2014

DATA ALTERATION ATTACKS IN WIRELESS SENSOR NETWORKS: DETECTION AND ATTRIBUTION

James Nugent
University of Rhode Island, janugent@mail.uri.edu

DATA ALTERATION ATTACKS IN WIRELESS SENSOR NETWORKS: DETECTION AND ATTRIBUTION

BY

JAMES NUGENT

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2014

DOCTOR OF PHILOSOPHY DISSERTATION

OF

JAMES NUGENT

APPROVED:

Dissertation Committee:

Major Professor	Lisa DiPippo
	Ed Lamagna
	Yan Sun
	Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2014

ABSTRACT

Wireless sensor networks are an emerging field where a large number of cheap sensors are dispersed over an area in order to gather information. This paradigm of a large number of relatively low power platforms brings a number of challenges. Because the lack of physical security associated with traditional networking environments, sensor networks are much more likely to be taken over by insider attacks where the attacker gains access to all the information on the node and can perfectly emulate its behavior if they so choose. In addition the lack of computing resources means care must be taken due to the overhead introduced by additional protocols.

This work elaborates on the dangers presented by insider attacks in wireless sensor networks. In particular, an adversary node that appears to be a legitimate member of the network can alter data that passes through it. This is a more dangerous attack than the traditional packet dropping models because standard networking models will not be able to tell which node altered the data. In this way, even a small number of insider's nodes (even 1) can negate a large fraction of the network's functionality because, even if the data alteration is detected, there is no way to determine the node responsible.

The large spectrum of possible applications and behaviors for sensor networks makes it difficult come up with a single best solution. Because of that fact, this work introduces a number of different protocols to both detect malicious data alteration and attribute the malicious behavior to a specific node. It then describes what properties of a sensor network make specific solutions appealing as well as providing analysis of

how the strengths and weaknesses of each interact with possible sensor network configurations.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Lisa DiPippo, as well as the faculty, staff and students of the Department of Computer Science and Statistics.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Malicious Data Alteration in Wireless Sensor Networks

The study of wireless sensor networks [1] is an emerging field with important applications such as remote tracking and monitoring. In these applications, large numbers of small sensors are distributed over an area in order to gather information. Sensor networks must deal with the possibility of an adversary gaining control of a node, because the nodes are placed in the environment. This means an adversary altering the function of the network is a concern. Due to the wide range of applications, there is considerable variability in network configurations. We start by describing a basic and common sensor network configuration and discuss additional aspects later.

Because sensors are embedded in the environment they are monitoring, an adversary may have physical access to the devices. Because of this and the wireless nature of the network, an adversary has more opportunity to gain control of a node. This is called an insider attack.

We expand upon the implications of insider attacks in wireless sensor networks and explain how these allow data alteration attacks that are more dangerous than the traditional packet dropping models. We investigate methods of both detection and prevention of such attacks.

1.1.1. Description of a Wireless Sensor Network

A basic sensor network consists of a *base station* or *sink* and a number of small sensors scattered throughout the environment called *nodes* or *motes*. The nodes sense the environment and send that information to the base station (BS) via wireless radio links. This information is called *events*. The node that generates a particular event is called the *source*. Data flows from the sensor nodes to the base station. Thus, we say the base station, and nodes closer to it, are *downstream* from a particular node, while nodes further from the base station than the current node are *upstream*. Because a sensor network is defined by its environment, all communication must be done wirelessly.

There are several challenges that almost all sensor networks share due to their nature, applications and universal features of the architecture. Thus, almost all work in this area is concerned with these to some degree or another.

The first challenge is *scalability*; many applications require large numbers (thousands to tens of thousands) of nodes (imagine monitoring a square mile with nodes with a 50 foot range). Another challenge is *cost* because supporting large numbers requires the nodes to be small and cheap (possibly even disposable). Another challenge is that all sensor network applications must deal with severe *resource constraints*. The fact that nodes have limited *processing power*, *memory*, and *storage* must be taken into account. The last challenge is that nodes are almost always battery powered, so *energy* efficiency (aka node lifetime, as recharging the nodes is rarely practical) is a concern. The largest source of energy consumption in sensor networks is the wireless radio, so *communication* efficiency must be considered. Because of the limited power available to sensor nodes, nodes have limited radio range, making it unlikely all nodes can reach the base station. This means a routing tree is needed so nodes know which of their neighbors to send data to so that it moves closer to its destination. For a given *event* the data originates at the *source* node, and is transmitted through some number of *routing nodes* before arriving at a *destination* (often the base station).

1.1.2. Security in Wireless Sensor Networks

We assume an application where security [25] is important. For example, many military applications fall into this category. That is, we assume there is an *adversary* who wishes to impede the function of the network for their own gain. The adversary can listen in on the radio traffic of the network and will often have physical access to the nodes. We assume the adversary can send whatever transmissions they wish as well as arrange for known inputs to the network and coordinate their own activities.

1.1.2.1. Basic Security Model

Encryption is required for security in a WSN, since without it, an adversary could easily access all the wireless communication. For applications with significant security requirements a single global key does not suffice; if any node is taken over, the key is compromised. Thus, we assume some sort of public key encryption [21] is in effect, i.e., a single node being taken over will not allow the adversary to read all communication in the network or impersonate other nodes.

Many WSN security systems are identity based [10][16]: the focus is on making it impossible for an adversary to add new nodes to the network or impersonate a node by sending a message that other nodes will think comes from the loyal node.

Nodes are placed in the environment where, unlike most networking domains, the adversary often has physical access to the nodes. Physical attacks [6] can produce what is called an insider attack. This is where the adversary gains complete access to a node, including all of its information (e.g., keys). The important feature of an insider attack is that the attacker gains all information that the node had at the time it was taken over. These attacks can defeat identity based security as the adversary causes a loyal node to work for them.

Identity based methods are ineffective against insider attacks, since the adversary has all the information from the corrupted node. The key feature of an insider attack is that they can perfectly impersonate the corrupted node. This means that an adversary can only be detected by their behavior, i.e., when they take malicious actions.

1.1.2.2. Adversary Model – Malicious Data Alteration

The goal of the adversary is to prevent the flow of events from the sensor network to the base station, while preventing the owner of the network from taking action to restore the flow. The fact that the data is encrypted means that an adversary who is trusted by the network cannot simply change data and pass it off as real. This means that traditionally, the adversary is restricted to dropping packets they don't want to reach the base station. Dropping is effective, but since it can also happen naturally due to unreliable data transmissions, a large variety of mechanisms have been created [2] to detect it and pinpoint unreliable nodes. These mechanisms generally work by noticing that data has not arrived when it should. The data can then be retransmitted.

A more powerful attack for the adversary is *data alteration*. In this case, data is changed and sent along the network to the base station. This is a more powerful attack for several reasons. For one, data alteration wastes network resources processing and transmitting useless data. This is undesirable even if the false data does not fool the destination. Another is that without additional mechanisms, routing nodes cannot detect that data has been changed after leaving the source. Thus, even if the fact that data has been changed can be detected, the node that did it cannot be identified. This means steps to prevent the same thing happening in the future cannot be taken. It is as if the data was dropped, except the mechanisms that detect this do not work because incorrect data arrived. Since this is an attack on data that is routed through a node, a single compromised node can alter all data that is routed through it. Depending on where the adversary node is in the network topology, a significant fraction of the network may not give correct information. Even a single compromised node can have a devastating effect while being impossible to detect with standard security mechanisms. We believe the exceptional power and detection difficulty of these attacks makes methods of detecting them worth studying.

Malicious data alteration by an adversary is different from the non-malicious data corruption caused by wireless physics [31], where transmitted data may randomly change due to physical effects (the same problem occurs in wired data transmission also, but the error rates for wireless are generally higher). We assume a standard packet level *checksum* [31] is in place to catch these sorts of errors. Since these are based on well-known methods an adversary can easily fake them however. *Our solutions focus on ways to detect data alteration outside of the checksums.* This also means that if data is received that is detected to be altered but has a correct checksum, we know it is the result of malicious activity because it is highly improbable that data corruption produced a correct checksum.

1.1.2.3. Adversary Goal

We assume the goal of the attacker is to prevent data reaching the base station. We also assume the attacker is using an insider attack as mentioned before. Because of this, we assume the only way to discover an attack is by its effects. We detect the presence of an adversary by detecting the fact that the data changed. Since we assume the attacker completely controls nodes they have taken over, we do not make any effort to detect newly created forged data from a compromised node (we will mention ways of dealing with this as related work, but they are outside the scope of this project). We are concerned about the attacker having an effect beyond nodes they have taken over and altering data routed through them, so we focus on the detection of altered data.

1.2. Goals

In this dissertation we present several solutions to the data alteration problem. The goals of these solutions are to:

Detect Malicious Data Alteration: Specifically, we ensure that data comes from a single source node and is not altered after leaving the source. Detecting alterations earlier in routing is better, due to less waste of resources on unproductive data, but what specific timeframes make sense depend on the application.

Attribute Alteration to Adversary Nodes: The detection of malicious data alteration is not useful if we cannot tell which node did it, as the attacker then can prevent data reaching the base station whenever they want. Attribution allows standardized mechanisms to be invoked to deal with adversary or unreliable nodes. In addition, they can detect, with a high probability, whether the alteration is natural data corruption in transmission or the result of an attack.

Minimize Overhead: Since resource constraints are a fact of life in sensor networks, we must consider the impact our solutions have. Message overhead (per packet and extra messages), computational overhead and space overhead (storage and memory) will be considered.

CHAPTER 2

BACKGROUND

In this chapter we introduce several topics related to the detection of malicious data alteration in sensor networks. We also expand upon some details of wireless sensor network functioning that are important to this discussion.

2.1. Encryption

Encryption is a mathematical process for controlling information. Here we provide general information on the process, while how it is applied to detect data alteration in wireless sensor networks will be described in later sections.

Encryption [21] is the process of turning usable data, known as *plaintext (p)* into a collection of bits that appears to be random, called *cypher-text (c).* The reverse of this process, turning cypher-text back into plaintext, is called *decryption.* Cryptography is the study of these two processes. Modern cryptography is based on *one-way, trapdoor functions*. A *one-way* function is one where $f(p)=c$ is easily computable, but the inverse, $f^{-1}(c)=p$, is much more difficult. Generally, there is no known efficient algorithm to compute the inverse of a function used for cryptography. A trapdoor function is a function that has a piece of information called the *key*. If one does not have the key, it functions like a trapdoor function. In short, the most efficient way to compute the inverse of a trapdoor function without the key is to guess the key. The function f is used for encryption ($f=E$), while f^{1} is used for decryption ($f^{-1}=D$). In trapdoor functions the key is used in both f and f^{-1}. Key sizes and key security are generally talked about in terms of the number of bits in the key. The implication is that it takes approximately $2^{|key|-1}$ tries to correctly guess the key. If the number of bits is large enough then guessing the key in this manner will take so long that the information is worthless to the adversary.

A *symmetric encryption* algorithm [21], also called a private key algorithm, is one where the same key is used for encryption (E) and decryption (D). Thus, $E_k(x1) = x2$ and $D_k(x2) = x1$. The key (k) is commonly shown as a subscript. Keys will often be named after the parties that hold them. Two parties who have the same key, k, use this to communicate security by computing $E_k(x1) = x2$ and sending $x2$ to the other party. The receiver then computes $D_k(x2)$ to read the original message, $x1$. An adversary can intercept $x2$, but without access to the key (k), they cannot compute D_k and cannot recover $x1$.

An *asymmetric encryption* algorithm (E) is one where there are two keys, $k1$ and $k2$, $k1$ not equal to $k2$. If one key is used for encryption, the other key is used for decryption. That is to say, $E_{k1}(x1) = x2$ and $D_{k2}(x2) = x1$. With asymmetric encryption algorithms, generally one key is kept secret by the user A and called the *private key* (*pri*) and the other key is publicly available from a reliable source, called the *certificate authority* (CA) and is known as the *public key* (*pub*). For this reason, asymmetric encryption is also called public key encryption. Thus, anyone who wants to send a message ($x1$) to A can get A's public key from the CA and send a message ($x2$) that only A can read by computing $E_{pub}(x1) = x2$. Then, only A can read it by computing $D_{pri}(x2) = x1$. It is worth noting that public key encryption algorithms are generally much more computationally intensive than private key algorithms.

Hashing is a deterministic computational process that produces a fixed length *hash* from an arbitrary length input. The hash is generally much shorter than the input. The key feature of hashes is that it is very unlikely that two inputs produce the same hash (which is called a *collision*). This means they are used as a way of quickly doing comparisons. If two messages have different hashes, one knows they are not equal. Hashes used for cryptography, sometimes called *cryptographic hashes*, have the slightly stronger property of it being difficult, given a hash value, to find an input with the same hash value. With a cryptographic hash function, hashing data is somewhat analogous to encryption, although there is no analog to decryption.

Digital signatures, which we generally call signatures for simplicity, use a combination of these two concepts. In this case, A wants to *sign* some data (D) such that other users can be sure the data comes from A. To do this, one generally uses a well-known cryptographic hash function H to compute $H(D)$, which is called a *hash* of D. A then computes the *signature*, $S=E_{pri}(H(D))$. When talking about different nodes, we may show the signature as $S=Sig_{ID}(D)$. This refers to node ID encrypting a hash of data D with it's private key. Suppose another user B gets *D* and *S* and wants to be sure *D* is from A. Since the signature is just an encrypted hash of the data, B can then get A's public key from the CA and check $E_{pub}(S,) = H(D)$. If this is true, then B can be sure that the hash of the data was encrypted with A's private key and be sure that *D* and *S* must have come from an entity possessing A's private key (i.e., A).

The motivating problem for *group signatures* [7] was to find a way that data could be signed in such a way that it could be verified as coming from a member of *a* group, but there was no way to know *which* member of the group. The original problem was stated in the context of whistleblowers, so that information could be verified as coming from an official source, but the specific person's identity was hidden. In a group signature scheme users are organized into groups, with all the keys in a group being created from a single source. Group signatures are created by the sources private key the same way they are in a public key encryption system. However, only a single key, known as the *group key*, is required to verify a signature from anyone in the group. The difference being that one cannot tell *which* member of the group created the signature, but one can be sure that it was a member of the group.

