sacrifice itself. Nodes that receive the Honeybee packet revoke the initiator as well. Thus, a subverted node could use this technique to revoke a legitimate node, but only at the cost of revoking itself as well.

The Honeybee technique will successfully eliminate malicious nodes under two conditions: first, the number of malicious nodes in the network must be less than the number of legitimate nodes, and second, legitimate nodes must be capable of accurately identifying malicious nodes. The first condition is a reasonable assumption – if the majority of the nodes in the network have been compromised, then the network already has little chance of successful operation. The second condition is more stringent, but equally necessary. If a malicious node can convince a legitimate node that a second legitimate node is malicious, then the first legitimate node will sting the second, eliminating two legitimate nodes without disrupting the malicious node. Thus, we reserve the Honeybee technique for situations in which a legitimate node has absolute evidence for stinging another node (e.g., if it detects a replica after the recursive grouping algorithm concludes or if a neighbor's signature repeatedly fails to verify properly). Additional detection techniques could also make use of the Honeybee mechanism.

8. ANALYSIS

In this section, we demonstrate the correctness of our routing algorithm, consider the extent to which the recursive grouping algorithm generates uniform group sizes, and analyze the security of our scheme.

8.1 Correctness

To demonstrate the correctness of our routing protocol, we establish that it generates a unique network address for every node in the network. We also show that no node has an address that is a prefix of another node's address. Given these addresses, our forwarding algorithm behaves analogously to area routing described by Kleinrock and Kamoun [18]; their paper demonstrates the correctness of the forwarding algorithm.

We begin by proving the following theorem:

THEOREM 1. *The network address of every node in a group is unique within that group.*

Proof of Theorem 1: Initially, when two nodes merge, they each choose an address entry based on Equation 2. Since the node IDs are drawn from a total order, each will choose a different entry, and thus both will have a unique network address. Suppose two groups G and G' are about to merge, every node in G has a unique network address within G, and every node in G' has a unique network address within G'; this implies that naming conflicts can only arise between two nodes from different groups. Suppose without loss of generality that $ID_G < ID_{G'}$. Then, the most significant bit in the network address of each node in G will be 0, while the most significant bit in the network address of each node in G' will be 1, so network addresses from the two groups cannot conflict. ■

Since the recursive grouping algorithm terminates with all of the nodes in the network forming a single group, every node in the network must have a unique network address. Furthermore, when two groups merge, the bit added as the most significant digit of each node's network address ensures that no network address in G can be a prefix of a network address in G' (and vice versa).

8.2 Performance

In this section, we analyze the size and balance of the groups created by the recursive grouping algorithm, since these two factors influence the number of rounds required for the network to converge into a single group, the length of the resulting network addresses and the efficiency of packet forwarding. We demonstrate that most networks will converge within a logarithmic number of steps. While there are pathological cases that will take longer, it can be demonstrated that even those cases still converge quickly.

To demonstrate that the recursive grouping algorithm generates regular groups, we prove the following theorem that applies to most non-pathological deployments.

THEOREM 2. *If two merging groups share at least one neighboring group, the network will converge into a single group within a logarithmic number of iterations.*

Proof of Theorem 2: Suppose two groups X and Y of size $|X|$ and $|Y|$ decide merge to form a new group Z. Also, assume that X and Y initially share a neighbor K. In that case, we must have $|K| \geq \max(|X|,|Y|)$. To show this, we assume without loss of generality that $|X| < |Y|$ and $|K| < |Y|$. If this were the case, group X would have merged with K instead of Y. If the new group Z subsequently merges with K, then the resulting group will have size:

$$|Z'| \geq |Z| + \max(|X|,|Y|) \geq |Z| + \frac{|Z|}{2} \tag{7}$$

This implies that the group will grow by a factor of at least 1.5 during each merge. Thus, we expect each group will merge at most $\log_{1.5}(n)$ times before the entire network forms a single group. ■

This bound ensures that the size of the routing tables and network addresses will be logarithmic in the number of nodes in the network.

