LIDeA: A Distributed Lightweight Intrusion Detection Architecture for Sensor Networks

Ioannis Krontiris
Athens Information Technology
19.5 km Markopoulo Ave., Athens, Greece
ikro@ait.edu.gr

Thanassis Giannetsos
Athens Information Technology
19.5 km Markopoulo Ave., Athens, Greece
agia@ait.edu.gr

Tassos Dimitriou
Athens Information Technology
19.5 km Markopoulo Ave., Athens, Greece
tdim@ait.edu.gr

ABSTRACT

Wireless sensor networks are vulnerable to adversaries as they are frequently deployed in open and unattended environments. Preventive mechanisms can be applied to protect them from an assortment of attacks. However, more sophisticated methods, like intrusion detection systems, are needed to achieve a more autonomic and complete defense mechanism, even against attacks that have not been anticipated in advance. In this paper, we present a lightweight intrusion detection system, called LIDeA, designed for wireless sensor networks. LIDeA is based on a distributed architecture, in which nodes overhear their neighboring nodes and collaborate with each other in order to successfully detect an intrusion. We show how such a system can be implemented in TinyOS, which components and interfaces are needed, and what is the resulting overhead imposed.

Categories and Subject Descriptors

C.2.0 [**Computer - Communication Networks**]: General—*Security and protection*; C.2.4 [**Computer - Communication Networks**]: Distributed Systems

General Terms

Algorithms, Security, Theory, Experimentation

Keywords

Sensor Networks, Intrusion Detection, TinyOS

1. INTRODUCTION

The pervasive interconnection of autonomous sensor devices has given birth to a broad class of exiting new applications in several areas of our lives, including environment and habitat monitoring, healthcare applications, home automation, and traffic control. At the same time, however, their unattended nature and the limited resources of their nodes have created equal number of vulnerabilities that attackers can exploit in order to gain access in the network and the information transferred within.

There are several classical security methodologies so far that focus on trying to *prevent* these intrusions. However, it is impossible, or even infeasible, to guarantee perfect prevention. Not all types of attacks are known, and new ones appear constantly. As a result, attackers can always find security holes to exploit in order to gain access in the sensor network. These intrusions will go unnoticed and they will likely lead to failures in the normal operation of the network, as Figure 1(a) suggests.

The last resort is intrusion detection, which can act as a second line of defense: it can *detect* third party break-in attempts, even if this particular attack has not been experienced before. If the intruder is detected soon enough, one can take appropriate measures before any damage is done or any data is compromised (Figure 1(b)). An effective IDS can also help us design better prevention mechanisms, by collecting information about intrusion techniques and attack patterns.

Any part of the sensor network can be a possible point of intrusion, since all nodes act as routers of information and they can be easily manipulated or subverted by an attacker. Therefore, an IDS architecture for wireless sensor networks has to be decentralized. In this paper, we present such an approach to organizing autonomous but cooperative IDS agents. Our approach organizes the cooperation of the agents according to the distributed nature of the events involved in the attacks, and, as a result, an agent needs to send information to other agents only when this information is necessary to detect the attack. The coordination mechanism arranges the message passing between the agents in such a way so that the distributed detection is equivalent to having all events processed in a central place.

In this work, we describe an IDS architecture based on this approach and implement an experimental intrusion detection system called LIDeA (Lightweight Intrusion Detection Architecture). But most importantly, this is the first work to present such implementation which is at the same time both realistic and lightweight enough to run on computationally and memory restricted devices such as the nodes of a sensor network.

2. RELATED WORK

Loo *et al.* [12] and Bhuse and Gupta [3] describe two IDSs for routing attacks in sensor networks. Both papers assume

SecureComm '08, September 22 - 25, 2008, Istanbul, Turkey

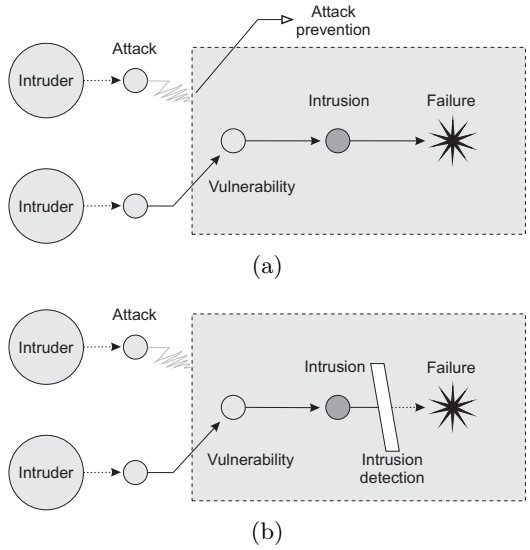

Figure 1: Intrusion sequence. (a) Attackers may exploit a vulnerability and intrude into the network, causing a failure. (b) Intrusion detection functions as a second line of defense.

that routing protocols for ad hoc networks can also be applied to WSNs: Loo *et al.* [12] assume the AODV (Ad hoc On-Demand Distance Vector) protocol while Bhuse and Gupta [3] use the DSDV and DSR protocols. Then, specific characteristics of these protocols are used like "number of route requests received" to detect intruders. However, to the best of our knowledge, these routing protocols are not attractive for sensor networks and they have not been applied to any implementation that we are aware of.

Anomaly detection in sensor networks has been studied in [5] and [14]. Nodes listen to messages in their radio range and store certain message fields that might be useful to the rule application phase. Each node has a fixed-size buffer to store the packets received. In [5] the authors focus on detecting some attacks, like message delay, repetition, data alteration, blackhole and selective forwarding. It is concluded that the buffer size to store the monitored messages is an important factor that greatly affects the false positives number. In [14] the authors monitor the packets arrival time and received power. If its power is not within certain limits, the packet is characterized anomalous. An intrusion alert is raised if the rate at which anomalous packets are detected over the overall rate at which packets are received is above a given threshold.