2.2. Checksums

Checksums are conceptually similar to hashes in that they create a fixed length output from a variable length input. The difference is that a checksum is focused on error detection as well as error correction. Specifically, it is highly unlikely that *random* changes to the data produce the same checksum. Since a checksum is a well known algorithm and easily computed, they can be easily forged by an adversary. Unlike a cryptographic hash, it is not necessarily difficult to find two pieces of data with the same checksum if one looks. It is just unlikely that random errors would change data in such a way the checksum is the same. Checksums tend to be slightly less computationally intensive than cryptographic hashes for this reason. Also, checksums tend to have shorter outputs (fewer bits) than cryptographic hashes for the same reason. For both these reasons we use checksums rather than cryptographic hashes when we only care about data correctness, rather than security. We denote the checksum of data D as $CHK_D=CHK(D)$.

2.3. Wireless Sensor Network Setup

Many wireless sensor networks use what is called a *secure setup phase*. Essentially, an enemy is assumed to not be able to penetrate the network during setup. A simple way to secure the setup phase is to use a global key that is shared by all nodes and then discarded when setup is done. This phase is too short for an enemy to crack the global key. Also, due to the short duration of the setup phase, there are no insider attacks during it. For the rest of this work we assume distributing secret information (keys) during the setup phase does not present a security problem. It is much more difficult to distribute secret information after the setup phase, so we address that when it is required. In general, we do not need to distribute keys after setup.

During this phase nodes figure out their *neighbor set*. The neighbors of a node are the nodes that a node can reach with its wireless signals. A single wireless link is called a *hop*, and consists of a single sender sending to a single receiver. Since not all nodes can directly communicate with the base station, *multi-hop routing* must be used to send packets to nodes that a node cannot communicate with directly. In this, a node sends a packet intended for a destination it cannot directly reach to one of its neighbors which then sends it to another node closer to the destination. By repeating this process the packet eventually reaches its destination. *Routing* is the process by which a node knows which of its neighbors it should send a packet to based on the packet's destination. This information is stored in a *routing table* at each node. The overall routing information for the network is called a *routing tree*. We assume that shortest-path routing is used, so that packets take the fewest hops to reach their destination. In a sensor network generally data flows *downstream* toward the base station. Nodes further from the base station than the current node are said to be *upstream.*

Routing in sensor networks is often done relative to the base station because that is the direction of data flow. Thus, nodes usually only have routing information about how to send packets toward the base station and how to return acknowledgements to nodes upstream rather than having general routing information

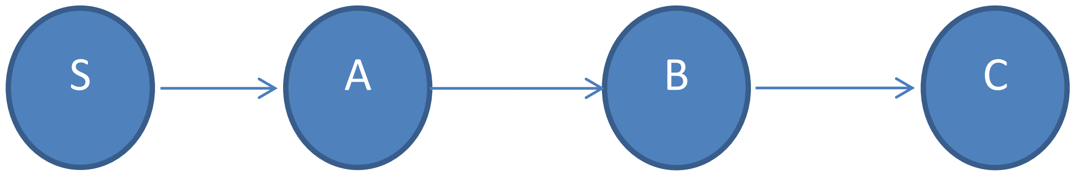

Figure 1 - Overhearing example

for all possible destinations as in a more general network. We will discuss other methods of routing in other sections as appropriate.

2.4. Watchdogs

Most security work for sensor networks focuses on preventing intrusion in the first place rather than detection intruders or data alteration.

One concept that does detect alteration is the *watchdog*, where nodes observe the behavior of their neighbors [19]. Consider Figure 1, where S is the source node, A, B and C are routing nodes and the arrows show the direction of data flow. The watchdog concept uses the fact that, since wireless nodes are often very similar to each other, the wireless links are usually (but not always) bidirectional. If the signal from A can reach B, then the signal from B can usually reach A. So, if A sends a packet to B, it can then listen and hear the packet that B sends to C. If A saved the packet that it sent to B, it can compare it to what B sent to C. The watchdog does not have to be a node on the events route to the base station, although that is the simplest case.

There are several limitations to this technique, simple as it is. One is the overhead. Memory is required to store packets for checking. More importantly, the node must use energy to listen at a time it might not otherwise be doing so. Another limitation is that only a single adversary node can be detected. If both a node and its watchdog are adversarial, then packets can be altered undetectably.

Another limitation is *collisions*; a collision occurs when a node receives signals from multiple sources at the same time. The signals interfere with each other and the node cannot make sense of either signal. We will assume all the links in Figure 1 are bi-directional. Collisions are taken care of by how the wireless *scheduling* [26] works. The scheduling algorithm manages when nodes send and receive such that collisions do not occur.

Normally, if B is sending to C, A can be sending to any node other than B or C, sending acknowledgements to S for example. If A wishes to watch what B is sending to C, that means that no node that A can hear can be sending during that time. It is possible to take this into account in the scheduling algorithm, but it can reduce the utilization of the network because fewer nodes can be sending at the same time. Alternatively, we can allow these collisions to occur and accept that the watchdog misses some packets. This means the packets will be kept in memory until they are removed by a timeout mechanism, wasting space. If the network is busy enough, an adversary may be able to use the traffic to hide alterations behind collisions.

The watchdog concept is useful for dealing with the problem of data alteration, but it does not give a general solution to the problem.

CHAPTER 3

DETECTION AND ATTRIBUTION OF MALICIOUS DATA ALTERATION

In this chapter we describe three solutions to the problem of detecting malicious data alteration and how to discover the source of it so that it can be prevented in the future. We first present a naïve solution that acts as a starting point. Most of our solutions are focused on removing the drawbacks from the naïve solution in different ways. We also present some methods that use a completely different approach in section 3.3.

3.1. Solution Introduction

To deal with the problem of data corruption caused by insider attacks we present a number of possible solutions. The contribution of this work is to elaborate on the problem of malicious data alteration in sensor networks. The wide variety of possible environments, applications and requirements seems to preclude a "one size fits all" solution, so we investigate a variety of approaches and compare them. Generally we try to force an attacker to choose between disrupting the network and being detected.

Due to sensor nodes being small and subject to environmental effects, malfunctions are more common than in traditional networking environments. Because of this we assume basic reliable networking protocols are in place [31] [33], i.e., some sort of timeout and retry mechanism that is used to ensure eventual delivery of import information and correct operation of the system. Because of the unreliability both due to the small nature of the nodes themselves and their interactions with the environment, wireless sensor network routing protocols have the ability to detect when nodes do not forward data they should and to route around them. Basically, we assume that an adversary or malfunctioning node that simply drops data will be discovered and removed from the network in an application appropriate time frame. Thus, dropped packets are not the concern of this work.

Second, we assume each node shares a unique symmetric encryption key with the base station. We call this the *source key*. This key is only used to encrypt data that the node sends to the base station (i.e., when it is the source). This symmetric encryption algorithm is chosen for efficiency and the required level of security. What we call the "data" for the rest of this work is actually a few fields: $D= \{ID_{src}, E_{src}(SD, CHK_{SD})\}$. The source id ($ID_{src}$) is the globally unique id of the source node. SD is the actual sensor data. The data checksum (CHK_{SD}) checks the sensor data and source ID. When sending a new event, first, the sensor data itself and data checksum are encrypted with the source nodes key. The presence of the data checksum means the base station can always tell if the data decrypted correctly. The data checksum is checked when the data is decrypted at the base station. If this check fails, we know the recovered data is meaningless and we say the decryption has failed. Next, this encrypted data is paired with the source ID so the base station knows what source key to use to decrypt it. The source key does not need to change over the life of the system. Even if the source key is compromised, the attacker can only forge data from that single node, so the return relative to the effort is minimal. An insider attack will give the adversary the source key for the compromised node, but all this allows them to do is forge data from that node. Also, the source key can easily be preloaded on nodes, perhaps even when the node is manufactured or initialized.

The reason for encrypting the sensor data with the source key is so we can easily detect when data is changed by nodes that are routing the data back to the base station. A routing node does not have the source key for the source node, so any change to the data will result in a checksum failure and failed decryption at the base station.

If the adversary alters the source field to refer to a node other than the compromised node or the source node, the base station will use the wrong key and not be able to decrypt the data. If the adversary alters the source field to refer to the compromised node and substitutes fake data encrypted with the compromised node's key, then the reliability mechanism will detect that the packet from the original source has effectively gone missing. Thus, an alteration to the source field is the same as an alteration to the data from the base station's perspective. In addition, the base station could simply try all the nodes keys if it suspects the source field is altered and possibly recover the data that way. For these reasons, it makes the most sense for an adversary to alter the data field itself.

Since the attacker does not have the source key for any node other than the ones it takes over, a change to any data that is routed through a malicious node will be detected by the base station because the data will not be decryptable. This does not help us detect data alteration in the network, but it does help prevent false data being used and avoids the question of determining whether data has been altered. With this precaution, since the base station knows what key was used to encrypt the data, it can always verify the source of good data.

This does not fulfill our requirements as it does not provide a way to know the origin of the bad data or give a way to prevent it from happening, so the attacker can still prevent data from reaching the base station while avoiding detection. The only benefit is that the base station will not use bad data.

We also use this node-base station key pair to assume that the base station can communicate securely with any single node. That is, messages from the base station to nodes (e.g., acknowledgements) also cannot be altered or forged. For systems that use public key encryption, we assume the base station also has a public-private key pair using the same encryption algorithm. The base stations public key is distributed to all nodes during setup. We do not assume this in cases where the public key encryption algorithm would not otherwise be needed. Asymmetric encryption is important in this case, as a symmetric scheme could be forged by an adversary.

3.2. Naïve Solution-Public Key Cryptosystems

To introduce our approaches to this problem we will first discuss a simple, if impractical, solution to the problem we put forth as a basis for further investigation. In the naïve solution we use a form of public key encryption [21] and each node has a private key known only to itself that is used to sign packets and a public key that other nodes use to verify its signature.

During the setup phase all nodes send their public key to the base station. Nodes then remember all keys that route through them and can use those public keys to verify signatures from their nodes.

3.2.1. Detection and Attribution of the Data Alteration with Naïve Solution

The source node computes the signature $S_D=E_{pri}(H(D))$ with its private key and sends it along with the data. Nodes along the way to the base station use the whole packet checksum to verify correct transmission (and ask for a retransmission if needed). If the packet is correct the node uses the public key specified by the source ID to verify the signature and sends it on if the signature is correct.

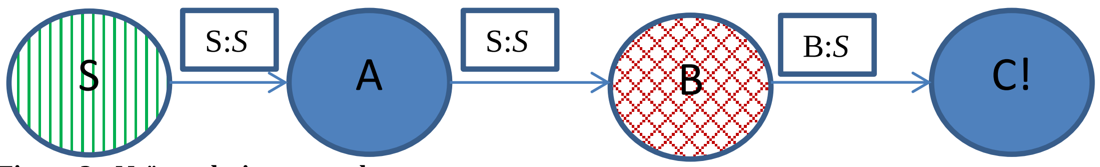

Figure 2 - Naïve solution example

Figure 2 shows this process via an example. S is the source node. It sends the rectangular data packet that consists of the data on the left and signature in italics on the right. The letter in the data and signature fields shows which node created them. Node A is loyal. It verifies the checksum and signature and then sends the packet to B. Node B has been taken over by an adversary. It alters the data and sends it onward. It can’t forge S’s signature because it doesn’t have S’s private key however. As mentioned in section 3.1, B altering the signature field doesn’t change anything. Node C is the loyal node that detects the data alteration. After verifying correct data transmission it checks the signature using node S’s public key and realize it doesn’t match. This allows C to realize that the packet has been altered and that node B either altered the packet or passed the altered packet without checking it. Either way, C can conclude that node B is an adversary. The facts can then be passed to the security framework for appropriate action.

We also use this system, along with the base stations public key to secure “important” transmissions from the base station in all protocols that use public key cryptography. What is important varies by application, and we will note the instances where the base stations public key is used in our protocols.

If there are multiple adversary nodes, only the one immediately preceding the loyal node that detected the change will be detected. This is because we know that the previous node should have performed the same check, so the fact that it did not notice the change means it is an adversary. Since there is no way to tell if an already changed packet is changed again we can only detect one adversary node per data packet. This is a general property of our solutions. Since we detect adversaries by their actions, we can only detect a single adversary per action taken by them, since there is no way to attribute changed data to more than one node. It is hoped that this is a poor exchange for the adversary.

Since every node has a unique public key, there are no additional actions needed if a node is removed from the network, as the key will not be used again. The only difficulty is if the routing tree changes, in which case nodes may need keys for nodes they did not before. The easiest solution is to have the base station send out the required public keys. If the number of nodes that need updates is small, the base station can send the keys directly, encrypting them with the node's source key. For a larger number of nodes, the base station could do a limited broadcast of the new keys to nodes that need them. This transmission would be signed by base station's own public key to prevent it being tampered with. Reconfiguration of the network requires an authenticated way of sending out routing updates in any case. Due to having information on the entire topology, it is not difficult for the base station to figure out what nodes need what keys.

3.2.2. Naive Solution Evaluation

This solution fulfills our goals of detecting data alteration and being able to attribute the alteration to a specific node. We will now consider the overhead. We assume any application that cares about security and reliability would use encryption to prevent an adversary listening in on the data and checksums to detect transmission errors, so we ignore the symmetric key encryption done at the source and checksum computations at each hop as these would happen in almost all applications.

Due to global key systems having a single point of failure in the global key, there has been a general research effort toward realizing public key cryptosystems in wireless sensor networks [10]. Both energy efficiency [32] and approaches such as coprocessors [24] have been studied. Mainly for reasons of efficiency, Elliptical Curve Cryptography has emerged as the preferred type of asymmetric cryptography for wireless sensor networks [30]. In short, we believe using public key systems in sensor networks is practical for applications with high security requirements, while hardware advances will make it more accessible in the future. The computational requirements of public key cryptography do appear to present a not insurmountable barrier for applications with sufficient security requirements.

To make this more concrete we will pick an implementation and use it as a basis for comparison. TinyECC is an elliptical curve based library targeting sensor network applications [16]; it was benchmarked on the common MicaZ[11] using the ATmega128 processor [4]. They used the standard SECG [9] recommended parameters for a 160 bit curve. This provides equivalent security [8] to the standard 1024 bit RSA keys [27].

Setup Phase: The setup phase generates 1 additional message per node that is used to distribute that nodes public key.

Message Overhead: No additional messages are generated by the naïve solution, so the message overhead is in packet sizes from the signature on each message. The signature has a size of 40 bytes.