The worst-case scenario for our grouping protocol is a set of groups arranged in a "flower," in which a single group G has d neighbors, but none of its neighbors have any neighbors other than G. In that case, the central group G will merge with each of its neighbors, one at a time, creating d routing entries. However, in a dense, regularly deployed sensor network, this scenario is highly unlikely since a dense network implies that a single group cannot have a large number of neighbors each of whom has no other neighbors. In addition, regardless of the density, it can be shown that the size of the group must always grow by at least the size of the smaller group in the previous merge.

8.3 Security

In this section, we analyze the security properties of our scheme by illustrating that an adversary cannot (undetectably) subvert the recursive grouping algorithm by injecting, modifying or dropping packets. Given the secure network addresses and routing tables created by the recursive grouping algorithm, we rely on the resilient forwarding techniques described in Section 5.2 to react to malicious behavior at that level.

Since the recursive grouping algorithm proceeds deterministically based on the size and IDs of the groups, an adversary must undermine one or both of these attributes. We prevent (or detect) this using the secure neighbor discovery procedure to bootstrap group IDs and sizes (since all groups are of size 1). The use of GVTs prevents malicious tampering in the subsequent rounds.

[5] If the nodes are loosely synchronized, the node could use μ-TESLA [25] to authenticate its broadcast. Otherwise, it could also use public-key cryptography to sign the Honeybee message. Public-key signatures are typically expensive, but the legitimate node will be sacrificed anyway, as part of the Honeybee technique, so it may as well use its remaining battery to eliminate the intruder.

Since each node initially constitutes its own group, the group ID is the same as the node ID, and hence the secure neighbor discover procedure securely establishes the first set of group IDs. An external adversary cannot inject a manufactured ID, since it will be unable to produce a proper certificate to match it. Similarly, a compromised node will be unable to alter its ID without invalidating its signature. Every node must announce its ID during this period, or it will be ignored by its neighbors during the rest of the protocol, and attempts to replay IDs from other nodes will be revealed by the replication detection algorithm.

In subsequent rounds, the verification of GVTs during each merge prevents an adversary from modifying or injecting false information. During the GVT verification process, the choice of challenger and responder for each group is deterministic, so an adversary cannot influence these choices. Since the GVT covers both the group IDs and the group sizes, the *VerifyTree* operation will detect any attempt to modify either one. The challenge value itself can only be calculated by the challenger node, and hence the adversary cannot predict its value. There is a small possibility that a malicious node is chosen as either the challenger or responder. However, the challenger's challenge is verified by the other nodes in its group and the responder's response is verified by both groups. Hence, any attempt to fabricate or alter information about group IDs or sizes will be detected by the GVT verification procedure.

Since a malicious node cannot fabricate or modify legitimate messages, its only remaining strategy is to selectively drop (or fail to initiate) grouping messages. In a relatively dense network, most groups will have multiple shared edge nodes, as shown in Figure 3. In that case, if a malicious edge node fails to announce the neighboring group to its own group, the other edge node(s) will still provide the proper notification, and the malicious node will only succeed in removing itself from the routing tables of the internal nodes. If the malicious node is the only edge node for a group, then it can prevent its own group from learning about the neighboring group. However, this only serves to sever the link between the two groups, and thus they are not actually neighbors. If the malicious node does inform the neighboring group about its own group, it cannot persuade them to merge with its group without performing the GVT verification procedure, which will require assistance from its own group, hence revealing the presence of the neighboring group. Thus, selectively dropping or omitting messages will not undermine the recursive grouping algorithm.

9. ATTACKS AND DEFENSES

In this section, we consider possible attacks on routing protocols, and we show how our protocol defends against them.

9.1 Routing Attacks

Researchers have identified several severe routing protocol attacks [11, 15], which we summarize below.

Routing loop: An attacker injects malicious routing information that causes other nodes to form a routing loop. Packets injected into this loop (both by legitimate and malicious nodes) are then sent in a circle, wasting precious communication and battery resources. Generally, a routing loop attack is only considered successful if the loop does not include the attacker.