Another problem of intrusion detection systems for WSN is where the IDS agents should be placed. In [1], the authors argue that detection should be based only on the analysis of packets that pass through a node and they determine which nodes should be loaded with an IDS agent based on the concepts of dominating set and minimum cut set and on the requirement that the nodes running the IDS module should be tamper resistant. On the other hand, Roman et al. [17] propose an IDS architecture where all nodes are loaded with an IDS agent, and they promiscuously monitor the traffic. This agent is divided into two parts: local agents and global agents. Local agents are active in every node and are responsible for monitoring and analyzing only local sources of information. Global agents are active at only a subset of nodes. They are in charge of analyzing packets flowing in their immediate neighborhood.

Our work is different from the above approaches in the sense that we try to generalize the problem of intrusion detection for sensor networks and build an architecture that tolerates the presence of other compromised nodes that may exist and collaborate with the attacking node in order to hinder the detection process. In our model, *all nodes are loaded with the same IDS agent and they dynamically become activated around the attacking node and collaborate in order to isolate it from the network*. We do not concentrate on how to detect specific attacks, although we provide a use case and the necessary modules to describe rule patterns for defending against various attacks. We focus on the distributed computing nature of the architecture in order to show that with collaborative processing an IDS system can become lightweight enough to be realistic for sensor networks.

3. SYSTEM MODEL

We consider an asynchronous multihop wireless sensor network. Each node of the network has a single wireless transceiver through which it can communicate with the other nodes within its communication range. We do not assume a unit-disk graph model for the network. Instead, we consider a realistic representation for the communication model, where the range can be affected by various reasons and change from one transmission to the next. The nodes themselves are assumed to be static, as it is the case for many sensor network paradigms.

We also consider unreliable links and unpredictable delays for the wireless links. When a node transmits a packet, it does not know which nodes successfully received the message, since the MAC layer of the receivers does not send any acknowledgments or requests for retransmissions. A node may miss to receive a message, either because a collision occurs or because its radio is not available at the time of the transmission.

4. INTRUSION DETECTION IN WSN

Intrusion detection not only means to detect that a node has been attacked, it also includes identifying the source of an attack. In our case, the cooperative intrusion detection process is triggered by an attack and the subsequent alerts received by the neighboring sensors. The process ends by having the participating sensors jointly *expose* the source.

More formally, the task of intrusion detection (ID) can be defined as follows: Find an algorithm that satisfies the following properties:

- If an honest node s exposes a node t, then t is the source of the attack.
- If the attacker attacks, then at most after some time τ all honest nodes participating in the detection process expose some node.

Our approach for solving this problem is to have sensor nodes around the source of the attack exchange a list of suspects, resulting from their partial view of the network, and apply a voting scheme to conclude to which node they are going to expose. This protocol should be able to tolerate the presence of the attacker and its collaborator nodes,

which might try to hinder its proper operation and successful outcome.

This problem of reaching consensus in a network of n nodes among which t nodes may behave faulty is well known as the Byzantine Agreement Problem, first introduced by Pease, Shostak and Lamport [15] in 1980. According to this problem, there is a set of processors some of which may be faulty. Each nonfaulty processor has a private value that must be communicated to the rest. The problem is to devise an algorithm that will allow each nonfaulty processor to compute a vector of values corresponding to each of the n processors, such that

1. the nonfaulty processors compute exactly the same vector;
2. the element of this vector corresponding to a given nonfaulty processor is the private value of that processor.

After this *interactive consistency* has been achieved, each nonfaulty processor can apply a function to the vector, and since this vector is the same in all of them, an exact agreement is necessarily reached.

The analogy to the problem we define in this work is clear. In our case the processors are the sensor nodes, and the private values are the lists of suspected nodes. If the nodes can exchange messages and achieve interactive consistency despite the presence of other compromised nodes, then they can all apply a simple function, in this case a majority rule, and conclude to the same result.

However, the system model in our case is different than the one in the original Byzantine Agreement Problem, due to the nature of sensor networks. In particular, the characteristics of sensor networks that change the conditions of the Byzantine Agreement Problem are the following:

1. Communication Medium: In the Byzantine Agreement Problem, it is assumed that each processor communicates only by means of two-party messages. On the other hand, in sensor networks the nodes use a broadcast channel, meaning that every node that is within range of another node can overhear its messages.
2. Communication Range: In a sensor network, nodes that need to reach consensus (in our case, the nodes around the attacker) are not necessarily within range of each other. Therefore some sort of forwarding has to be used for a message to reach all the participatory nodes.

Moreover, as we mentioned in Section 3, sensor nodes are asynchronous and messages are not guaranteed to reach their destinations. It is well known that solving consensus deterministically requires some synchrony assumptions [6]. However, some papers have considered the consensus problem in wireless ad hoc networks, by using two main ways of circumventing the impossibility result; failure detectors [20, 4] and randomization [18]. These solutions are not attractive for sensor networks, since the former pose a hierarchical model, where some nodes need to act as "clusterheads" and the latter result in a large number of rounds with relatively high message overhead per round.

Having said this, let us note that this paper does not propose a new consensus algorithm nor a new model for solving consensus. We accept the possibility of message losses, which may result in inconsistencies in the vectors that the nodes compute. However, we compensate this by exploring the possibilities of redundant paths, message overhearing and a simple advertise-request scheme. In this way, we manage to keep the protocol lightweight enough for highly resource constraint networks and at the same time achieve our primary goal of distributed intrusion detection.

5. THREAT MODEL AND ASSUMPTIONS

We assume that an attacker can capture a number of nodes to launch an attack. We model this by allowing these nodes to behave in an arbitrary manner (Byzantine failure). We distinguish among compromised nodes one single node which is the *source* of the attack, i.e., this node is the first to behave in a faulty way. All non-source faulty nodes are called *collaborators*. We assume that collaborators are neighbors of the source of an attack and that they are less than its honest neighbors. The attacker can follow the protocol for a certain period of time and therefore behave in a way which cannot be detected. However, at some point in time the attacker must deviate from the protocol in some faulty node to launch an attack. At this point in time, we say that the attacker attacks.