Computational Overhead: Data is signed at the source and the signature is verified at each hop downstream. Thus, the computational overhead varies by node, but is solely determined by the role a node plays for a particular packet.

S(S) = Signature Generation Time: 2.00162s (source node)

V(S) = Signature Verification Time: 2.43646s (routing nodes)

Memory Overhead: A node's private key is 20 bytes. The main source of memory overhead consists of a table that maps an ID_{src} to the corresponding public key for every upstream node. *U* is the number of upstream nodes. *PK* is the size of the public key, which is 20 bytes. Let *I* be the size of a node id. For *I*, 2 bytes gives 65535 possible node IDs. This is probably sufficiently large for most cases. If more keys than 65535 can be stored, it is likely that memory is less of an issue. We will treat node ids as two bytes in size from here on. The total memory overhead is $(PK+I)*U$, or 22 bytes per upstream node plus 20 bytes for the private key.

This is where the naïve solution fails our goal of minimizing overhead. To use this system *a node must have the public key for any node that could send data through it on its way to the base station*. For a node that is near the base station, this could be a very large number of nodes. For example, if there are 3 nodes that are a single hop from the base station, then one node would need keys from a third or more of the network (if one of the nodes serves less than a third of the network, then another must serve more than a third). Even 1000 keys would take 22KB of space, which is about 1/6 of the 128K of program memory on a MicaZ mote and much larger than its 4K RAM size.

Thus, this solution fails to provide a reasonable overhead cost, while fulfilling the other requirements. It does serve as a basis for the solutions we discuss in the following sections however.

3.3. Topology based solutions

The problem with the naïve solution is that a node requires keys from any possible source node that could route through it. In this section we investigate a

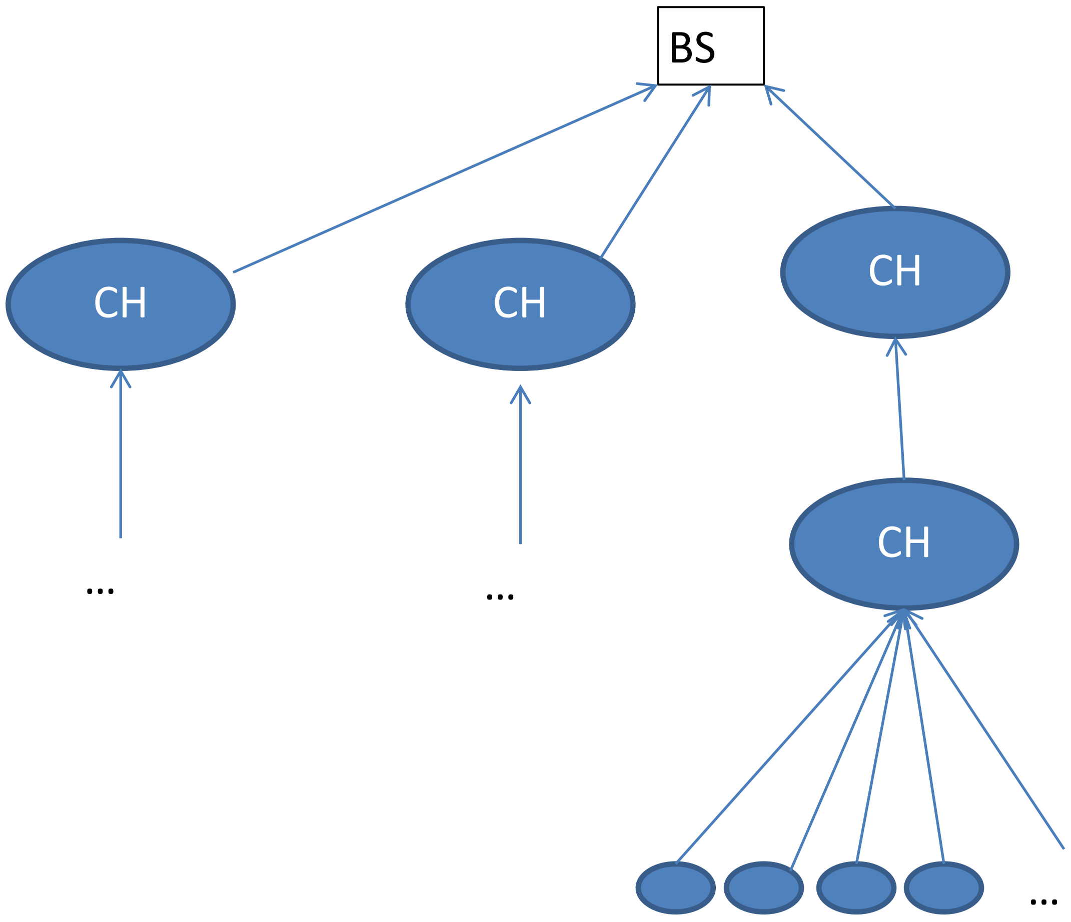

Figure 3 - Hierarchical Network

topology that reduces that problem by allowing a large number of nodes to be organized in a manner that effectively reduces the number of source nodes at the cost of requiring a specific network setup.

3.3.1. Hierarchical Networks

In a *hierarchical network* [28] [22], the network is organized into *clusters* as shown in Figure 3. Each cluster consists of a number of leaf nodes (small circles) and a cluster head. Data is sent to the cluster head and then routed to the base station through other cluster heads. Each leaf node is within a small number of hops, usually 1, of its cluster head. Since the cluster heads route all data from their cluster, they transmit more data on average than a node in a more general topology. Because of this, cluster heads are often more powerful than the leaf nodes. Cluster heads may also execute other functions, such as *data aggregation* [17], where data is processed and combined in some way, such as computing an average, before transmission.

3.3.2. Naïve Solution in Hierarchical Networks

3.3.3. To utilize the naïve solution in a hierarchical network we treat a cluster the same way we would a single source node in a more general network. Leaf nodes send their data to their cluster head, called the *source cluster head*. The cluster head then signs it and sends the data to the base station. Other nodes that route the packet check the signature of the source cluster head against its public key to see if the data has been altered in the same way that routing nodes check the signature of the source node in the naïve solution. As in the naïve solution, a failed signature check indicates altered data. In addition, we can attribute the data alteration to the node (which must be another cluster head) that sent the packet. The detection and attribution goals are fulfilled in the same way as in the naïve solution. As in the naïve solution only a single adversary can be detected per changed packet. Topology changes are also dealt with in the same way as in the naïve solution. **Evaluation of the Naïve Solution in Hierarchical Networks**

The network topology has no effect on the individual public key computations themselves, so these are the same as in the naïve solution. The overhead that is different is the memory overhead required to store public keys. Like the naïve solution, a node requires the public key for any *cluster head* that could route data through it. Because the topology has clusters and most of the network is leaf nodes, the number of public keys a given node needs to store is much smaller. To check for data alteration, routing nodes only need to store the public key of *any source cluster head* that can route data through them, not any *source node*. Since each source cluster head can be linked to a number of leaf nodes, the number of keys a routing node needs to store is greatly reduced.

If, say, the network is 90% leaves and 10% cluster heads, then since we only check the signatures of cluster heads, the number of potential sources is reduced by a factor of 10, and the number of public keys that need to be stored on average is reduced by the same amount. A hierarchal topology tends to have shorter paths to the base station than a more general layout with the same number of nodes due to most nodes being leaves. This reduces overall computational overhead also, but it remains the same per node.

Since leaf nodes connect to the cluster head directly or by very short paths, a leaf node that becomes an adversary can only affect data from a small number of other nodes. Because of this we only worry about cluster heads altering data. The only effect of the topology change is to reduce the number of public keys that need to be stored at routing nodes to check for data alteration.

3.4. Extending to General Sensor Network Topologies

The difficulty with the naïve solution is the number of public keys a node can be required to store in a large network. In this section we present a method to work within the storage limits of a node without requiring a specific topology. The basic approach is for each node to store as many keys as it can from upstream nodes and to use the naïve solution when a node has the key for a source node. We assume a node can store N public keys, where N is smaller than the number of possible upstream sources. So a node stores only a subset of the keys it would need to verify signatures from all possible sources. In this section we show how this subset can be used to protect data from alteration.

We can think about the problem of having a node only able to store N keys in two ways. From the perspective of a source node, as data gets closer to the base station, the number of possible upstream sources increases and space required increases. Nodes allocate key space for nodes closer to them so that the range where a given nodes key can be verified remains contiguous.

To protect data from alteration with a subset of the upstream keys, the general idea is to store as many keys as is practical from upstream nodes and re-sign the data when the distance from the source node is such that nodes did not want to allocate key space for that source node.

3.4.1. Re-signing: Setup phase

For this section we concentrate on a simple, intuitive algorithm. The best way to allocate the public key space available to a node is a subject for future research. To set up the re-signing algorithm a node gathers all the public keys it can store. We assume for simplicity that all nodes can store the same number of public keys and that nodes store keys for nodes closest to them rather than far away. To start with, each node sends the IDs of its neighbors to the base station. This allows the base station to figure out the entire network topology. The base station then tells each node how many nodes are upstream for each neighbor it has and how it should split key space to each upstream branch. This information is then used to allocate the key storage space at each node. Generally, the base station tries to allocate each node’s key space so that nodes within a certain number of upstream hops can be verified.

We will consider the setup phase from the point of view of node *A* in Figure 4. The node labeled *X* is in the direction of the base station relative to node *A*, i.e., it is downstream. We assume *A* has space to store 9 public keys. Node *A* starts by creating 9 virtual tokens, one for each key it can store. Any node that receives a token from *A* sends its public key to *A*. *A* then splits its tokens proportionally to how many upstream nodes there are for each neighbor. It sends 4 to node C, 2 to node F and 3 to node B. Node C then takes a token and has 3 left and 3 upstream neighbors (G, H, I), so sends them each a token. The other nodes repeat this process until they have no more tokens from *A* to send. At the end, *A* has public keys from nodes *B-J*, so it can verify data signed by any of those nodes. Nodes that get a token count of 1 from a node include this fact when they send back their public key. Node *A* then notes that it has a *re-sign flag* for those nodes. The nodes for *A* where this happens are *D, E, J, G, H* and *I.*

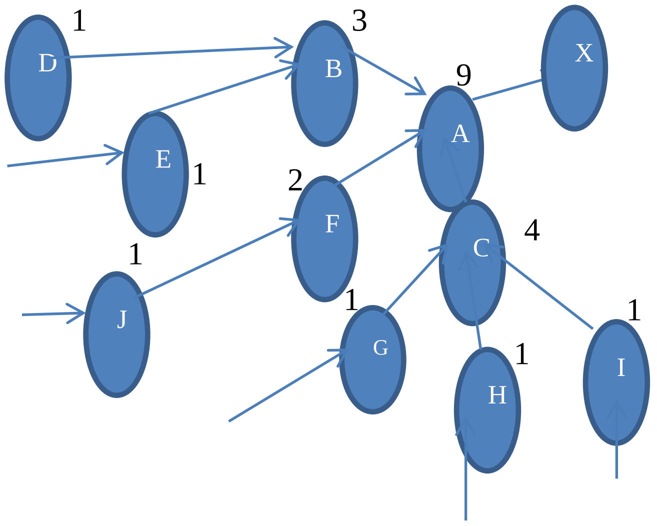

Figure 4 – Re-signing Setup

Remember that other nodes are doing this protocol at the same time. When node *A* forwards keys to node *X* during setup, it notes which ones it forwards that have the re-sign flag set and unsets the flags for those nodes. At the end of the setup phase, node *A* only has the re-sign flag set for nodes for which it did not forward the public key to node *X*. These are the nodes' keys that *A* knows *X* does not have the public keys for, because it did not send them, so it knows that *A* is a re-sign point for those keys. We call the nodes that can verify a particular key the *key span* for that key.

Each node is told how many tokens to send out by the base station. When a node has received public keys for all its tokens it notifies the base station. When all nodes have notified the base station the first phase of the setup is done. The base station then sends a broadcast message indicating the start of the second phase of the re-signing setup. In the second phase, all nodes send a single message to the base station. This message acts like a regular data message. However, each node simply places its node ID in place of the signature fields. When a node sees one of these fake messages it stores the ID_{src} and the IDs for the two signatures into a table, called the *source table*.

This allows each node to build up a table, called the *source table,* of all nodes that route through it toward the base station. This table tells the node, based on the ID_{src}, what the signature on the packet is at the time it reaches that node. This table is needed for many routing schemes anyway in order to route acknowledgements back to source nodes from the base station. We also note that the adversary cannot perform an insider attack during the setup phase of the network. Another advantage of the source table is that, for a given source node, it fixes where the re-sign points are. If a signature is changed by a node that is not a re-sign point for that packet the next signature check will fail, even if the signature is valid, because the node checking the signature will use the key from the node it is expecting to have signed it, rather than the node that did. This means an adversary node cannot pretend to be a re-sign point when it isn't one.