General Denial-of-Service (DoS) attacks: By injecting malicious information or altering legitimate routing setup messages, an attacker can prevent the routing protocol from functioning correctly. For example, an attacker can forge messages to convince legitimate nodes to route packets in away from the correct destination. Wood and Stankovic analyze general DoS attacks against sensor networks [32].

Sybil attack [5]: A malicious node creates multiple fake identities to perform attacks. In geographic routing protocols, fake identities can claim to be at multiple locations.

Slander and framing attacks: In systems that route based on reputation, a malicious node may attempt to slander legitimate nodes by accusing them of malicious behavior. In a subtler framing attack, an adversary causes a legitimate node to act (or appear to act) in a way that leads other legitimate nodes to decide it has been compromised.

Black hole attack: A malicious node advertises a short distance to all destinations, attracting traffic meant for those destinations. The attacker can selectively forward messages (although it may be difficult for them to leave the black hole).

Wormhole attack [12]: Two nodes use an out-of-band channel (e.g., a directional antenna) to forward traffic between themselves, enabling them to mount other attacks.

Replication attack [24]: An adversary may compromise a single legitimate node and insert copies throughout the network, increasing his presence in the network and thus allowing him to influence and subvert the network's performance.

(Selective) Suppression: A malicious node may decide to drop some or all of the packets that it receives, in an effort to disrupt routing setup. A malicious node may also drop packets during regular routing, but at that point, the attack should be considered an attack on the forwarding system, and not on routing.

Jamming: An adversary may jam the radios of legitimate nodes in the network to prevent them from receiving important routing messages.

9.2 Defending Against Specific Attacks

Since the recursive grouping algorithm functions entirely deterministically given the network topology, malicious nodes are prevented from manipulating the resulting routing information to introduce routing loops or routing-based denial-of-service attacks. The secure neighbor discovery portion of the algorithm (described in Section 4.2) prevents an adversary from introducing Sybil nodes into the network. The Honeybee technique prevents a malicious node from slandering a legitimate node, unless it is willing to sacrifice itself to eliminate the legitimate node. Furthermore, we reserve the use of the Honeybee technique for cases in which the legitimate node has proof of malicious behavior, thus preventing framing attacks. Since routes are not chosen based on advertised distances, we inherently prevent black hole attacks as well.

Several recent studies [26] show how to prevent and/or detect wormhole attacks in sensor networks, and most could be readily added to our protocol. Furthermore, our resilient routing techniques allow a legitimate node to route around a wormhole that drops too much traffic.

In general, the use of GVTs at each stage of the recursive grouping algorithm allows us to detect an adversary that interferes with routing setup. By running a replication detection algorithm after the recursive grouping algorithm concludes, we can detect and recover from malicious replicas. An adversary's attempts to modify the recursive grouping algorithm to attract additional traffic (e.g., by merging promiscuously with other groups) will be detected, since the malicious node will have multiple network addresses.

Finally, our resilient routing techniques allow senders to route around malicious nodes that suppress traffic and may also assist with defending against jamming attacks. During address setup, our use of resilient broadcasts for group coordination defends against DoS attacks and localized jamming. Previous research also suggests additional mechanisms to cope with jamming [15, 34].

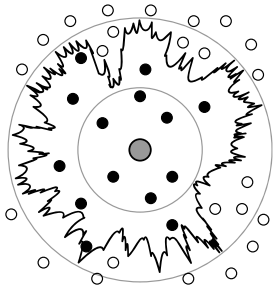


Figure 5: DOI Model. The inner and outer circles represent r_{min} and r_{max}, for the node at the center. The jagged line indicates the node's communication range, allowing communication with the solid black nodes.

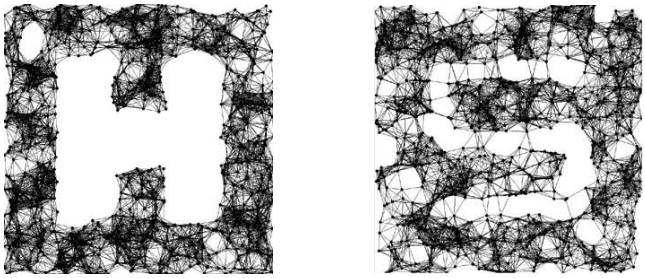


Figure 6: Sample Irregular Topologies. To generate an irregular topology, we define a void, and then deploy the nodes at random within the remaining space. Lines indicate neighbor connections.