While an adversary can completely take over nodes and extract their cryptographic keys, we assume that such an adversary cannot "outnumber" legitimate nodes by replicating captured nodes or introducing new ones in sufficiently many parts of the network. This assumption is needed because an IDS for WSNs should exploit the massive parallelism in such a network to detect intrusion attempts. In the sections that follow, we will see that as long as this assumption holds, the proposed architecture can be used to identify the attacker.

We also make the implicit assumption that the time required for the completion of the initialization phase of our protocol (Sections 7.2 and 7.3) is smaller than the time needed by an adversary to compromise a sensor node during deployment. Therefore, the initialization phase runs uninterrupted by malicious nodes. We also assume that the topology of the network cannot change during this phase. After that and throughout the lifetime of the network, nodes may leave the network, possibly because of energy depletion, or new nodes may be deployed. In the latter case we assume that there is a secure node addition protocol that is followed to prevent an attacker from introducing her own nodes.

6. ARCHITECTURAL OPTIONS

An IDS for sensor networks should be network-based, in the sense that raw network packets should be used as the audit source. A popular technique in sensor networks, and the one that we are going to follow for our system, is the *watchdog* approach [13]. Each packet transmitted in the network is not only received by the sender and the receiver, but also from a set of neighboring nodes within the sender's radio range. Normally these nodes would discard the packet, since they are not the intended receivers, but for intrusion detection this can be used as a valuable audit source. Hence, a node can activate its IDS agent and monitor the packets sent by its neighbors, by overhearing them. Since any node can act as a router and traffic is usually distributed for load balancing purposes, packet monitoring should take place in several nodes of the network.

Usually, when the activities involved in an attack fall be-

yond the scope of one IDS component, distributed IDS systems require that the audit data collected from different places must be forwarded to a central location for analysis. In sensor networks such a location could be the base station. However, it is not a wise choice to make, given the big communication overhead involved. This means that some sort of aggregation must take place locally, at the area of the attack, either at a specific node in the network or in a distributed fashion.

In the first case, the intrusion related information from different locations can be collected by a node (e.g. cluster head) and correlated together to make the final decision on the intrusion. The rest of the nodes do not participate in this decision. In such architectures, the decision-making nodes can attract the interest of an attacker, since their elimination would leave the network undefended. Furthermore, they restrict computation-intensive analysis of overall network security state to a few key nodes. Their special mission of processing the information from other nodes and deciding on intrusion attempts results in an extra processing overhead, which may quickly lead to their energy exhaustion, unless different nodes are dynamically elected periodically. Therefore, the second case, where audit data are aggregated in a distributed fashion is more appropriate for sensor networks.

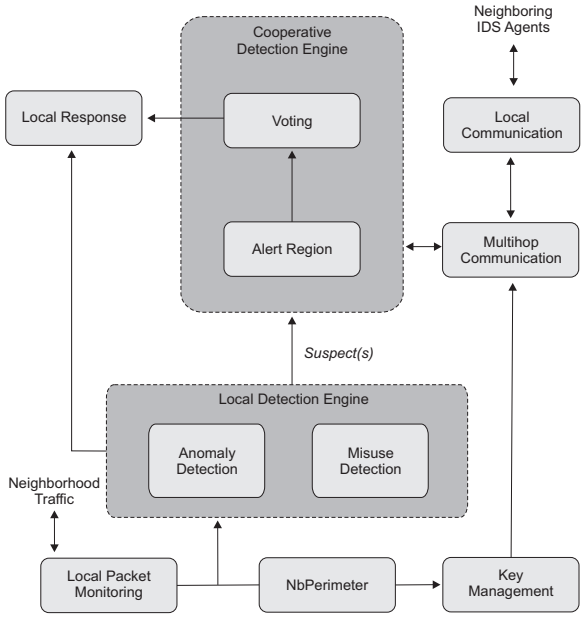

Figure 2: The IDS Architecture.

7. LIDEA ARCHITECTURE

The proposed IDS is based on a distributed intelligent agent-based system. The agents that are hosted by the nodes are capable of sharing their partial views, agree on the identity of the source and expose it. By distributing the agents throughout the network and have them collaborate, we make the system scalable and adaptive. When a malicious node is found, an alarm message is broadcasted to the network. Each node then makes a final decision based on the detection reports from other nodes. To avoid drastic flooding over the network caused by broadcasting local detection results, the alarm messages are restricted to a region formed only by the alerted nodes.

In its configuration, the system model does not include timing assumptions and is characterized by communication between 1-hop and 2-hop neighbors and the use of modern cryptography. There are no *a priori* trusted nodes, or any reputation system, as that would raise considerably the energy requirements. Instead, the system allows the arbitrary behavior of the nodes: a node may behave normally with respect to routing in order to avoid being detected by the IDS, but it can expose a malicious behavior to obstruct the successful detection of another intruder node. We call such nodes *faulty*. The IDS system is based on the power of the *majority* to protect itself from these misbehaving nodes.

We build the architecture of the IDS agent based on the conceptual modules shown in Figure 2. Each module is responsible for a specific function, which we describe in the sections below. The IDS agents are *identical in each node* and they can broadcast messages for agents residing in neighboring nodes.

7.1 Local Packet Monitoring Module

This module gathers audit data to be provided to the local detection module. Audit data in a sensor networks IDS system can be the communication activities within its radio range. This data can be collected by listening *promiscuously* to neighboring nodes' transmissions. By promiscuously we mean that since another node is within range, the data collection module can overhear communications originating from that node.