3.4.2. Re-signing: Operation

The operation of the re-signing algorithm is very similar to the naïve solution on which it is based. Data is encrypted using the source key and signed with the source's private key. The data packet that is sent looks like this: packet = $Sig_{src}(D)$ where D is the data defined in Section 3.1.When a node gets a packet, it looks up what

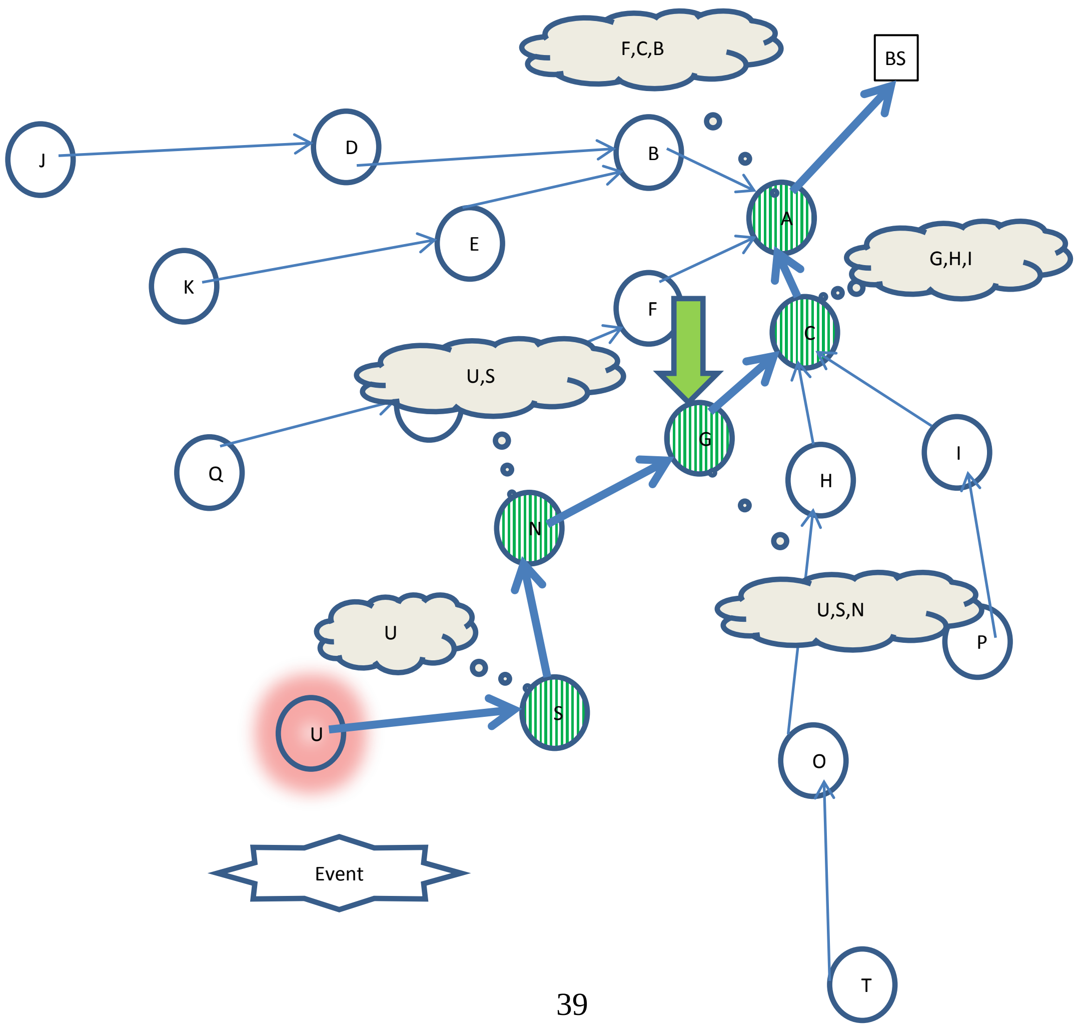

Figure 5 - Re-signing

keys to use in the source table according to the ID_{src} and uses the key specified by the source table to verify the signature on the packet. When we reach the point that the original source node's signature can no longer be verified, that next node calculates a new signature with its private key and puts the new signature in the signature field. In Figure 5, *U* is the source node where the event occurs. The path the packet takes to the base station is marked by the striped nodes. Each striped node is labeled with its public keys in the bubble. Nodes can store up to 3 public keys in this example. The packet starts being signed by node *U*, but we run into a problem at node *G*, marked by an arrow. The problem is node *C* did not have the space to store *U*'s public key, so it does not have it and can't verify *U*'s signature. *G* knows this because it has the re-sign flag set for *U* in the source table a node uses to look up public keys for signature verification. In this case, after verifying *U*'s signature, *G* then signs the packet with its private key. The next node, *C*, does have *G*'s public key so it can simply verify the signature before sending the data onward. In this way the packet makes its way to the base station, getting a new signature whenever needed to check that it has not been altered. As in the naïve solution, a failed signature check indicates altered data and that the preceding node is malicious.

The difficulty with this approach is that, since *G* re-signs the data with its private key, there is no way for the downstream nodes to tell if *G* has altered the data. Using source keys the base station will still be able to detect that the data has been altered, but, except for one special case, it won't be able to tell what node altered it. In the special case of data only being re-signed a single time, the base station can blame

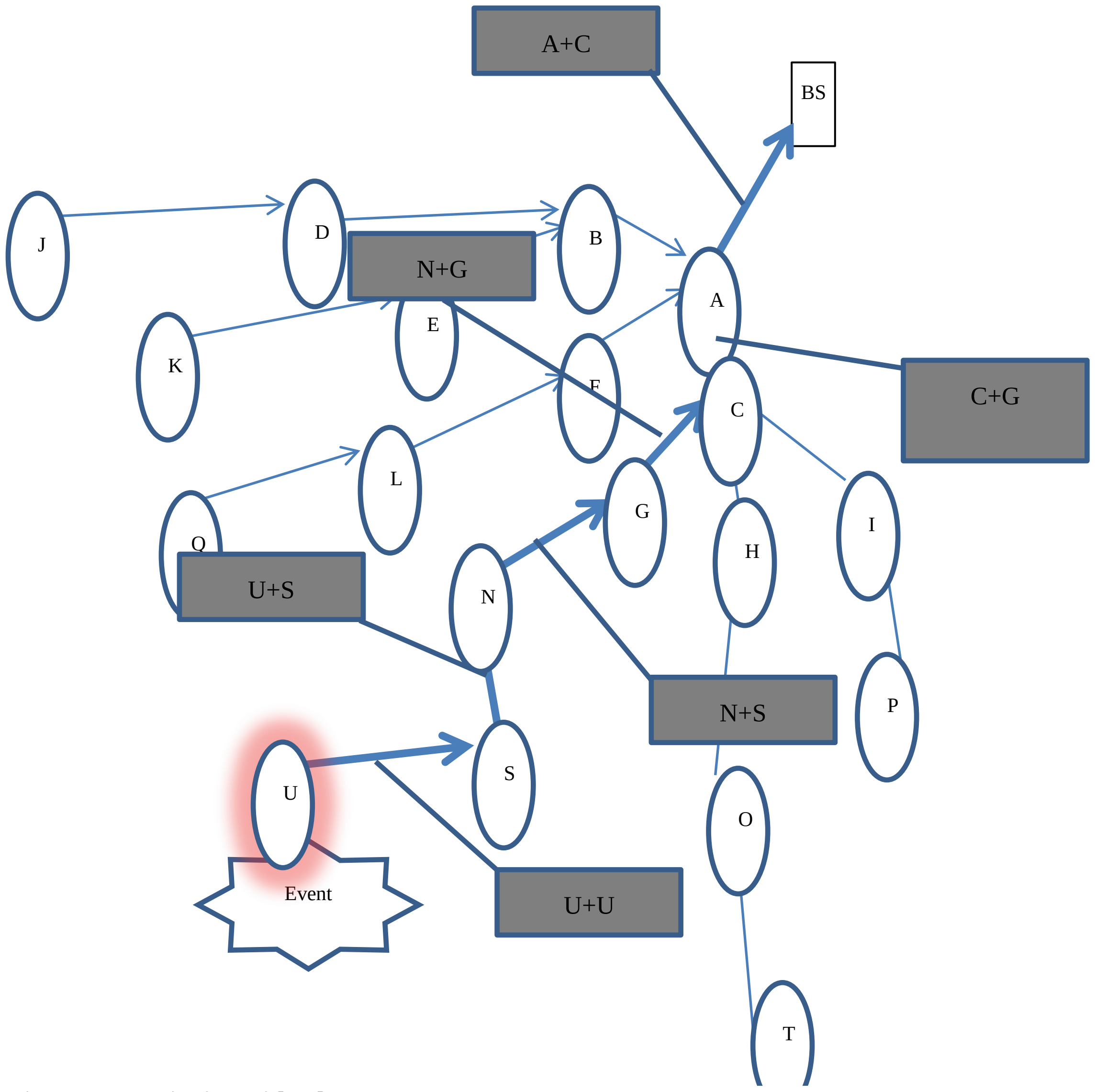

Figure 6 - Re-signing with 2 keys

the node specified in the signature ID field of the packet for the alteration. However, if data is re-signed multiple times on its trip there is no way to know which node did it.

The solution to this attribution problem is to use two signatures, both from the same node. This means we only need the pair of signatures to be unique, rather than the individual signature. We also change the source table to point to two keys per source node, rather than one. By using the source table, nodes know what keys to use to verify the signatures. Each node checks both signatures against the data; if either one fails, it indicates that the data was changed and the previous node altered it.

Figure 6 shows the case where we use two keys in the packet. The key storage space and the keys at each node are the same as in Figure 5. At each hop, we show the names of the nodes that have signed the packet in the call-out box. For simplicity in this example, we will name the signature with the name of the node that signed it. *U* is still the source node. The packet that node S receives has *U* and *U* as its signatures. Since both are the same, *S* replaces one of them with its own signature. *N* then gets a packet with the signatures of *U* and *S*. In this example, as in the previous one, *N* is a re-sign point for *U*, so it signs the data with its private key, replacing the *U* signature with its own signature. *N* cannot take this opportunity to alter the data because the next hop, *G*, can verify *S*'s signature and will be able to tell that the data has been altered when that signature check fails.

We note that this solution fails if there are two adversary nodes in a row that are both re-sign points. This is because two nodes can collaborate to change both signature fields, with the second node changing the data at the same time. Generally, M

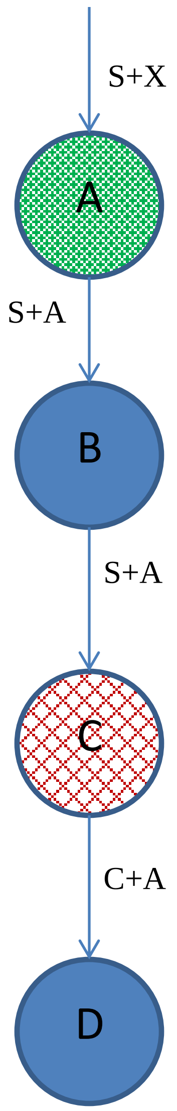

Figure 7 - 2 Signature re-signing

signature fields can deal with M-1 adversary controlled re-sign points, as each re-sign point allows an adversary to replace one signature.

Controlling nodes that are not re-sign points does not help the adversary as the downstream nodes expect the signatures to come from the re-sign points and only use the keys from those nodes to attempt verification, thus other keys are not helpful to the adversary.

3.4.3. Re-signing Evaluation

In the same way as the other solutions based on using public key cryptography, data alterations are detected by the first loyal node that can then conclude that the node that sent the data either altered it or knowingly passed bad data and is thus an adversary. We use the same cryptography system (TinyECC [18]) for the implementation as we did with the naïve solution in Section 3.2.2.

Setup Phase: In the Setup Phase, each of the nodes send a single message with its public key and a list of nodes it got tokens from back to base station. Nodes that forward the message keep the public key, remove their ID from list and send it onward. The node that takes the last ID drops the message after storing the key. The only additional message in the setup phase is the list of IDs.

Message Overhead: Like the naïve solution, there are no additional messages during normal operation. The packet size overhead is the two signature fields, which are 40 bytes each, for a total message overhead of 80 bytes.

Computational Overhead: Data is signed at the source, at the next node and verified at each hop downstream. When it reaches a re-sign point, one of the signatures is updated. Costs for the various operations use the same numbers as in 3.2.2 and are as follows:

S(S) = Signature Generation(also, the time at the source node): 2.00162s

V(S) = Signature Verification: 2.43646s

For routing nodes the total time is the two signature verifications, for a total time of 4.87292s. Nodes that do a signing operation have a total time of 6.87454 seconds.

Memory Overhead: The source of memory overhead is the source table, which stores the two signatures on a packet for a given source. As in the naïve solution, public and private keys are 20 bytes, node ID's are two bytes. We assume 1 byte is needed to index into the table of keys. The size of the table of keys is 20 bytes times the number that are stored. This value can be selected based on the available memory and the needs of the application. Each entry in the source table maps a source node ID to a pair of indexes into the key table. Thus, each source table entry is (1 node ID + 2 key table indexes) = four bytes. There is one source node entry for each upstream node. If *U* is the number of upstream nodes and *T* is the size selected for the key table, then the total memory overhead is *4*U+T*20*. This amount of overhead is a problem because of the resource constrained nature of sensor nodes.

3.5. Tier Naïve

In this section we present a different modification to the naïve solution that vastly reduces the number of keys a node needs to store. This is based on the observation that we only need to keep the same key from being re-used along a path to the base station.

3.5.1. Tier Naïve Setup

To set up this system, the base station figures out what the maximum depth *(D)* of the network is. Depth in this case refers to the number of hops a message takes to reach the base station. Messages from the node with the greatest depth take the most hops. The base station then sends *D* public-private key pairs out in a list. Each node then removes the top key off the list, stores all the rest of the keys in the list in its key table and sends the updated list to all of its upstream children. This ends with each node having the public keys of all nodes further from the base station. Also, all nodes at the same depth, or hop count from the base station, have the same public-private key pair.

3.5.2. Tier Naïve Operation

Consider the fragment of a network shown in Figure 8. The nodes labeled S_1 – S_{I-1} and node A_1 are all the same distance from the base station. This means they share the same public-private key pair. Note that no node that shares a key with node A_1 sends data through A_1. This means that A_1, by *itself,* cannot alter data from any node without being detected. It can alter data from any of the S_* nodes, but they never send data through A_1. Unlike the naïve solution, we run into a problem if there is more than one adversary node. Suppose A_1 decides to send its key to another adversary node, A_2. In this case, then A_2 can alter data from any node that shares a key with A_1 without being detected. In this case, this means any of the nodes with the box around them.

Thus, this solution fails to detect data alteration in a useful manner in the presence of more than one adversary node.