10. SIMULATIONS

To evaluate and compare our scheme with other protocols, we developed a simulator and ran a number of experiments. Since, to our knowledge, no other routing protocol provides secure point-to point routing without geographic information, we simulated the Beacon-Vector Routing (BVR) protocol [6], as a point of comparison for routing performance. Unlike geographic routing protocols, BVR does not require additional hardware, nor does it impose an impractical amount of state at each node like other recently proposed protocols for routing without geographic information (e.g., NoGeo [28]). While BVR serves as a pertinent baseline for performance, we note that it assumes a trustworthy and cooperative environment. Security rarely comes for free, so we expect our costs to exceed those of BVR. In our experiments, we find that while our setup costs exceed those of BVR, we provide superior load distribution, particularly in networks with voids, which translates to a longer network lifetime. In many applications, the higher setup overhead is worth the gain in security and load distribution.

Setup. In our simulations, we deploy n sensor nodes at random within a square planar region (500x500 square units), where $n = 100$ or 500. We adjust the radio range of the nodes so that an average node will have approximately 10 neighbors. To make our simulations more realistic, we use the DOI (Degree of Irregularity) communication model described by He et al. [8] and depicted in Figure 5. The communication range of a node is modeled as a random walk around a disc, bounded by maximum range r_{max} and minimum range r_{min}, resulting in many unidirectional links. For our simulations, we choose $r_{min} = r_{max}/2$, and a DOI of 0.2.

In BVR, a set of R nodes elect themselves as beacons. All other nodes establish network addresses based on their distances from each beacon. Routing a message involves greedy forwarding to a node with lower minimum distance to the destination. If greedy forwarding fails, the routing algorithm resorts to "scoped flooding," in which a beacon node initiates a flood with a bounded radius to guarantee delivery. Based on the values suggested in the BVR paper, we set the number of beacons at $R = 10$ and the number of beacons used for routing at $K = 10$.

Experiments. Below, we describe the experiments we simulated. In all cases, we use the basic version of our forwarding protocol (i.e., without the resilient routing described in Section 5.2).

Routing Setup Overhead. First, we measure the number of packets sent per node during routing setup. For BVR, this includes the packets flooded through the network to allow the nodes to determine their distances from the beacon nodes. For our protocol, it includes all of the messages exchanged during the recursive grouping algorithm (e.g., merge requests, refusals, and status updates).

Path Stretch. After the routing infrastructure has been created, we consider routing between all possible ordered pairs of nodes (for a total of $n(n-1)$ pairs). For each pair, we evaluate the path stretch, i.e., the ratio of the number of physical hops required using either our scheme or BVR versus the optimal path computed using Dijkstra's shortest path algorithm.

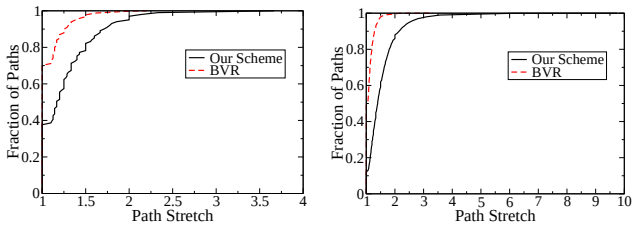
Load Distribution. Ideally, a routing protocol should evenly distribute the overhead of message forwarding across all of the nodes in the network, since hotspot nodes will quickly exhaust their battery power. To evaluate load distribution, we measure the number of packets each node must forward in order to route packets between every pair of nodes

Load Distribution with Voids. Dealing with irregular topologies is an important attribute for a general sensor network routing protocol, since real-world topologies often include obstacles and voids. Thus, we ran additional simulations to evaluate load distribution in the presence of large voids, such as the ones shown in Figure 6.