7.2 NbPerimeter Module

This module is responsible for maintaining consistent information about 1-hop and 2-hop neighbors of the nodes. Information about 2-hop neighbors is needed because, as we will see, the detection process involves the communication of the nodes which are neighbors of the (yet unknown) attacker, but they might be 2-hops away from each other. During an initialization phase that takes place immediately after the deployment of the network, the `NbPerimeter` module broadcasts the node ID and the IDs of the node's immediate neighbors within a packet that has a TTL field equal to 1, meaning that each packet will be forwarded just once by the sender's 1-hop neighbors. The discovered neighborhood information is stored in a table, which we call the 2-hops neighborhood table.

7.3 Key Management Module

After the deployment of the sensor network, the `KeyManagement` module of the node generates a one-way key chain of length n, using a pre-assigned unique secret key K_n. A one-way key chain [11] $(K_0, K_1, \ldots, K_{n-1}, K_n)$ is an ordered list of cryptographic keys generated by successively applying a one-way hash function F to the key seed K_n, such as $K_j = F(K_{j+1})$, for $j = n-1 \ldots 0$. Therefore, any key K_j is a commitment to all subsequent keys K_i, $i > j$. In our implementation, SHA-1 hashing is used for the production of the key chain. As the last step in the initialization phase, the `KeyManagement` module in each node announces the resulted K_0 to all of its 1-hop and 2-hop neighbors.

The `KeyManagement` module also stores the corresponding information for the neighboring nodes (up to 2-hops), i.e. the node IDs and their keys. This information needs to re-

main consistent and up-to-date during the lifetime of the network. So, the `KeyManagement` module updates the corresponding key every time a node publishes a new one from its key chain. But, also, the topology can change, as nodes may be removed or added, and for that reason, the `KeyManagement` module is linked with the `NbPerimeter` module and is informed for the new or deleted nodes.

7.4 Local Detection Engine

This module collects the audit data and analyzes it according to some given rules. A set of rules is provided for each attack, and whenever one or more rules are satisfied, a local alert is produced by the module. Whether a rule is satisfied or not does not just depend on information from the intercepted packets, but also on information from the 2-hop neighborhood table or information from past observed behavior.

As depicted in Figure 2, the `LocalDetectionEngine` incorporates both classes of intrusion detection techniques, i.e., *misuse detection* and *anomaly detection*. In misuse detection [8], the observed behavior is compared with known attack patterns that may pose a security threat and the IDS tries to recognize any "bad" behavior according to these patterns. For example, if a node receives a packet sent by a non-neighboring node, this clearly stands as an attack behavior.

However, it is not possible to cover all classes of attacks with this technique [2]. That's why some rules may be expressed according to the anomaly detection technique. Anomaly detection [9] first describes what constitutes "normal" behavior and then flags as intrusion attempts any activities varying from this behavior by a statistically significant amount. The normal behavior can be based on manually defined specifications that describe what a correct operation is. For example, a small packet drop rate is an expected phenomenon due to the wireless medium of communication. However, if that rate rises above a threshold, it could be an indication of (say) a selective forwarding attack.

The `LocalDetectionEngine` of a node s outputs an alert. This alert can contain one of two things: either the node ID of the attacker or a list of *suspected nodes*. In the first case, the node detecting the attack was able to identify the source (e.g. a node dropping packets), so it directs the alert to the `LocalResponseModule` for immediate measures. In the second case, it simply outputs some set $Suspect(s)$ of possible attacking nodes. $Suspect(s)$ will contain a subset of neighbors or may even be equal to the whole neighborhood of s. In any case, it cannot contain any non-neighboring node, since node s could not have observed an attack outside its radio range. By communicating its list of suspected nodes to the other nodes and collaborating with them is what can lead to recognizing the attacker's identity.

7.5 Alert Region Module

This module is activated only in the case where the `LocalDetectionEngine` module was inconclusive on the identity of the attacker and a suspects list was produced. In this case we call the node an *alerted node*. The set of alerted nodes define an *alert region*. Since not every node that belongs to the alert region has to be within communication range of each other, we need to define a communication abstraction intended to provide "connectivity" between them, or else a "neighborhood relationship". The `AlertRegion` module is responsible of exactly that: to let each alerted node find out about each other, so that they can form a group and start communicating.

The construction of the alert region is dynamically formed at the time of the attack and proceeds in iterations. Each alerted node s broadcasts a short message, which we call the *alert message* $m_a(s)$, in order to include itself in the alert region. The payload of the message contains the node ID of s. This message must reach all other alerted nodes. However, since not all of them are within 1-hop distance from each other, all messages $m_a()$ should be forwarded by intermediate nodes in order to reach 2-hop away alerted nodes. The code is given in Algorithm 1.

Algorithm 1: The *Alert Region* Algorithm

```
Data: Node ID s being in alert mode
Result: Alert region vector AR
begin
    Create m_a, PL_{m_a} = {[nodeID = s]};
    Broadcast m_a;
    Set timer T_1 = τ_1;
    while !T_1 (expired) do
        if receive m_a then
            if m_a.nodeID ∉ AR then
                Forward m_a;
            end
            Add m_a.nodeID in AR;
        end
    end
    Call VotingAlgorithm();
end
```

The phase of the alert region construction lasts time τ_1 in each node. A node will enter this phase when it detects the attack locally and produce the local alert. Therefore, it's not necessary that all nodes will enter this phase simultaneously, nor we require it. The value of the timer T_1 is set experimentally so that all nodes have sufficient time to exchange their alert messages and find out about each other.

Note that a faulty node (the attacker itself or a node collaborating with the attacker) may try to include itself in the alert region. Since we don't know the attacker *a priori* and, as we said, we don't require any reputation system to establish a trust relationship amongst the nodes, we have no other choice but to accept this possibility. By the use of the voting algorithm that we describe in the next section, we cancel out the effect of the faulty nodes.

A faulty node can also choose not to forward an alert message, in order to exclude an honest node from the alert region. However, we expect that the original sender will have at least one honest neighbor who will forward the packet and it will eventually reach the 2-hop neighbors. But still, in the case that there is only one path passing through the attacker, it is not to the benefit of the attacker to drop messages, since this will signify an instance of selective forwarding and the attacker will be identified more easily.