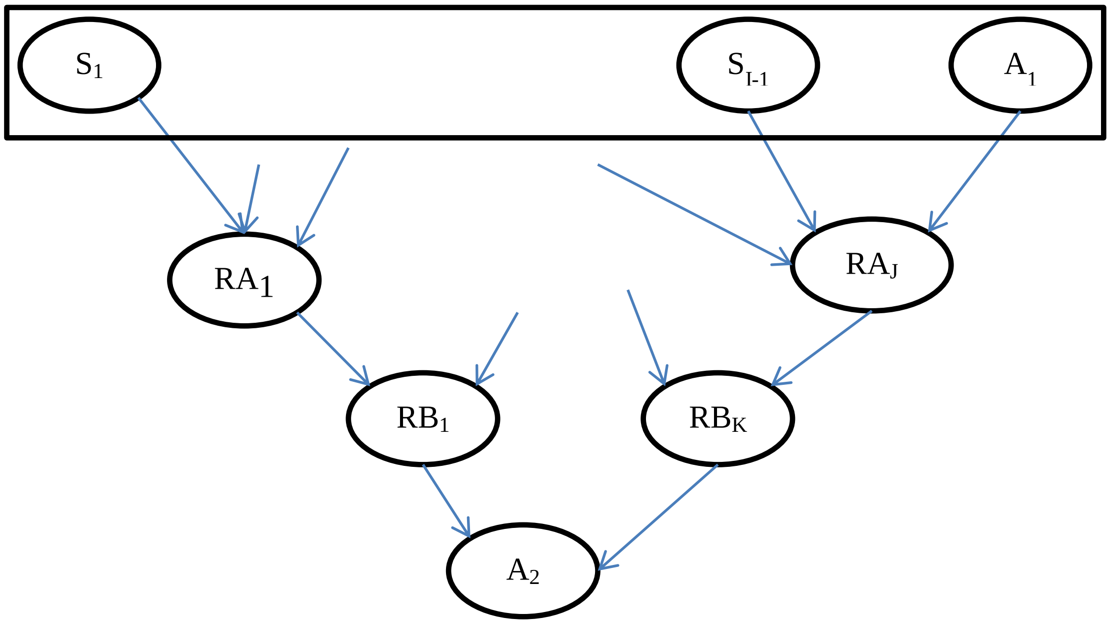

Figure 8 – Tier Naive

3.5.3. Tier Naïve Evaluation

Setup Phase: The base station has to send out a message with the keys. Alternatively, one could send out the seed values used to randomly generate the keys and then send the public keys back to the base station as in the naïve solution.

Message Overhead: The message overhead is exactly the same as the naïve solution, it's just the size of the signature.

Computational Overhead: The computational overhead is also exactly the same as the naïve solution because only a single signature verification is done at each hop.

Memory Overhead: The maximum number of keys any node has to store is equal to the maximum depth of the network, so it is considerably reduced from the naïve solution because the space is proportional to the depth of the network rather than the total size of the network. Space for the key table is exactly the same.

3.6. Tier Plus

In this section, we show how we can improve the tier solution to remove the problem with multiple adversary nodes. We do this by combining it with the resigning solution in a way that reduces the problems of both. The basic problem with the tier solution is, while the number of keys a node needs to store is reduced, the fact that a number of nodes that have the same key means that an adversary that acquires one key can use it to forge data from multiple nodes. Thus, we need a way to limit the damage an adversary can do while keeping the number of keys low. Basically, *we would like the number of nodes an adversary can change data from to be proportional to the number of nodes taken over*.

3.6.1. Tier Plus Setup

A solution here is to use two signatures, as in the re-signing solution. In this case, every node is given 2 sets of public and private keys. Since forging data from a node requires two keys, this means the number of unique combinations is large, while the number of actual keys needed is much smaller. The number of combinations for *N* keys is given by the *binomial coefficient* . If there are *X* nodes and we have *N* keys, we want to make sure $\binom{N}{2} \geq X$. A value of *N=142* would yield 10,011 combinations and is sufficient for a 10,000 node network. To set this up, the base station makes sure no pair is reused when handing out the keys. The one drawback to this solution is that the enemy gets two keys for every node they take over.To reduce this problem we can go with a system similar to re-signing where there are two signatures on the packet but each node has a single key. Since a packet gains its two signatures from its source and first node and first hop after that, we make sure each such pair is unique. The same number of keys can be used, with one exception. To distinguish the two, we append the number of keys per node to the name to get Tier Plus1 and Tier Plus2.

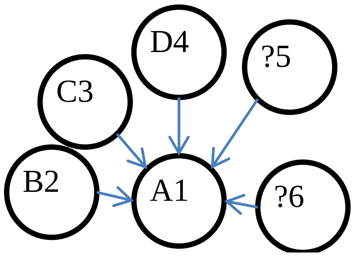

Figure 9 - Tier Plus Bad Case 1

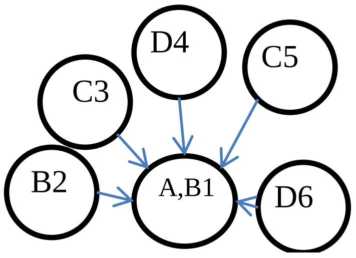

Figure 10 - Tier Plus Bad Case 2

If we have *N* keys and *X* nodes, we run into a problem if a single node has *N* or more children. This is shown in Figure 9. We have four keys and six nodes since $\binom{6}{2} = 4$. In the figures we use letters to represent keys and numbers to represent the node ID’s in this case. While there are enough combinations to give each node a unique pair, we run into a problem because node 1 and key A is the first hop for all of the other nodes. We can solve this by giving node 1 two keys, A and B. It can then pick which key it uses to sign a packet depending on what the source is. If we don’t have this case, we just assign nodes keys that do not reuse an existing combination.

Once the keys are in place, this functions like the re-signing solution. Packets are signed at the source, then at the first hop, and signatures are checked at every hop after that. The fact that the number of keys is limited means we don’t need re-sign points. This solves the difficulty with the re-signing system. The use of two signatures and the number of combinations reduces the problem with the original tier system. In addition, we can allow a single re-sign point in order to further reduce the number of keys. A single point is not a problem because, if the data is changed, the base station knows which nodes to blame as the re-sign point is the only place it could have happened. This requires there be no more than one resign point on any path however.

3.6.2. Tier Plus Operation

The operational aspect of this protocol is the same as in the naïve solution. Data is signed by the source node and then the signature is verified at each hop. As in the naïve solution only a single attacker can be detected per changed packet.

Reconfiguration of the Tier Plus2 protocol is simple, especially since nodes may be able to hold enough keys for the entire network. The base station can send out the required source table changes, signed by it’s private key.

That fact that keys are re-used means that an attacker may gain the ability to alter data from more nodes than they physically control. We will examine this more closely now.

In Tier Plus2, each node has two keys, so an adversary that takes over *n* nodes would have access to $\binom{2n}{2}$ combinations. We can reduce this problem by adding additional keys to draw from a larger pool of combinations, thus reducing the number of combinations that use any one key. Also, we may be able to reduce vulnerability by changing the spacial distribution of keys, i.e. by ensuring that the combinations used on a particular path are distinct in some way. This is an area for future work.

The Tier Plus1 protocol addresses the problem of the adversary using the keys acquired from insider attacks to generate the key combinations of loyal nodes by having only one key per node rather than 2, which reduces the number of combinations an adversary who takes over *n* nodes has the ability to create to $\binom{n}{2}$ because they only acquire one key per node that is successfully attacked.

3.6.3. Tier Plus Evaluation

Setup Phase: The base station has to send out a message with the keys. Alternatively, one could send out the seed values used to randomly generate the keys and then send the public keys back to the base station as in the naïve solution.

Message Overhead: The message overhead is the same as in the resigning solution, there are two signature fields.

Computational Overhead: The computational overhead is exactly the same as in the resigning solution, the difference being that resign points are not necessary.

Memory Overhead: The number of keys needed is bounded by the binomial coefficient. For a network of *N* nodes, the number of keys, *X,* needed is the smallest value such that $\binom{X}{2} \geq N$. As we mentioned before, 142 keys gives 10,011 combinations, which is sufficient for a fairly large network. Space for the key table is exactly the same as in the resigning solution.

3.7. A cryptographic solution

In this section we present a way to modify the naïve solution using more recent cryptographic research to negate its disadvantages. We use what are called *group signatures* [7]. Group signatures are an anonymous version of public key cryptography. In most public key algorithms a user signs data with their private key. The user's public key can then be used to verify the signature. In group signatures there are groups that, effectively, share the same public key, called the *group key*. The group key can be used to verify a signature from any member of the group, but it is impossible to tell *which* member of the group a signature is from using it.

3.7.1. Group Signature Operation

To utilize group signatures, we swap the group signature scheme for the public key signatures in the naïve solution. The base station is the group leader. No other changes are needed. Data is signed at the source with the source's private key and the signature is verified at each hop using the group key rather than the public key of the source node. We do encounter one difficulty with this scheme. The anonymity property of group signatures means that an adversary can change the data and sign it using their own private key and other nodes will not be able to detect the change because they have no way of knowing the signature does not match the source.

The fact that the base station is also the group leader presents a solution to this problem. If an adversary alters data and signs it with their own key, the base station will still be able to detect that the data has been altered, because the data was encrypted with the private source key of the original source node and therefore won't decrypt as described in Section 3.1. Also, due to being the group leader, the base station can extract the signature and discover the identity of the signer. Since we know the data was altered because of the failed decryption and we know the alteration was malicious due to the packet having correct checksums, the base station can then deduce that the signing node is an adversary and take appropriate action. This removes the last drawback of this scheme.

3.7.2. Group Signatures Evaluation

Spreitzer and Schmidt examine several group signature schemes in the context of constrained devices [29]. Their work is invaluable to us here. The scheme that seems to be the most efficient and secure is HLCCN [14]. Signature size is 171 bytes [14]. They also performed benchmarking using a ATmega128 processor. This allows comparison with other algorithms.

Setup Phase: The base station, as group leader, sends a message to every node with their private key and the group key. So there is one setup message per node.

Message Overhead: There are now extra messages, so the overhead is the signature added to each message. The signature size is 171 bytes.

Computational Overhead: The signature is generated once at the source and checked at each node. For this scheme the signature generation time on an ATmega128 is 27 seconds while the signature verification time is ~ 29 seconds. Even using the faster 32 mhz ATxMega256 they suggest that signature generation would take 6 seconds and verification about 7. Hardware support or much faster hardware is needed for most practical applications.

Memory Overhead: The main benefit of this scheme is the low memory overhead. All that nodes need to store is their private key and the group key. Group key size for this scheme is 342 bytes, private key size 86 bytes. Total storage needed is 428 bytes.

3.8. Recovery: A Non-Cryptographic Solution

In this section we examine a solution to detecting data alteration that does not depend on encryption. While the encryption based solutions do provide timely detection of data alteration and can be used to attribute alteration to the attacker easily, public key encryption and the more advanced versions of encryption used have considerable computational loads. Thus, we examine a solution that is based on a different approach to the problem and has a different cost in overhead.

Consider a system using the basic assumptions outlined in chapter 1, but no additional mechanism for detecting data alteration. In this case, the base station can detect that data has been altered, but has a problem detecting who did it. In this section we present an attribution mechanism that can be used once data alteration has been detected.

A simple way to detect data alteration is to remember what you sent and compare that at various points in the network. To do this, we assume that nodes remember a hash of data they received and then sent onward. When they see an acknowledgement from the base station for that data they can forget the hash. If the base station detects that the data has been altered it can, instead of sending an

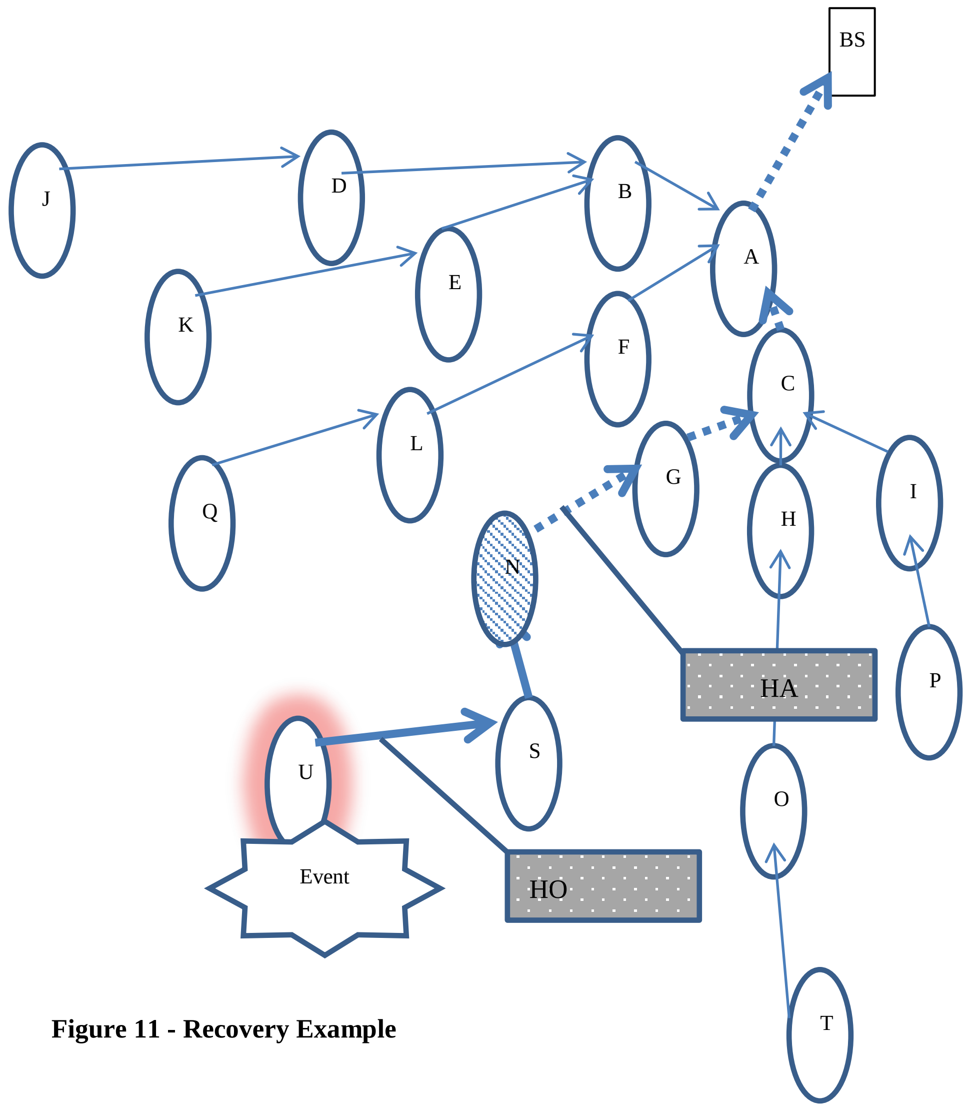

Figure 11 - Recovery Example

acknowledgement, ask for the hashes of the data received at each node along the path. In this way the origin of the alteration can be attributed to a specific node.