Path Diversity. Finally, we ran a separate set of experiments to quantify the performance of our resilient routing techniques (as discussed in Section 5.2). In these experiments, we first randomly select a pair of nodes. Then, we compute the intersection of the set of nodes on the default route between them and the set of nodes on a route produced by a randomly chosen direction string. We collect the average and maximum size of the intersection over 10 randomly chosen direction strings for 100 randomly chosen pairs of nodes.

10.1 Simulation Results

Routing Setup Overhead. As expected, BVR incurs less setup overhead than our protocol. With R beacon nodes, BVR only needs to flood the network R times so that nodes can establish their distance from each beacon. Our protocol requires coordination within the groups during the recursive grouping algorithm; as a result, with $n = 100$, we require each node to send or forward 139 packets on average (with a maximum of 199 for any one node), and for $n = 500$, we require an average of 252 packets per node with a maximum of 392. We also evaluate the communication required to perform the recursive grouping algorithm in a trustworthy environment (i.e., without the GVT and other security mechanisms). In this setting, the network of 100 nodes requires 83 packets on average (with a maximum of 111), and the 500 node network requires 132 packets on average with a maximum of 201. This indicates that the security mechanisms do add considerable overhead to the setup process. Future work will look at further optimizing setup efficiency. While our protocol still requires a higher setup overhead than BVR, we note that routing based on BVR's coordinates can fail to find a direct route to the destination, resulting in a scoped flood, whereas our protocol provides 100% delivery, even



(a) **100 Nodes** (b) **500 Nodes**

Figure 7: Path Stretch. These graphs show the path stretch plotted as a cumulative distribution function (CDF). Path stretch measures the ratio of the number of physical hops required by the routing algorithm versus the length of the ideal shortest path.

without our using the resilient routing techniques we described in Section 5.2.

Path Stretch. In Figure 7, we plot the results for our routing stretch simulations. In the network of 100 nodes, BVR uses 10% of the nodes as beacons, and thus has fairly minimal stretch. As the network grows larger, their stretch approaches that of our protocol. The stretch results do not include paths upon which BVR routing failed and was forced to revert to flooding. In the 100-node network, BVR routed 98% of the paths without flooding, and in the 500-node network it routed 92.6% of the paths without flooding. Our protocol routes successfully on 100% of the paths for both 100- and 500-node networks without flooding.

Load Distribution. As shown in Figures 8(a) and 8(b), our protocol distributes communication overhead more evenly across the nodes in the network, despite a larger path stretch, while BVR demonstrates a heavy tail effect. This is due to BVR messages that do not reach the destination and need to resort to flooding, thus imposing a heavier communication burden on the nodes near the beacons.

Load Distribution with Voids. In general, irregular topologies have little effect on the performance of our protocol – routing success remains at 100% and load remains balanced. BVR has considerably more trouble. Their success rate drops to 94.7% for 100 nodes and 91.5% for 500 nodes without flooding. As a result, BVR must flood between 5 and 10 percent of the time, making load distribution even worse than in the uniform deployments. Figures 8(c) and 8(d) illustrate this effect. Once again, the heavy tail indicates the presence of hotspots in the network.

Path Diversity. In our experiments to evaluate path disjointness, we found that for a network of 100 nodes, a path based on a randomly chosen direction string intersected with an average of 60% of the nodes on the default path. In a network with 500 nodes, the average intersection was 73%. This indicates that by simply varying the direction string in a packet, a legitimate node can significantly alter the composition of the path.

11. IMPLEMENTATION

To further evaluate the practical issues that arise on real motes, we have developed an implementation of our routing protocol (as presented in Sections 4 and 5) in the TinyOS environment and run it on our testbed of Telos motes. In the future, we plan to extend our implementation to include the detection techniques presented in Section 7. In addition to providing more realistic results, the resource constraints and wireless medium produced a number of new challenges.

11.1 Challenges

Reliable Broadcast. During the recursive routing algorithm, each edge node in a group G independently calculates the appropriate merge target G'. To ensure every edge node arrives at the same conclusion, they must all have up-to-date information. Thus, information about neighboring groups must be reliably broadcast to the entire group.