7.6 Voting Module

The `Voting` module is responsible for executing the protocol of the voting phase, which follows after the construction of the alert region. The goal of the voting phase is to have the nodes collaborate and exchange their suspect lists (we call them *votes*), so that they can agree on the identity of the

attacker. What is important here is to have honest nodes receive the votes of the rest of the honest nodes in the alert region and each vote is indeed the one transmitted by the corresponding node.

To achieve this, we must ensure of two things. First, that the votes of the honest nodes do not get lost, i.e. all honest nodes receive all votes from the rest of the honest nodes. This is possible because of the alert region that we constructed in the previous phase. Each node expects to receive the votes from the rest of the nodes in the alert region. If a vote gets lost, a node can request it and receive it (given that it has at least one honest alerted neighbor). The second thing to ensure is that the votes of honest nodes do not get spoofed by intermediate faulty nodes. For this reason, each node signs the votes using the next key from its key chain.

Algorithm 2: The *Voting* algorithm

```
Data: Node ID s being in the alert region
Result: Vector of collected votes
begin
    Create m_v,
    PL_{m_v} = {[nodeID = s], Suspect(s), [TTL = 1]};
    Sign m_v(s) with the next key K_j from the one-way
    key chain,
        m_v(s) = Suspect(s)||H(Suspect(s)||K_j);

    Set timer T_2 = τ_2;
    while ![T_2(expired)] && ![receive m_adv from all
    alerted nodes] do
        if receive m_v then
            for i ∈ m_v.Suspect(s) do
                if i ∉ Neighbors(m_v.nodeID) then
                    Discard m_v;
                    Break;
                end
            end
            Store m_v;
            if m_v.TTL == 1 then
                m_v.TTL = 0;
                Forward m_v;
            end
        end
        if receive m_v from all alerted nodes then
            Create and broadcast m_adv;
        end
        if [receive m_req ] && [s in advertise mode] then
            Forward the requested vote m_v;
        end
    end
end
```

The detailed protocol of this phase is described in Algorithm 2. Let us denote the message that bears the vote of node s as $m_v(s)$. Node s signs its vote with the next key K_j from its one-way key chain

$$m_v(s) = Suspect(s)||H(Suspect(s)||K_j)$$

and sets the TTL field of the message equal to 1. After it broadcasts this message, it sets a timer T_2 to expire after time τ_2. During that time it expects the votes from the rest of the alerted nodes. When it receives a vote, it first checks that the suspect list included is consistent with the neighbors of the initial sender. We need this check, in order to prevent a faulty node that has included itself in the alert region from voting against arbitrary nodes. It then buffers the received vote and forwards it if TTL=1. For the time being it does nothing more with the votes it receives. It has to wait for the key publishing phase that follows, in order to authenticate the votes.

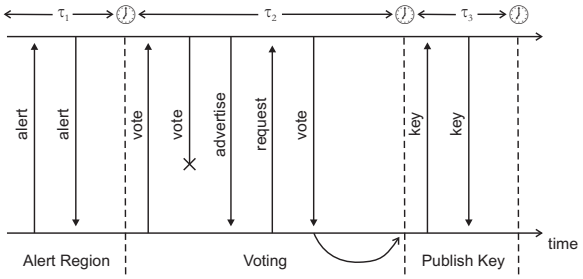

Figure 3: Message exchange between two nodes while they move through the phases of the protocol. Clocks indicate timer interrupts that force passage into the next phase.

If votes from all 1-hop alerted neighbors have been received, it broadcasts an advertisement message m_{adv}. This will allow any neighboring node to request for a vote that it may have missed. Thus, if a message m_{req} is received, the node forwards the corresponding vote again. This phase will end if the timer T_2 expires or the corresponding messages m_{adv} from all 1-hop alerted neighbors have been received.

The exchange of votes is optimized in terms of time. Votes can be lost, either by collisions or bad links. However, each node knows how many votes it should receive and by which nodes. So, if a node receives all the votes, it advertises it to its immediate neighbors and they request any votes that they have missed. We allow enough time for this process by setting a timer T_2 to expire in time τ_2. If a vote gets lost (not received by any node), the timer allows them to move on to the next step of publishing their key and verifying the authenticity of the received votes. On the other hand, if all neighbors of a node (itself included) advertise that they have received all the votes, there is no need to wait for the timer. That node can move to the next phase (see Figure 3).

In the Publish Key phase each node broadcasts the next key of its hash chain, K_j, which was used to sign the vote. When a node receives the disclosed key, it can easily verify the correctness of the key by checking whether K_j generates the previous one through the application of F. If the key is correct, it replaces the old commitment K_{j-1} with the new one in its memory. Then the node can now use the key to verify the signature of the corresponding vote stored in its buffer. If this process is successful, it accepts the vote as authentic.

Since a packet with the key can get lost, we need a third timer, T_3 as a "hard deadline" for receiving the keys. This timer is initialized just after a node publishes its own key and it's set to expire at time τ_3. When all the keys from the alerted nodes are received or the timer expires, the nodes move to the final step of processing the votes. In the case where a key has been missed, the corresponding vote is discarded. The code for this phase is given in Algorithm 3.

Since nodes are not time synchronized, and some nodes may start publishing their keys while others are still in the

voting phase, we need to consider "man in the middle" attacks. When a node sends its vote, an attacker may withhold it until that node publishes its key. Then it can change the vote, sign it again with the new key, and forward it to the next alerted node. Following that, the attacker also forwards the key, and the receiver will be able to verify the signature and accept the fake vote as authentic.

An explicit defense against this attack would be to require the nodes to be loosely *synchronized* as in μTESLA[16]. Here, however, we have decided to keep things simple and deal with this problem implicitly by relying on *residual paths* amongst the nodes (although we plan to investigate the synchronization approach and consider its possible benefits). As votes are forwarded by all nodes, even if an attacker refuses to forward a vote, it will arrive to the intended recipients via other paths. In the unlikely case that there is only one path passing through the attacker, dropping votes will signify an instance of selective forwarding and the attacker will be identified more easily. Thus, it is not at its benefit to modify/drop votes. Finally, in our algorithm, a node accepts a vote only while it has not publish its own key and it has not received the key from the node that sends the vote.