3.8.1.1. Recovery Operation

In Figure 11 we can see an example of how recovery would work. *N* is the adversary node. The solid lines represent the unaltered data, while the dashed lines represent the data that have been altered. The original data have the original hash

(*HO*), while the altered data have a different hash (*HA*). A cryptographically secure hash function is chosen so it is effectively impossible for the adversary to make *HO* = *HA*. Nodes remember the hashes of packets that they receive. If the base station detects that data has been changed, it asks that all nodes send their oldest unacknowledged hash for a given source node to the base station. Hashes are sent encrypted with the nodes' source key, so they also cannot be altered by the adversary (and it will be detected if they fail to arrive). We assume the original source node sends an acknowledgement that includes the SHA1 hash of the request of the call for hashes itself with it's source key. This acknowledgement will be missed if the original request for nodes hashes from the base station does not arrive. This prevents the adversary from tampering with the request. However, if the code space for the public key system is not an issue, it is simpler to include the ECC library and use the naïve system to protect these types of messages from the base station. Alternatively, nodes could send the hash of the call they get back along the path they received it from toward the base station. This would also prevent the adversary from tampering with it, as this would obvious from the hashes from each node.

One difficulty is that the adversary can still alter packets sent during the recovery protocol. We note that the base station is able to tell that the data packet has been altered, but not which node did it. We will first consider the case where the adversary does not alter upstream packets. When looking at the hashes received at each node, there will be a point where one node received *HO* and the next received *HA*. Thus, we know the adversary node is one endpoint of that link. It is not easy to say which one however. The problem here is that the adversary node (*N)* can send either *HO* or *HA* as the data they received, which makes it difficult to decide which end of the link with between *HO* and *HA* is the adversary. We consider the 3 node sequence *S->N->G* (where *N* is the true adversary) in Figure 11.If the adversary sends *HA* back as the hash it received then the base station gets *HO* from *S*, *HA* from *N* and *HA* from *G*. This means the adversary is either *S* or *N*.

If the adversary sends back *HO*, then the base station gets *HO* from *S*, *HO* from *N* and *HA* from *G*. This means the adversary is either *N* or *G*. And it is difficult to say which of these two cases is correct. A simple solution is to treat both nodes as guilty. The difficulty of insider attacks means that even being able to kick out an innocent node along with the one that was compromised is not much of a win for the adversary, since the adversary is discovered when it took action. Thus, the amount of damage is limited.

Next, we consider the case where the adversary alters the recovery packets in an attempt to confuse things. Recall that altered hash packets are detected as altered at the base station. The hash from *S* must be altered, as *S* will send *HO* otherwise and having altered packets from before *S* won't change anything if *S* isn't altered.

If the adversary alters upstream packets but sends *HO* as its hash then we get the case where the base station gets something like this: *S={altered}, N=HO, G=HA*. In this case, we know the adversary node is *N*. We know "altered" packets come from nodes upstream of the adversary or the adversary itself, so the only way the original hash could be received from a node downstream of the altered packets is if that node is the adversary. In short, if the adversary is going to alter the upstream packets, it makes no sense for them to send *HO*. The adversary would then either send *HA* or send their own packet as altered.

To think of it another way: The adversary can only alter packets routed through itself. Thus, every packet the adversary could alter and come up as altered at the BS must have been *HO* because only nodes upstream of the adversary route through it. Thus, we can actually treat altered packets as though they contained *HO*. We then end up with a case where there is a link that has an altered hash on one side and *HA* on the other. As in the previous case there is no way to know which node is the adversary, so we treat both nodes as guilty.

While more complicated than other protocols, the result is the same. Malicious data alteration is detected and it is possible to blame a pair of nodes and prevent it from happening again. As with our other protocols, we can only detect a single adversary per changed packet. The first adversary on the retraced path that altered the packet is found.

Reconfiguration of the network is relatively simple in the recovery protocol; there are no keys to distribute. The only requirement is that existing hash paths may no longer be valid. A simple solution would be to flush all hash lists on affected paths and have those sources, who need to keep the data anyway to allow for retransmission, resend them after the reconfiguration. Since we cannot assume the the presence of a

3.8.2. Recovery Evaluation

The overhead of the recovery protocol can be divided up by phase. Normal Operation Phase and Recovery Phase. During normal operation the computational overhead consists of the hash computations. The memory overhead is that required to store the hashes. The exact amount of memory overhead greatly depends on two factors. These are the following:

The Hash Function: Most hash functions create a fixed sized value. Common sizes would be 160 bits (20 bytes) for SHA-1 or 224 bits (28 bytes) for the SHA-3 algorithm selected as its replacement. SHA-1 is a common function used for applications such as SSL certificates.

We note that long term security may not be needed in this case. In order to alter the data undetectably the adversary must find a data value with the same hash as the existing data, which is called a collision. This is considered computationally infeasible for standard cryptographic hash functions. The amount of time they have to do this is until it is noticed that the packet is missing, at which point detection protocols will be triggered. They will be easily detected in this case as nodes before the adversary node will have the hash. In applications where missing packets will be detected relatively quickly we can get away with a slightly less secure hash function as there is still not sufficient time from the adversary to find a collision.

Round Trip Time: Hash values must be stored until the acknowledgement or a call for the hash values comes from the base station.

Depending on various factors, the memory overhead of this algorithm can be significant. There are several ways it can be reduced.

One way to reduce the memory overhead would be to *trigger* the algorithm when an altered packet is detected. A message is sent out that tells nodes to start remembering hashes. This is appropriate if there will be more than 1 packet generated by a given event and losing the first one is not a big loss.

Setup Overhead: There is no setup overhead in this case.

Message Overhead: There is no change in packet sizes. There are also no extra messages during normal operation.

When the recovery protocol is triggered an extra message from each node along the path from the source to the base station is sent with that nodes hash for the message.

Computational Overhead: The computational overhead consists of hash function chosen, 1 hash computation per packet. Ganesan et al analyzed various cryptographic primitives on a variety of common sensor network platforms. We use their results for the ATmega128 for our comparison [12]. While performance varies according to the size of the message, hash time was 7.7 ms on a 64 byte message, and about half of that for a 1 byte message.

Memory Overhead: The memory overhead is that required to store hashes until an acknowledgement can be gotten to discard them. If we assume each cycle includes an upstream and downstream transmission slot and messages are always coming and going at the maximum rate. For a node that is *N* hops from the base station then, a message will take *N* cycles to reach the bases station from a node and *N* more cycles for the acknowledgement to get back. The node must store the hash during this time. During this time, new messages are also coming in. Thus, we expect the number of messages stored at any one time to be proportional to twice the hops from the base station, *2N*. Each hash is 20 bytes(160 bits), so the memory overhead is *40 bytes * N*.

3.9. Hybrid

In this section we present a hybrid between the signing systems and recovery that reduces the disadvantages of both. The disadvantage of the signing systems is the computational cost associated with public key cryptography at each node relative to the hash computation used in recovery. The disadvantage of recovery is the inability to pinpoint a single adversary node as well as the time it takes for an adversary to be detected. We can reduce both these problems by starting off with a signing model and transitioning to recovery as we get closer to the base station. This system is based on the observation that computational and time overhead is more important near the base station because those nodes have to handle more traffic. Thus, we can divide the network into three groups roughly based on depth relative to the base station. The first group is the *signature zone* that uses the naïve protocol where a single signature is checked at each hop. Nodes closest to the base station are in the *recovery zone* and use the recovery protocol. This allows them to handle traffic quickly from the large number of upstream nodes. Finally, nodes between the recovery zone and the naïve zone are in the *hybrid zone* and use a hybrid protocol where some nodes check signatures and some use recovery. What role a node has for a particular packet depends on the source node for that packet.

3.9.1. Hybrid Setup

In setting up the Hybrid protocol there are various parameters that are needed; these will be discussed in turn in this section and explored in greater depth in a later chapter. As in the signature based protocols, all nodes have a public-private key pair. They send their public key to the base station. What the forwarding nodes do with the keys depends on what zone they are in.

The first parameter is the point where we switch from using the signature scheme to using the hybrid scheme. The primary concern for this parameter is the amount of memory available to nodes to store keys. Nodes that would need too much memory cannot be in this zone. Nodes in the signature zone store all public keys they forward during setup.

For the hybrid zone, the main concern is which nodes should use which protocol, as well as where the changeover should be to the pure recovery protocol. In the hybrid zone, nodes divide their memory between recovery and key storage. It is convenient that both ECC public keys and SHA-1 hashes are 20 bytes. Recall that the memory usage of the hash storage for the recovery protocol in a steady state is twice their depth (number of hops from the base station, or *D*). The hash storage size is *2*D*20* bytes, while the rest of the available memory is key storage. Nodes in the hybrid zone use this formula to divide their available memory for this protocol into *hash storage*, used for the recovery protocol, and *key storage*, used to store public keys for checking signatures.

To simplify things, we pick a *step value*. When the public keys are sent to the base station at the end of setup, a counter is included. Every hop toward the base station increments the counter. When nodes in the signature zone get a public key during setup they make a note in their source table. If the counter is a multiple of the step value (*counter mod sv = 0*), the node stores the public key and makes a note in the source table that they are a signature node for that source. That means that they will check the signatures of nodes from that source before forwarding packets. If the counter is not a multiple of the step value the node makes a note in the source table that they are a recovery node relative to that source. This means that the node stores the hash value of packets from that source as part of the recovery protocol.

One can think of the step value, *sv*, as dividing the number of keys a node needs to store by *sv*. Since the base station told each node how many upstream nodes there are, a node can figure out that the number of upstream nodes should be less than the step value times the amount of key storage. This allows each node in the hybrid zone to figure out their step value.

Finally, as the distance to the base station increases and the number of upstream nodes decreases, we enter the *recovery zone*. These nodes use the recovery protocol no matter what the source is. In this zone, it is possible to drop the packet signature to save packet overhead.

3.9.2. Hybrid Operation

Figure 12 shows the setup and operation of the hybrid protocol. We use numbers to label the nodes in this case. We show the operation of the protocol relative to two source nodes, nodes 1 and 2. Along with the node names, we also show the role of each node relative to source nodes 1 and 2 in that order. We use 0 to indicate that a node has no role relative to that source node, and S to indicate the source nodes themselves. The letter G is used to indicate that a node is a signature node relative to a particular source, while R is used to label nodes that do recovery for a particular source. The solid line represents the signature zone. Nodes in this zone check the signature of all packets. Thus, node 3 is labeled 3:GG because it is a signature node relative to both source nodes, 1 and 2. The dashed line represents the hybrid zone for nodes 1 and 2(nodes 3,4 and 5) and the remaining nodes are in the recovery zone for nodes 1 and 2(nodes 7 and 8).

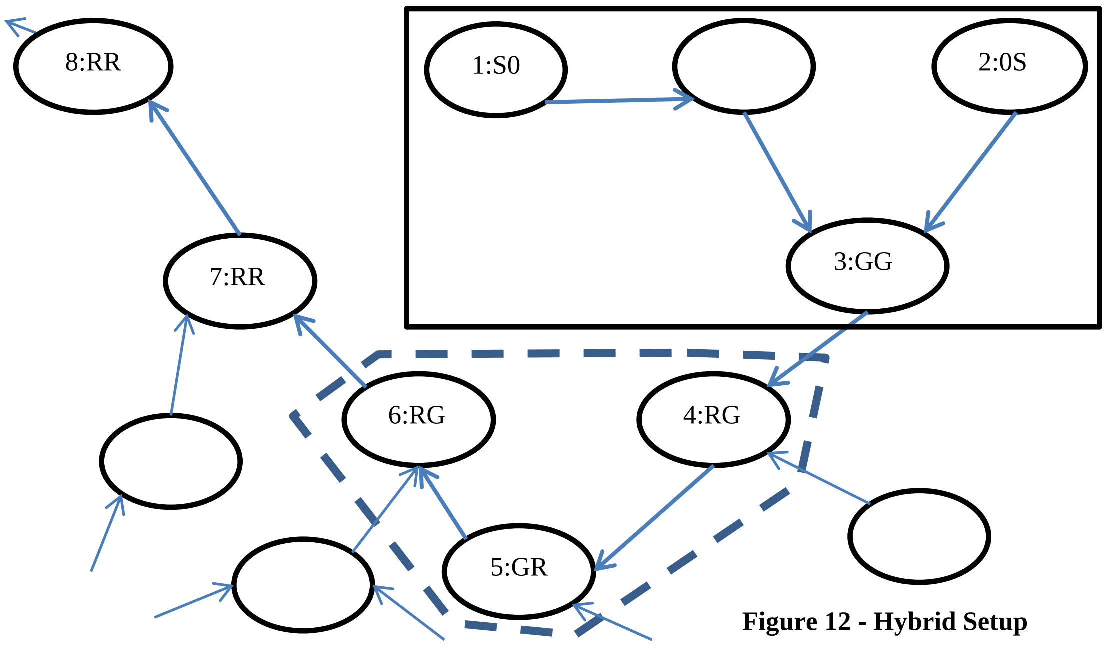

Figure 12 - Hybrid Setup

We note that in the hybrid zone, nodes may have different actions for packets from different source nodes. Also, all nodes in the hybrid zone in this example use a step value of 2. A packet that originates at node 1 would have its signature checked at node 3. Node 4, with the label RG is a recovery node for source 1, so it would store the hash value before forwarding the packet. Node 5 is a signature node for source 1 so it would check the packet signature before forwarding the packet. Node 6 is again a recovery node for source 1, like node 4: it would store the hash value before forwarding. Nodes 7 and 8 are in the recovery zone for all sources, so they store the hash value regardless of the source. In this case, node 7 can remove the packet signature to save packet space.

Attribution and detection are almost the same as in their respective protocols. If a node in the signature zone detects malicious data alteration, it knows the previous node is a fault, as it had to have either altered it or knowingly forwarded a bad packet. This is the same as in the signature based protocols. In summary, if node detects data alteration via a signature check and it knows the previous node also did a signature check, it can assume the previous node is the adversary.

In the Hybrid zone, a node that detects data alteration via a signature check cannot assume the previous node is at fault unless it also did a signature check. If the previous node is a recovery node for that source then the node that detects data alteration initiates the recovery protocol starting at itself. The advantage of doing this is that bad data does not need to go all the way to the base station to be detected and the recovery process is initiated quicker. In the recovery zone, nodes use the recovery protocol as described in an earlier section. As with the other protocols, the hybrid protocol can only find a single adversary per altered packet. This holds true no matter how the alteration is detected or in what zone it is detected.

Reconfiguration of the hybrid protocol is different depending on what zone the affected nodes are in, but in all other aspects, it is the same as in the component protocols.