While reliable broadcast is a simple concept, it remains a difficult research problem in wireless sensor networks, and it created a significant challenge for our implementation. To further complicate the issue, we must perform reliable broadcasts during routing establishment, when nodes have limited information about the actual topology.

We achieve reliable broadcasts using the nodes' network addresses. As shown earlier, at each stage of the grouping algorithm, the network addresses are unique within each group.

Thus, at any stage of the recursive grouping algorithm, a node can use its partially established routing table to send a reliable broadcast to its group while maintaining low per node overhead. For example, node A can start the broadcast by looking through its neighbor table and choosing two nodes: one node B with a network address matching the prefix 0*, and another node C with a network address matching the prefix 1*. Node B, upon receiving this packet, chooses two new recipients to propagate the broadcast by resolving 1 additional address bit 00* and 01*. This protocol terminates when the length of the network address exceeds r, or if no matching addresses exist.

Asynchronicity. Motes wake up and execute asynchronously, which makes the implementation and debugging process more complicated than in simulations. Moreover, because of radio contention, it is actually undesirable to allow motes to progress synchronously. To address these issues, we implement the recursive grouping algorithm as an event-driven finite state machine, with radio messages and timeouts triggering state transitions.

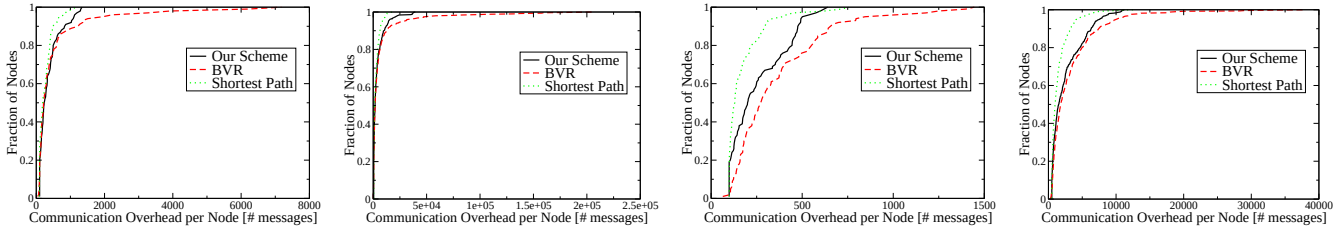
Asymmetric Links. The protocol design assumes bidirectional links between nodes; however, in real deployments, asymmetric links are endemic. To resolve this issue, we use the received signal strength indication (RSSI) to discover and agree on neighbors. During the secure neighbor discovery protocol, each node keeps RSSI values for every node it hears from. Then, it sends neighbor invitations to the k nodes with the strongest RSSI values. A node α adds another node β as a neighbor if it receives an invitation from β, or if its own outgoing invitation is acknowledged by β.

11.2 Results

We implemented and ran the routing protocol on our sensor network testbed. The testbed consists of 16 Telos motes distributed across a floor of our building. Each mote was connected to a USB hub to supply power and to facilitate debugging. We also set up several motes as base stations to monitor network traffic. We plan to deploy our protocol on a larger network upon completion of our testbed.

After routing information has been established, we test the routing protocol by trying to route between each pair of motes. Each mote sends 5 packets to all possible network addresses. The other motes forward the packets using their routing tables. When a packet reaches the intended recipient, a reply is generated and sent back to the original sender. As Table 1 shows, our protocol routes successfully 100% of the time on a deployment of 16 motes.

We also measured the memory overhead used by each mote during routing establishment. As our results show in Table 1, our protocol achieves very low memory usage. The code size is a constant overhead of 21KB. More importantly, our protocol scales well,



(a) **100 Nodes, Uniform** (b) **500 Nodes, Uniform** (c) **100 Nodes, Irregular** (d) **500 Nodes, Irregular**

Figure 8: Load Distribution. These graphs show the distribution of communication overhead required to route between all possible pairs of nodes in the network. We plot the results as a CDF to show the distribution of the load in the network. A protocol with a perfectly even distribution would appear as a vertical line.