Algorithm 3: The *Publish Key* algorithm

```
Data: Buffer of received votes
Result: Attacker's ID
begin
    Release key K_j;
    Set timer T_3 = τ_3;
    while ![T_3(expired)] && ![receive K_j from all
    alerted nodes] do
        if receive K_j then
            if Validate(K_j) && Validate(m_v^j) then
                Store K_j;
            end
            else
                Discard m_v;
            end
        end
    end
    Aggregate all validated m_v to find the attacker's ID;
end
```

Let us also stress that the length of the key chain is finite and at some point all the available keys will have been used. Older keys cannot be reused, since they have been revealed by the nodes. Therefore, the nodes should be able to regenerate the key chain in a possibly compromised environment. To do that we follow the following method: before the node uses the last commitment, it creates a new hash chain and broadcasts the new commitment authenticated with the last unused key of the old chain. This essentially provides the connection between the two chains and the alerted nodes will be able to authenticate the votes as before.

When each alerted node has collected and authenticated the votes from the other members of the alert region, it will have knowledge of the corresponding suspect lists, itself included. Then it applies a local operator on these lists, which will produce the final intrusion detection result, i.e. the attacker's ID. In particular, it applies a count operator, which counts the number of times each node i appears in the suspect lists, or else the number of votes it collects. The node with the majority of the votes is declared as the attacker and its ID is passed to the `LocalResponse` module.

Since we assume the existence of t faulty nodes that collaborate with the attacker, we expect that they will have included themselves in the alert region and voted in order to affect the final result to the attacker's benefit. The best strategy for them would be to vote against a specific honest node, hoping that it will collect more votes than the attacker. If, however, there is a set of at least $t + 1$ honest alerted nodes, the majority vote will still point to the attacker.

7.7 Local Response Module

Once the network is aware that an intrusion has taken place and have detected the compromised area, appropriate actions are taken by the `LocalResponse` module. The first action is to cut off the intruder as much as possible and isolate the compromised nodes. After that, proper operation of the network must be restored. This may include changes in the routing paths, updates of the cryptographic material (keys, etc.) or restoring part of the system using redundant information distributed in other parts of the network. Depending on the confidence and the type of the attack, we categorize the response to two types:

- *Direct response*: Excluding the suspect node from any paths and forcing regeneration of new cryptographic keys with the rest of the neighbors.
- *Indirect response*: Notifying the base station about the intruder or reducing the quality estimation for the link to that node, so that it will gradually loose its path reliability.

IDS systems in other types of networks always report an intrusion alert to a human, who takes the final action. Correctly, this approach is usually neglected in WSN IDS literature. Sensor networks should (and they actually are) able to demonstrate an autonomic behavior, taking advantage of their inherent redundancy and distributed nature. Autonomic behavior means that any response to an intrusion attempt is performed without human intervention and within finite time.

8. CASE STUDY: DETECTING SINKHOLE ATTACKS

The architecture that we described in this paper is a general architecture for detecting intruders in a sensor network. In this section we take a closer look at an example of how this schema can be used to detect a specific attack, namely the Sinkhole attack. We use this example to show how the Local Detection module generates alerts based on the messages that it monitors and which rules one should built to analyze these messages.

In a Sinkhole attack [10] a compromised node tries to draw all or as much as possible traffic from a particular area, by making itself look attractive to the surrounding nodes with respect to the routing metric. As a result, the adversary manages to attract all traffic that is destined to the base station. By taking part in the routing process, she can then launch more severe attacks, like selectively forwarding, modifying or even dropping the packets coming through.

In this example we concentrate on MintRoute [19], which is the standard routing algorithm of TinyOS [7]. MintRoute

uses link quality estimates as the routing cost metric to build the routing tree towards the base station. For the calculation of these link estimates, each node periodically transmits a packet, called "*route update*". Each node estimates the link quality of its neighbors based on the *packet loss* of the route update packets received from each corresponding neighbor. The list of these estimates for each neighbor is broadcasted by the node periodically in its own route update packets.

In the case of a routing protocol like MintRoute, the compromised node launching the sinkhole attack will try to persuade its neighbors to change their parents and choose the sinkhole node as their new one, by trying to make these parents look like they have much worse link quality than itself. One method to achieve this is to change the link quality estimates sent by the nodes, within the route update packets.

To do that, the attacker listens to the route update messages from its neighbors, alters them and replays them impersonating the original sender. Even if there is an underlying key mechanism that nodes can use to communicate securely with each other, most probably the attacker will be using a broadcast key shared with the nodes to be able to overhear change and send these packets.

In order to detect the sinkhole attack we add a rule that will trigger an alert whenever the malicious node tries to impersonate another node, according to the attack we described above. The intuition is that route update packets should originate only from their legitimate sender and the nodes should defend against impersonation attacks.

Rule: *"For each overhead route update packet check the sender field, which must be the node ID of one of your neighbors. If this is not the case, produce an alert and broadcast it to your neighbors."*

Note that a node, which detects an anomaly according to the above rule, can infer the existence of an attacker, but it has not enough information to conclude on the attacker's identity, since the sender field of the packet is altered. The only conclusion it can draw is that the attacker is one of the neighboring nodes, since the route update packets are only broadcasted locally. Then, the node needs to rely on the cooperative detection engine described in this work, in order to reduce the candidates down to one node.

9. IMPLEMENTATION

In this section, we present experimental results from our implementation of the IDS system described in this paper. The goal is to show a framework that actually works on the motes, and can be used as a reference point. Moreover, it will become clear to the reader that such a system for sensor networks is lightweight enough to be a viable and realistic solution from implementation and real deployment perspective.