3.9.3. Hybrid Evaluation

The evaluation of the hybrid protocol depends on the parameters chosen, so we explore it more fully using simulation in later chapter. However we present an overview at this time.

Setup Phase: Like the signature algorithms, the setup phase generates 1 additional message per node that is used to distribute that nodes public key. The message is 1 byte bigger due to the counter.

Message Overhead: The message overhead is almost the same as the naïve solution. The change in packet size comes from the signature on each message. The signature has a size of 40 bytes.

The only extra messages generated are during the recovery protocol. Depending on where the data alteration is detected, the number of extra messages is the same as in the recovery protocol or less. The number of messages is the same if alteration is detected at the base station and less if it is detected by a signature node in the network. During the recovery protocol, a message is sent from every node between the detection point and the source to the base station.

Computational Overhead: The computational overhead varies depending on the role a node plays for a particular packet. If a node is a recovery node for a packet, the computational overhead is the SHA-1 hash check. If a node is a signature node, then the overhead is the signature verification.

Hash Computation: 7.7 ms

Signature Verification: 2.43646s

Memory Overhead: The memory overhead is actually the same for almost all nodes. The difference is how we allocate the available memory between the recovery and signature protocols. The memory storage for recovery is 20 bytes times twice the depth of the node. Public keys are also 20 bytes in size, so we use 20 bytes as the block size for this discussion.

Nodes in the signature zone that have fewer upstream nodes than the amount of memory allocated will use less memory. Similarly, nodes very close to the base station that only use recovery, but have a quick turnaround due to a shallow depth, will also use less memory.

CHAPTER 4

FINDINGS

In this section we will compare the different algorithms across each type of overhead, while discussing the different situations in which each one is useful. The solutions we presented broadly fall into three categories.

Public Key Signatures: These solutions use public key encryption and the signing of data to detect malicious alteration. We consider the Tier Plus system the representative of this category, due to it being a combination of the other systems.

Group Signatures: While the usage of this from a users point of view is almost the same as the public key version, the encryption scheme used is different and not as widely studied. For this reason, it has its own entry. We generally refer to this and the Public Key Signatures as signature based schemes, which differentiates them from the next one.

Recovery: The recovery system is based on the idea of remembering data that is seen and backtracking to determine the culprit. Since this uses a different mechanism than the signature based schemes, we primarily focus on the strengths and weaknesses of the two approaches.

The hybrid protocol uses a combination of the recovery protocol and a signature scheme, which is usually the naïve system. In it, some nodes check signatures and some use the recovery protocol to detect and attribute malicious data alteration. The overheads involved vary according to the source node and the role a particular node plays. We evaluate this empirically in section 4.6.

4.1. Detection and Alteration

It is worth remembering that we can distinguish between packets that have been altered by an adversary and those that have been corrupted by wireless physics. The former will have valid checksums, while the latter will not. This allows us to distinguish *malicious* data alteration.

All signature based schemes work in the same way. One or more digital signatures of the data is added to packets and is checked by all loyal nodes along the path to the base station. A failed check indicates that the previous node is hostile, as we know it either altered the data itself or passed bad data. This makes the detection and attribution of malicious data alteration fairly straightforward in these schemes.

Recovery is based on remembering the data that was sent and back tracking the trail to find the adversary node. The difference is, since the adversary can claim to have gotten altered or unaltered data, we usually cannot uniquely identify the adversary; instead, the alteration can attributed to one of two nodes. We treat both nodes as adversaries in this case. An additional weakness of the recovery protocol is that altered data must reach the base station before it can be detected. This means the effort spent sending it is wasted. Thus, recovery could be considered to be weaker than the signature based schemes in both the detection and attribution of malicious data alteration.

4.2. Setup Overhead

In all cases the setup overhead is fairly straightforward. There is no specific setup for the recover protocol. For the signature based protocols the base station sends each node its key(s). Nodes then send their public keys along the path to the base station to allow nodes along the way to store them.

Tier Plus actually has the worst setup phase of all the public key signature based systems. This is because it allows nodes to have the same public-private key pair as long as repetition does not create packets with the same pair of signatures. This reduces the overall number of keys nodes need to store, but means selection of keys must be coordinated, which requires more data to be sent from the base station. The Hybrid protocol uses the same setup phase as the signing protocol it uses for its signatures (usually, the naïve protocol), so its setup overhead is the same as the naïve protocol.

Individual Data: This is the data sent from the base station to each node that is unique to that node. For a particular node, individual data needed by upstream nodes must be forwarded to them, while that node will never see the individual data for nodes closer to the base station than itself, since they will have already received it. Thus, the total amount of setup data for a node in this category depends on the number of upstream nodes.

Broadcast Data: This is data sent from the base station to all nodes. This data is always forwarded to upstream nodes during setup, but it is the same for everyone.

Public Key Size: In the second setup phase, nodes send their public keys to the base station so downstream nodes can remember it.

We assume the base station sends 64 bit(8 byte) seed values that nodes use to generate their own key pairs where applicable. We also show the sizes for the naïve or re-signing algorithm here, in order to allow comparison with Tier Plus. Algorithms such as the naïve solution allow each node to select its own key individually. These do not send seed values from the base station. Instead of individual public keys, the Group Signatures system broadcasts the group key to all nodes. All sizes are in bytes.

Table 1 - Setup Overhead

Algorithm	Individual Data	Broadcast	Public Key Size
Naïve/Hybrid	0	0	20
Tier Plus	8	0	20
Group Sig	86	342	0
Recovery	0	0	0

In all cases the amount of data sent from the base station to each individual node is less than 100 bytes. Since setup is a onetime phase, no algorithm stands out as having a significant strength or weakness in this area.

4.3. Message Overhead

Extra messages generated due to the protocol and extra data added to packets are both considered message overhead. The only protocol that generates extra messages is the recovery protocol, so we simply discuss that separately before examining the packet overhead of the signature based schemes.

Recovery detects altered data by having nodes remember the values they see and send those values to the base station, encrypted with their own private key, when alteration is detected. This allows the base station to look at the trace of values and determine who the adversary is. Since we assume the base station sends out acknowledgements of data it gets, the call for hash values from each node does not count as overhead, as it takes the place of the acknowledgement. While executing the recovery protocol, one extra message (containing that node's hash value of the message in question) is sent to the base station for every node between base station and the source. In fact, only the values of loyal nodes downstream of the adversary nodes must be received to discover the adversary's (approximate) identity.

Signature based data alteration detection appends signatures to the packet that are verified at each hop. Thus, the amount of data added to each packet depends on the size of the signatures. The three factors we consider here are the number of signatures per packet, the size of each signature and the total data added to each packet. We show the naïve solution (which can be used in hierarchical networks) for comparison.

The Hybrid protocol uses both signatures as in the naïve system and the recovery protocol. Its message overhead is a single signature in the naïve system, but we note that nodes in the recovery zone drop the signature when no more nodes along the path that can verify the signature. All sizes are in bytes.

Table 2 - Message Overhead

Algorithm	Signature Size	# Signatures	Total Size
Naïve	40	1	40
Tier Plus	40	2	80
Group Sig	171	1	171

Recovery	0	0	0

While the group signature algorithm uses only a single signature, the fact that this signature is over 4 times as large makes this system less appealing. This causes the tier plus system to be clear winner in this area.

4.4. Computational Overhead

Computational costs refers to the time spent doing algorithm related computations at each step. In all cases the cryptographic primitives used are the dominant source of computation, with no significant work required outside of these. We are primarily interested in the amount of work done at each hop, for each packet that is forwarded, as these are the dominant costs associated with the system as a whole. We list signature generation times also, but these are only done once signature on the packet, so their contribution to the overall work required is fairly small.

We call the primary operation for each algorithm the *primitive operation* with the total computation overhead being the product of the magnitude of each primitive operation as well as how many there are per packet.

For Recovery, the primary source of computation is computing the hash value of the packet and storing it. For the signature based systems the dominant factor is the signature verification. These constitute the primitive operations for those algorithms.

All operations that are only done a single time per packet are added to the startup time. For signature algorithms, this is the time to sign the data.

The hybrid protocol uses both signatures and recovery, where weather a given node checks a packet signature or stores a hash value for recovery varies according to the role that node plays for that source node. Thus, we evaluate it more globally in a later section.

Table 3 lists the time for the primitive operation for each system, the number of primitive ops per packet and the total startup overhead per packet and total computational overhead per packet per hop. Startup overhead is done once per packet and isn’t part of the total. We once again include the naïve algorithm for comparison. All times are on an ATmega128 microprocessor, a common sensor network platform and are in seconds.

Table 3 - Computational Overhead

Algorithm	Primitive Op	#	Startup Time	Total Time
Naive	2.436	1	2.002	2.436
Tier Plus	2.436	2	4.004	4.872
Group Sig	29	1	27	29
Recovery	.0077	1	.0077	.0077

Due to the fact that hash operations are much faster than public key operations the recovery protocol is the best here.

4.5. Memory Overhead

In this section we examine the amount of memory used by the different algorithms. We assume all algorithms need a basic form of the source table in order to route acknowledgements and other control data to upstream nodes. Thus, one category of memory use we consider is the amount of data added to the source table. This is the number of bytes each entry is increased by. This data takes form of pointers into the key table. We assume 1 byte is sufficient for each pointer, as 255 keys tend to take up a decent amount of space for all the algorithms we consider. In addition the Tier Plus algorithm can deal with over 10,000 nodes using fewer entries than this.

The other category of memory consumption is the amount of data used for *detection table (DT).* This is space dedicated to information used to detect data alteration. For signature based algorithms this is the space used for keys, while for the recovery algorithm it is the space used to store hashes. Since the memory of the algorithms is topology dependent we consider the node with the highest memory usage in a given topology. This is because this is often the limiting case that determines if a system is practical. We also consider worst case topologies for all cases. For completeness, we also mention the space used by each node's private key, although each node only has one.

We will use the following variables:

N**:** Number of nodes in the network.

X**:** The smallest integer value such that $\binom{X}{2} \geq N$

U**:** The number of upstream nodes. The worst case for this value is the neighbor of the base station that the most nodes route through. Generally, we expect the base station to have more than 1 neighbor, so *U* can be conservatively approximated as 60% of *N*.

D:Depth of the network. The worst case for this parameter is the node with the longest path to the base station.

The categories we consider are the private key size, the amount of overhead per Source Table entry(ST Overhead), the number of entries in the detection table(DT

Size), the size of each DT entry and the total memory used by the detection table(DT total). All sizes are in bytes.

Table 4 - Memory Overhead

Algorithm	Private key	ST Overhead	DT Size	DT Entry Size	DT Total
Naïve	20	1	U	20	20*U
Tier Plus	20	2	<X	20	20 * X <
Group sig	86	0	1	342	342
Recovery	0	0	2*D	20	40 * D

One thing we notice is that the ECC public keys and SHA-1 hashes used in recovery are both 20 bytes. This helps make comparisons clearer. The clear best version here is the group signatures, with a total space usage of 428 bytes, which is enough for 21 keys or a network depth of about 10. This value does not change according to the size of the network.

Between the Recovery and Tier Plus algorithms, we can that the memory usage of Tier Plus increases faster than recovery. To simplify things, we will assume that the network is a balanced binary tree. This means the depth is $log_2(N)$, despite each node having only two children. For 100 nodes, the network would have maximum depth of $log_2(100)=6.64$ or 7. For 100 nodes, $X = 15$, since $\binom{15}{2} = 105$. For 1000 nodes then $X=46$, $\binom{46}{2} \geq 1035$, while $log_2(1000)\sim 10$. As we can see the difference in growth means that recovery will almost always use less memory than the other systems.

4.6. Hybrid Evaluation

On a high level the signature based protocols could be said to provide an excellent ability to detect adversary activity, at the cost of a high computational overhead due to the public key operations. The recovery protocol is the other way around. The computational overhead at each node is a fairly lightweight single hash computation, but the recovery protocol itself involves a number of trips through the network in order to gather the hashes necessary for identification so the amount of time before an adversary can be identified is increased. Adversary attribution accuracy is also reduced.

The hybrid protocol uses a combination of the other protocols, so we evaluate it in this section. On a per-node level, the overheads of the hybrid protocol are the same as the other protocols depending on what role that node plays, so we focus on more global metrics in this section.

To evaluate the hybrid protocol we implemented a simple MatLab [20] based simulator that simulates a sensor network in a tree configuration with a basic TDMA[26]-like protocol where time can be divided up into slots. Each node sends and receives a single upstream and downstream message in each slot. The metrics we evaluate are:

Average Computational Overhead: We define the average computational overhead per source as the total computation time spent by nodes processing the packet on its way to the base station divided by the source node's depth. The average computational overhead for the network is the average over all possible sources, which is all nodes in this case. The numbers are for the ATmega128 as in the rest of this section.

Average Attribution Time: We measure the average amount of time until the base station can be notified of an adversary. For the recovery protocol, this is the time taken by the recovery phase until the required hashes can be gathered at the base station. For

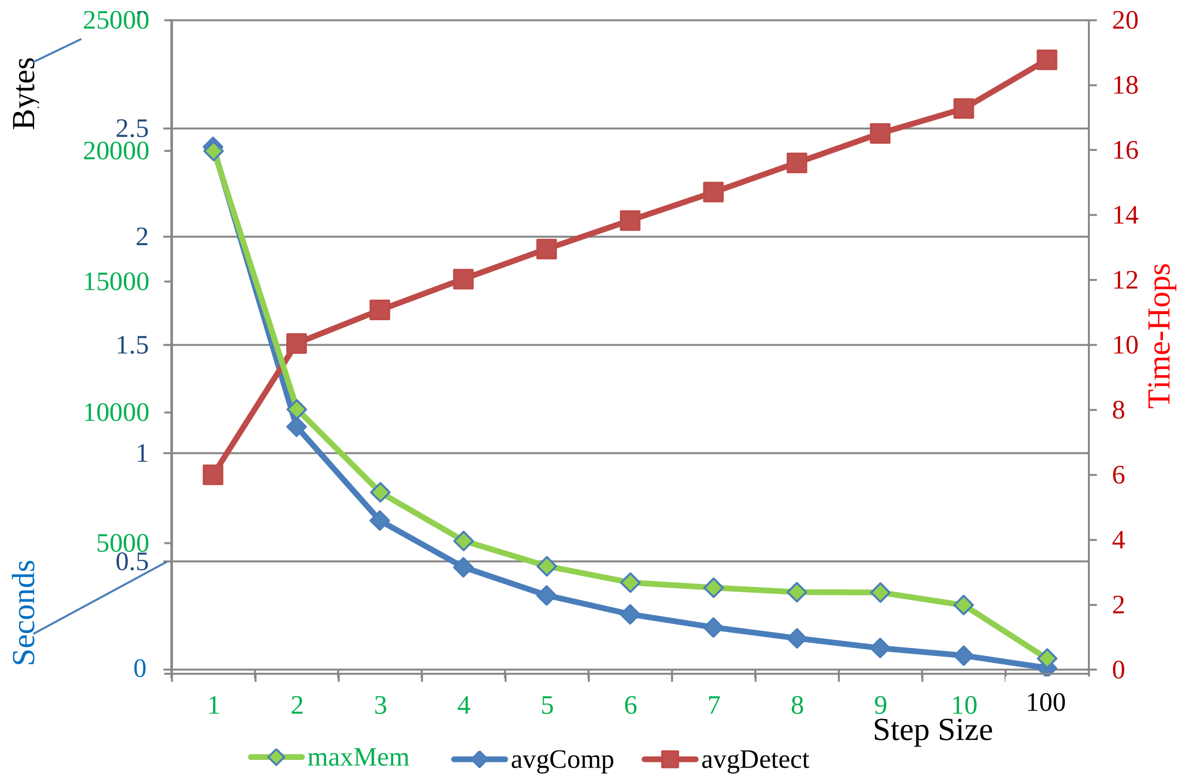

Figure 13 - Step Hybrid Performance

signature based detection protocols the detecting node just notifies the base station.