Number of Merges Per Node	4
Code Size	21 KB
Data Size	50 bytes
Avg. Num of Packets Sent Per Node	101
Success Rate	100%

Table 1: Implementation Results. This table summarizes the overhead required to setup the routing information for 16 motes. It also notes the routing success rate.

since the dynamic data does not grow linearly with the network size. Instead, each node only needs to keep two tables, a routing table and a neighbor table. The routing table adds an entry for each merge, and hence grows logarithmically in the number of nodes in the network. The size of the neighborhood table is constant, given the network degree d. In general, if the setup requires m merges, the memory overhead of our implementation is $10d + 5m$ bytes.

12. RELATED WORK

In this section, we summarize related research on routing in trustworthy environments and discuss other work on secure sensor network routing.

Routing in Trustworthy Environments. The idea of creating a hierarchy of regions has been previously proposed by several researchers [18, 30, 31]. However, we are not aware of the use of such hierarchies for either routing in sensor networks or secure routing in general.

Several approaches to provide efficient routing for node-to-node communication in sensor networks have been proposed [1, 7, 9, 13, 17, 22, 28]. However, they all assume a trusted environment without considering security.

Karlof, Li, and Polastre propose the ARRIVE protocol, a protocol resilient to failures, but not against attacks [14].

Secure Sensor Network Routing. Karlof and Wagner describe attacks on standard (unsecured) sensor network routing protocols and propose some generic countermeasures, without proposing a complete protocol [15].

Deng, Han, and Mishra propose INSENS [3, 4]. INSENS provides routing between nodes and base stations, but not between arbitrary sensor nodes (except by relaying through the base station). In contrast, we design a general secure routing protocol that can relay messages between arbitrary nodes.

SIGF, designed by Wood et al., achieves secure routing properties based on the assumption that nodes know their own geographic locations [33]. Their scheme prevents spoofing using local keys, and they add resiliency through nondeterministic selection of forwarding nodes. Our scheme imposes more complexity but does not require geographic information.

Some of the security mechanisms we leverage in this paper have also been used by other researchers. Routing around an area of poor connectivity and sending messages over multiple paths have been proposed for sensor networks in various contexts [4, 14, 32, 34].

13. FUTURE WORK AND CONCLUSION

To secure networking protocols, researchers often add security mechanisms to existing protocols that were designed for benign environments. In the case of routing protocols, this has worked well in the context of Internet and ad hoc network routing protocols. Unfortunately, in the context of highly resource-constrained sensor networks, we found that securing existing protocols introduced either an unacceptable level of complexity or an excessive performance penalty. For these reasons, we decided to design a secure sensor network routing protocol from a clean slate.

By leveraging all three approaches to design secure routing protocols (prevention, detection and recovery, and resilience), we obtain a secure routing protocol that is highly robust to attack. We demonstrate the security and performance of our protocol through theoretical analysis, simulation, and implementation. The performance overhead is reasonable given the security achieved.

In the future, we plan to analyze the performance of our protocol under attack, to investigate techniques to improve its efficiency, and to develop our implementation into a more robust platform.

14. ACKNOWLEDGEMENTS

The authors gratefully thank Yongdae Kim and Abhishek Jain for helpful discussions, Haowen Chan for his feedback and his suggestion of the Honeybee technique, Diana Seymour for her excellent editing assistance, and the anonymous reviewers for their comments and suggestions.

15. REFERENCES

[1] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, 1999.

[2] H. Chan and A. Perrig. PIKE: Peer intermediaries for key establishment in sensor networks. In *IEEE Infocom*, Mar. 2005.

[3] J. Deng, R. Han, and S. Mishra. A performance evaluation of intrusion-tolerant routing in wireless sensor networks. In *IEEE Workshop on Information Processing in Sensor Networks (IPSN)*, pages 349–364, Apr. 2003.

[4] J. Deng, R. Han, and S. Mishra. Intrusion-tolerant routing for wireless sensor networks. *Elsevier Journal on Computer Communications, Special Issue on Dependable Wireless Sensor Networks*, 2005.

[5] J. R. Douceur. The Sybil attack. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, Mar. 2002.