The current development of the IDS protocol builds on Moteiv Telos motes - a popular architecture in the sensor network research community. It features the 8 MHz TI MSP430 micro-controller and a 16-bit RISC processor that is well known for its low energy consumption. Yet, even though the implementation is tested on the Telos motes, all the components are designed with adequate generality, such that porting them to different sensor platforms should yield similar performance results.

9.1 Memory Requirements

The memory footprint of the LIDeA architecture is an important measure of its feasibility and usefulness on limited memory constrained sensor nodes. The total memory footprint is composed of the memory footprint of the compiled code, which is present in ROM, and the memory footprint of the data memory required to run the code. An IDS system for constrained devices must be compact in terms of both code size and RAM usage, in order to leave room for applications running on top of the system. Table 1 lists the memory footprint of the LIDeA modules, compiled for the MSP430 microcontroller.

Table 1: Size of the compiled code, in bytes.

Module	RAM usage	Code Size
NbPerimeter	136	968
Key Management	318	3764
Alert Region	94	766
Voting	260	4548
Total	808	10046

The largest module in terms of RAM footprint in Table 1 is the Key Management module. This is because the Key Management module contains statically allocated tables for the neighbors and their keys. In terms of ROM, the largest module is the Voting module, since it has the most lines of code. In total, the IDS consumes 808 bytes of RAM and 10,046 bytes of code memory. This leaves enough space in the mote's memory for user applications. For example, the total RAM available in Telos motes is 10 KB.

9.2 Experiments

To evaluate the performance of the implementation of the IDS, we tested it in a real environment. In particular we deployed several nodes in random topologies on the floor of an office building. We set a node to be the "attacker" and we gradually incremented the number of its neighbors to form larger alert regions. For each alert region size, we repeated the experiment for 20 different random topologies. The experiments were performed by having the motes running a typical monitoring application. In particular we loaded the Delta application, where the motes report environmental measurements to the base station every 5 seconds. We also deployed the MultihopLQI protocol at the routing layer, which is an updated version of the MintRoute protocol [19] for the Chipcon CC2420 radio. We tuned it to send control packets every 5 seconds. Our goal is to demonstrate how well the IDS will function, even under the presence of traffic on other layers. Then we simulated an attack to trigger the IDS protocol.

Figure 4 depicts the communication cost of the protocol measured in packets sent by a node. In particular, we broke it down to the packets exchanged for the alert region phase and the voting phase (as a total of exchanging the votes, ADV, REQ and keys). For small alert region sizes the cost is only about 12 packets, while for more dense regions the cost still remains low (21 packets). This is the total communication cost per attack and involves only the nodes in the alert region. It is also measured as a mean time averaged on different random topologies. The number of packets depends on the topology and the number of nodes in the alert region, as these parameters determine the number of alert messages, votes and keys circulated amongst them.

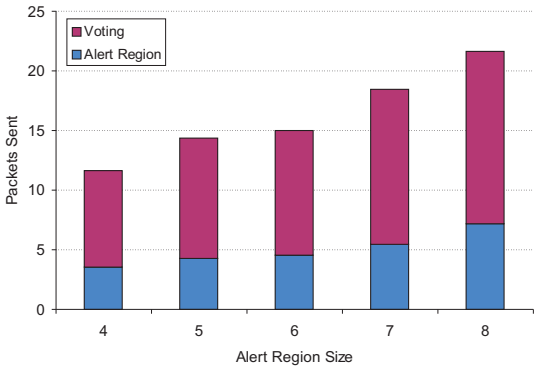

Figure 4: Measured communication cost for different sizes of the alert region.

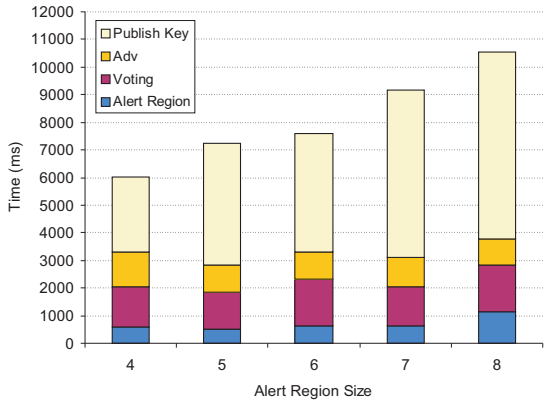

Figure 5: Detection time for different sizes of the alert region.

Next we measured the time that each phase of the IDS protocol required, i.e., the alert region formation, and the voting phase. We break the latter down to three smaller phases, to make it more transparent: the exchange of the votes along with possible requests (Voting), the exchange of the ADV messages and finally the publication of the keys along with the authentication of the votes and the computation of the final result (Publish Key). Figure 5 shows the measured mean times for each of the above phases, for different alert region sizes (i.e. attacker's neighborhood).

What we can infer from Figure 5 is that the times for the first three phases have small deviations as the alert region size increases, and constitute a small overhead from the total time. The most time-consuming phase is the last one of exchanging the keys and verifying the votes. To get a better insight of this, we measured the time needed for the computational operations within this phase. In particular, the time a node needs to authenticate each received key (i.e., to check if the hash of the new key matches with the previous one) is approximately $15ms$. The validation of the signature of the vote takes about $25ms$ and the aggregation of the received vote with the rest in order to produce the final result takes $150ms$. For the construction of its own vote, a nodes needs $60ms$ and signing it with its key takes $25ms$.

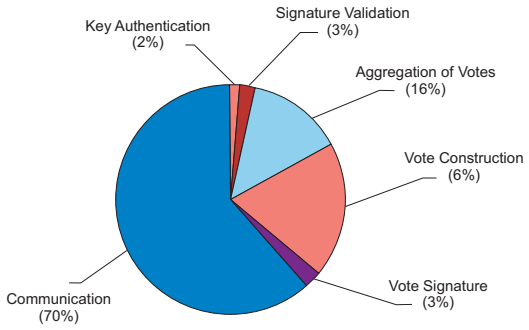

Figure 6: Costs of computation and communication in terms of time for the Publish Key phase.