[6] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon-Vector Routing: Scalable point-to-point routing in wireless sensor networks. In *Proceedings of USENIX/ACM Symposium on Networked System Design and Implementation (NSDI)*, May 2005.

[7] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin. Highly resilient, energy-efficient multipath routing in wireless sensor networks. *Mobile Computing and Communication Review*, 5(4):10–24, 2002.

[8] T. He, C. Huang, B. Blum, J. A. Stankovic, and T. Abdelzaher. Range-free localization schemes for large scale sensor networks. In *Proceedings of ACM Mobicom*, pages 81–95, Sept. 2003.

[9] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Hawaii Int'l Conference on Systems Sciences*, 2000.

[10] Y.-C. Hu, D. B. Johnson, and A. Perrig. SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks. *Ad Hoc Networks*, 1(1):175–192, 2003.

[11] Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *Proceedings of ACM Mobicom*, Sept. 2002.

[12] Y.-C. Hu, A. Perrig, and D. B. Johnson. Packet leashes: A defense against wormhole attacks in wireless ad hoc networks. In *Proceedings of IEEE INFOCOM*, April 2003.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of ACM Mobicom*, pages 56–67, 2000.

[14] C. Karlof, Y. Li, and J. Polastre. ARRIVE: Algorithm for robust routing in volatile environments. Technical Report UCB/CSD-03-1233, University of California at Berkeley, May 2002.

[15] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*, 1(2–3):293–315, Sept. 2003.

[16] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 18(4):582–592, Apr. 2000.

[17] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Geographic routing made practical. In *Proceedings of USENIX/ACM Symposium on Networked System Design and Implementation (NSDI)*, May 2005.

[18] L. Kleinrock and F. Kamoun. Hierarchical routing for large networks: Performance evaluation and optimization. *Computer Networks*, 1:155–174, 1977.

[19] S. Marti, T. Giuli, K. Lai, and M. Baker. Mitigating routing misbehaviour in mobile ad hoc networks. In *Proceedings of ACM Mobicom*, pages 255–265, Aug. 2000.

[20] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[21] R. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, 1980.

[22] J. Newsome and D. Song. GEM: Graph embedding for sensor motes. In *Proceedings of ACM SenSys*, Oct. 2003.

[23] V. N. Padmanabhan and D. R. Simon. Secure traceroute to detect faulty or malicious routing. *SIGCOMM Computer Communication Review (CCR)*, 33(1):77–82, 2003.

[24] B. Parno, A. Perrig, and V. Gligor. Distributed detection of node replication attacks in sensor networks. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[25] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, Sept. 2002.

[26] R. Poovendran and L. Lazos. A graph theoretic framework for preventing the wormhole attack in wireless ad hoc networks. *ACM Journal on Wireless Networks (WINET)*, to appear.

[27] B. Przydatek, D. Song, and A. Perrig. SIA: Secure information aggregation in sensor networks. In *ACM SenSys*, Nov 2003.

[28] A. Rao, C. Papadimitriou, S. Ratnasamy, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proceedings of ACM MobiCom*, Sept. 2003.

[29] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *Proceedings of ACM WSNA*, Sept. 2002.

[30] D. G. Thaler and C. V. Ravishankar. Distributed top-down hierarchy construction. In *Proceedings of INFOCOM*, 1998.

[31] P. F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *Proceedings of SIGCOMM*, 1988.

[32] A. Wood and J. Stankovic. Denial of service in sensor networks. *IEEE Computer*, pages 54–62, Oct. 2002.

[33] A. D. Wood, L. Fang, J. A. Stankovic, and T. He. SIGF: A family of configurable, secure routing protocols for wireless sensor networks. In *Proceedings of ACM SASN*, Oct. 2006.

[34] A. D. Wood, J. A. Stankovic, and S. H. Son. JAM: A jammed-area mapping service for sensor networks. In *Proceedings of Real-Time Systems Symposium (RTSS)*, 2003.

[35] M. G. Zapata. Secure Ad hoc On-Demand Distance Vector (SAODV) routing. Internet draft, Internet Engineering Task Force, Aug. 2001.