Figure 6 expresses the percentage of costs for computation and communication for this phase. We can conclude that most of the overhead arises from the transmission of data rather than from any computational costs. This overhead for the communication is due to the inherent inability of TinyOS to receive the next packet before finishing the processing of the current one. In our implementation, upon receiving a key, the node has to verify it is a valid one before accepting it. To save memory space, we don't buffer the key for later processing, but rather we authenticate it on the fly. Meanwhile, TinyOS cannot receive the next key. That's why we had to include a random delay so that nodes publish their keys in different time instances. This delay, although experimentally minimized, contributes significantly to the results of Figure 6.

It is also important to see how the behavior of the IDS is affected by the traffic introduced by the application layer. That is, if the application needs to increase the data rate of the information routed in the network, the bandwidth that remains for the communication of the IDS agents decreases. We performed experiments to see how this affects the detection delay. In particular, we gradually increased the data rate of the Delta application. In Delta, each sensor node reports measurements periodically sending a packet to the base station. Delta is based on the MultihopLQI at the routing layer, which broadcasts route update packets periodically to maintain the routing tree. We tuned MultihopLQI to send a route update packet every 5 seconds and experimented with different data rates for Delta. For the MAC layer we used the default in TinyOS, i.e., the CSMA protocol.

Figure 7 shows the detection delay of the IDS as the aggregative data rate of packets at the routing and application layers increases. This increase actually corresponds to different packet rates of Delta (1 packet every 1, 3, 5 and 10 seconds), as the rate of the route update packets was fixed. The alert region size was set to 6 nodes throughout the experiments. As we see from the figure, for an increase of 300% in the data rate (from 34.8 bps to 139.2 bps) the detection delay is increased only by 1.6 seconds. As more and more packets are sent and received from the nodes, a delay to exchange the necessary packets for the intrusion detection is unavoidable, due to the CSMA back-off waiting time. We believe that a better MAC layer protocol would drop this delay further.

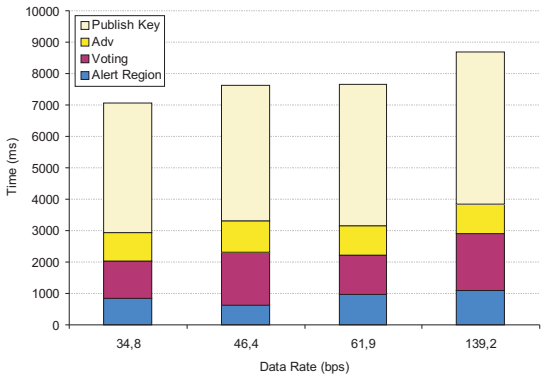

Figure 7: Behavior of the detection process under the presence of traffic on other layers.

10. CONCLUSIONS

In this work, we discussed the problem of intrusion detection in sensor networks that uses a large number of autonomous, but *localized*, cooperating agents in order to detect an attacker. The nodes use coordinated surveillance by incorporating inter-agent communication and distributed computing in decision making to collaboratively infer the identity of the attacker from a set of suspicious nodes.

The IDS system we discussed is novel but most importantly realistic, considering the current state of the art in wireless sensor networks. The demonstrated implementation details show that it is lightweight enough to run on sensor nodes, in terms of communication, energy, and memory requirements. We are currently working on making the system more robust against desynchronization and message loss, by considering the use of an ADV-REQ scheme. In general we believe that studying the problem of intrusion detection in sensor networks is a viable research direction and with further investigation it can provide even more attractive solutions for securing such types of networks.

11. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of this paper.

12. REFERENCES

[1] F. Anjum, D. Subhadrabandhu, S. Sarkar, and R. Shetty. On optimal placement of intrusion detection modules in sensor networks. In *BROADNETS '04: Proceedings of the First International Conference on Broadband Networks*, pages 690–699, 2004.

[2] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers University of Technology, March 2000.

[3] V. Bhuse and A. Gupta. Anomaly intrusion detection in wireless sensor networks. *Journal of High Speed Networks*, 15(1):33–51, 2006.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[5] A. P. da Silva, M. Martins, B. Rocha, A. Loureiro, L. Ruiz, and H. C. Wong. Decentralized intrusion detection in wireless sensor networks. In *Proceedings of the 1st ACM international workshop on Quality of service & security in wireless and mobile networks (Q2SWinet '05)*. ACM Press, October 2005.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, 2000.

[8] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *Software Engineering*, 21(3):181–199, 1995.

[9] H. S. Javitz and A. Valdes. The NIDES statistical component: Description and justification. Annual report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1994.

[10] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *AdHoc Networks Journal*, 1(2–3):293–315, September 2003.

[11] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981.

[12] C. E. Loo, M. Y. Ng, C. Leckie, and M. Palaniswami. Intrusion detection for routing attacks in sensor networks. *Int. J. of Distributed Sensor Networks*, 2005.

[13] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of the 6th International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 255–265, August 2000.

[14] I. Onat and A. Miri. An intrusion detection system for wireless sensor networks. In *Proceeding of the IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, volume 3, Montreal, Canada, August 2005.

[15] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[16] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. Spins: security protocols for sensor networks. *Wirel. Netw.*, 8(5):521–534, 2002.

[17] R. Roman, J. Zhou, and J. Lopez. Applying intrusion detection systems to wireless sensor networks. In *Proceedings of IEEE Consumer Communications and Networking Conference (CCNC '06)*, pages 640–644, Las Vegas, USA, January 2006.

[18] E. W. Vollset and P. D. Ezhilchelvan. Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in MANETs. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 166–178, 2005.

[19] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, 2003.

[20] W. Wu, J. Yang, and M. Raynal. Design and performance evaluation of efficient consensus protocols for mobile ad hoc networks. *IEEE Trans. Comput.*, 56(8):1055–1070, 2007. Senior Member-Jiannong Cao.