We define the average detection time per source as the detection time average over all possible adversaries relative to that source. Recall that only nodes that route that source's packets will have the opportunity to change them as an adversary. The attribution time for the network is averaged over all possible sources.

Max Memory Overhead: We evaluate the maximum memory overhead, rather than average. Since sensor networks are often homogenous, the maximum usage by any single node is the limiting factor.

4.6.1. Hybrid Step Evaluation

First we evaluate the step protocol used in the hybrid zone of the hybrid protocol. Here, nodes decide whether or not to store a given node's public key during setup based on whether the hop count from that node is an even multiple of the step. Nodes are called signature nodes for sources where they have a public key and recovery nodes for all other sources. Nodes store hashes and participate in the recovery protocol for all sources they don't have a key for. Since there is no standard topology for sensor networks, to evaluate the step protocol we generate random 1000 node network trees where each node may have a maximum of 5 children. We evaluate the computational overhead, attribution time and memory overhead while varying the step size between 1 and 10. We also include a step size of 100, which, being larger than the network depth, is equivalent to the pure recovery protocol. We note that a step size of 1 is equivalent to the naïve protocol.

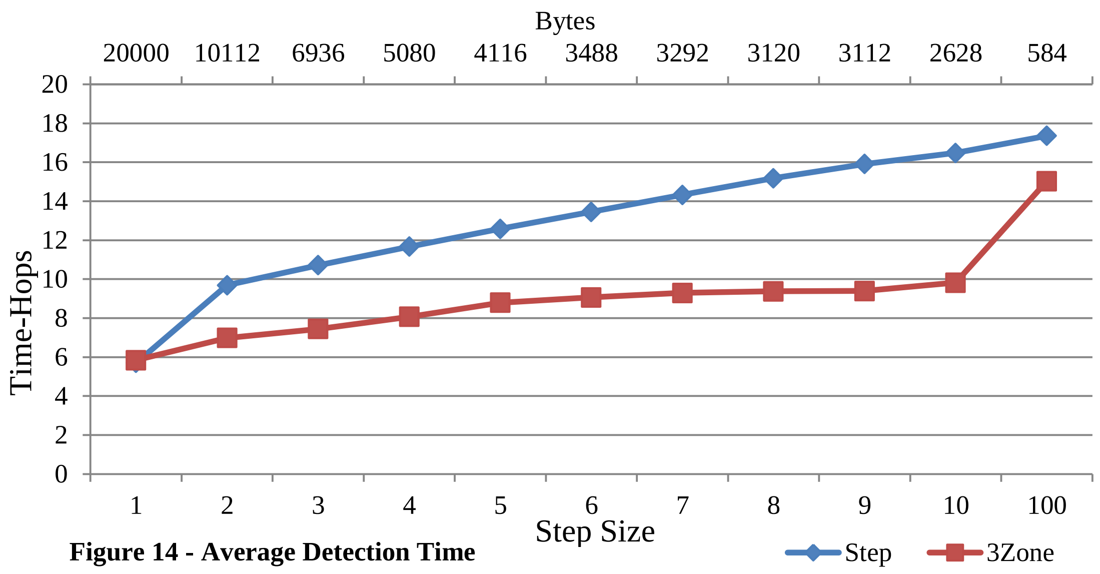

Figure 14 - Average Detection Time

Figure 13 shows the operation of the step portion of the hybrid protocol as the step value is varied from pure signatures at step of 1 to pure recovery protocol at a step value of 100. The most interesting feature is that the detection time rapidly increases with the step value, while the computational overhead goes down as the number of nodes doing signature checks decreases.

4.7. Zone Hybrid Evaluation

The zoned version of the hybrid protocol divides the network into specific zones in order to achieve both a target memory usage and better detection times. To evaluate the effects of zoning we use the same networks we used to evaluate the step version while setting the memory target to be the max memory usage of the step version. Nodes divide their memory up between different roles and won’t exceed the target value.

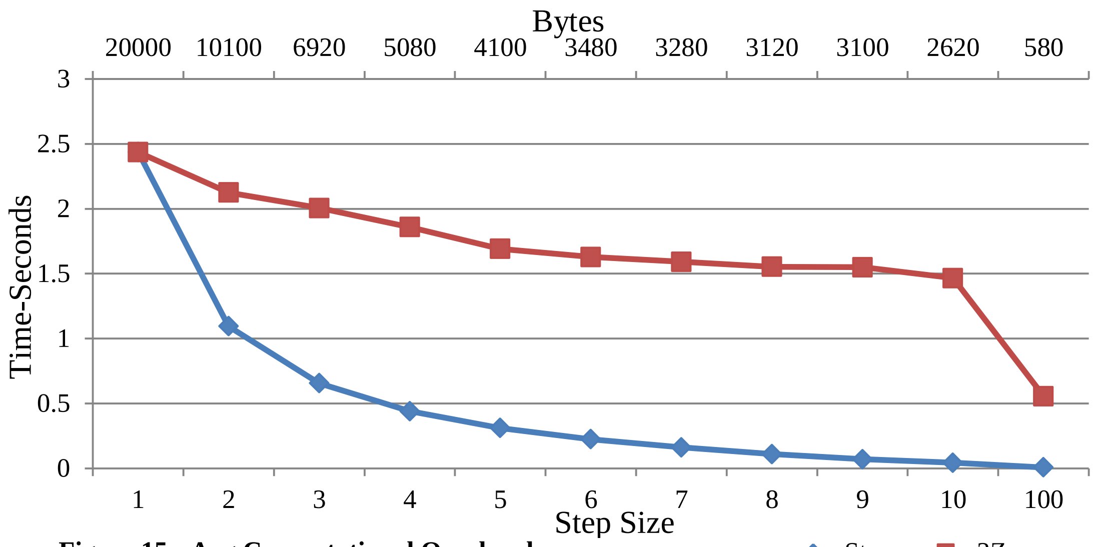

Figure 15 - Avg Computational Overhead

In Figure 14 we see the average detection time of the zone hybrid vs. the basic step algorithm. We placed the memory usage along the top axis, since both protocols use the same amount of memory. In Figure 15 we see the average computational overhead of the zone protocol compared to the basic step protocol. The zoned hybrid generally uses more sign checks, which reduces detection time but also increases the computational overhead. The zone protocol tends to use sign nodes near the leaves, so the extra time is less of an issue due to the fact that these nodes do not have to route as much traffic. In certain applications, such as military ones, nodes further from the base station may be considered more likely to be compromised also, so that having more checking in those areas may make sense. We see that by tuning the memory consumption of the zone protocol we can affect the number of sign check nodes which in turn affects both the detection time and computational overhead. This provides a way of tuning the tradeoff toward a specific application domain.

CHAPTER 5

CONCLUSION

In this section we tie together the results of the previous sections as well as expanding on situations where one makes sense compared to the other. Much like previous chapter, we mostly concentrate on three schemes while mentioning others as appropriate. The Tier Plus and Group Signatures are the representatives of the signature based systems while Recovery is the non-cryptographic representative.

5.1. Experience and Principles

In this section we talk about experiences and principals we discovered while trying to address the problem of malicious data alteration in sensor networks. It is hoped that these may be of use to future research.

Due to attacking the most general case of an adversary node that was a part of the network the options for detection were limited because the adversary could perfectly imitate a loyal node. This meant we could only detect malicious nodes by their behavior. In this case an adversary is detected when they alter data. We assume all nodes have equal difficulty for the adversary to attack, so that the only barrier we can raise is the *number* of nodes that need to be taken over.

The concept of a secure channel between each individual node and base station turned out to be very useful. This was realized by the source key in our system. It not only settled the question of “how does one know something changed”, it also allows one to distinguish between malicious data alteration and that caused by wireless transmission, as well as providing a simple way to secure necessary control communications.

The signature based schemes are all based on the idea of verifying the data based on the signature of the original source. This is because the source is the only time we can be sure the data is not corrupted. The primary problem to overcome here was the number of keys required for this. Any sort of key reuse is problematic due to the adversary's ability to take over any node. Re-signing similarly proved problematic due to the re-sign points being shared points of vulnerability**.** The nice thing about using signatures was the simplicity of detection and attribution. An asymmetric scheme is required as an symmetric one would give the key to the adversary.

The inverse of this was the recovery protocol, which depended on there being a loyal path of nodes back to the adversary. By tracing that path, one could almost determine where the change was. The difficulty is that there was no way to force the adversary to send either the value they received or the value they sent, so we would always end up with two potential suspects. The performance benefits of this generally seem to outweigh being able to narrow the adversary down to one node rather than two.

5.2. Algorithm Evaluation

Here we make an overall evaluation of each of the algorithms we considered along with details of a particular application that influence the choice of for a specific application. As with the evaluation section, the three main protocols we consider are

Group Signatures, Tier Plus and Recovery. The first two algorithms are signature based, while Recovery is not.

5.2.1. Group Signatures

The Group Signature based algorithm is easily the least useful of the solutions we found. Since it functions like a public key algorithm with a shared public key, the memory overhead is quite attractive. In addition, its ability to detect data alteration is also among the best. It is also quite insensitive to the properties of the network other than to the degree its overheads cause a problem. While the amount of memory required is quite low, the computational overhead is severe. It is almost 5 times the overhead of the elliptical curve based signature schemes, even accounting for the fact that there are two signatures. In addition, the fact that the keys and signature are both larger than the ECC schemes reduces its memory advantage. This makes it difficult to recommend this scheme.

We did not originally expect this to be the case. Group Signatures are a type of pairing based cryptography [5] that uses discreet logarithm problem and bilinear maps between a pair of cyclic groups of large prime order. Type-1 pairings have both groups be the same. An advantage of this is that some optimizations can be used to speed of the calculations considerably. The problem is that some recent attacks [15] [13] on the discrete log problem in these situations have cast doubt on the security of commonly used parameters here. This makes it impractical to use systems based on these types of pairings. (The Group Signature algorithm we evaluated used a different type of pairing.) The problem comes from the fact that the algorithm is almost five times slower with the optimizations available to type-1 pairings. This makes it difficult to recommend this system except in cases where memory is severely limited while the computational power is not.

5.3. Related Work

This work falls under a general umbrella of work that endeavors to address the problem of security in wireless sensor networks. This research is complementary to many of these approaches. The concept of a “watchdog”, where nodes attempt to listen to their neighbors transmissions, as well as the limitations of this approach were discussed in Chapter 2.

This work deals with the problem of nodes corrupting data that passes through them, and does not address the problem of malicious nodes creating false data out of thin air. This is also called “false data injection.” In general, solutions to this problem use sensor redundancy within the network. If a node detects an event, then its neighbors should also detect something. This can be used to distinguish between real events and fake ones [3]. This is orthogonal to the problem we address, which involves preventing nodes from tampering with data they forward. In a real sensitive application, both problems would need to be addressed.

An insider attack allows an adversary to generate data that appears valid to my method, while this method does not address nodes altering forwarded data as all reports related to a single event could route through a single node at some point and thus could be subject to an alteration by an adversary. This would allow an adversary routing node to generate the appearance of consensus among a local neighborhood reporting an event.

This work deals with a specific attack and attribution of blame for it. This information would then be reported to higher level Intrusion Detection System[23], which can then take action. This research is complementary to such systems as it can identify the targeted attack and culpable node with a high degree of accuracy. In addition, we also discussed the idea of triggering a defensive technique in the face of an elevated threat level or if data alteration is detected at the base station.

5.4. Wireless Sensor Network Application Background and Conclusion

In order speak about differences between the remaining two algorithms, some background knowledge of the different types of operational modes is required. Since this is the only place it is required, we speak of it here. [26] We also mention which of the schemes is better in this case.

Multiple Base Stations: We usually assumed a single base station for simplicity. We assume all nodes can route to any base station, as if nodes are divided between base stations they can be treated as separate networks. Multiple base stations imply multiple routing trees. This increases the overhead of algorithms that rely on the routing tree.

Mobility: Even worse is if the nodes can move. In this case, routing is a much more difficult problem. In addition, nodes may frequently be disconnected and reconnected to the network. In this case, we recommend solutions that do not depend on routing tree at all. Recovery cannot be used in this case, as there will be no path to trace back.

The two key version of Tier Plus (where each node has two keys and only the pairs are unique) is the best solution here, as any node can verify data signed from any source in this case with a fairly reasonable number of keys stored.

Mobile sinks are another option that can be considered. There is the additional possibility that they could be compromised [34]. This could be considered an area of future work.

Duty Cycle: In order conserve power sensor nodes will often sleep. The pattern of this is referred to as the *duty cycle*. This may cause a node to have to sleep due to the fact that its neighbors are not awake at the right time. While sleeping the node must store whatever data it needs. Unless the network mostly sleeps as group, signature based schemes are favored in this case. This is due to an important difference in the memory usage between the signature and recovery systems. The keys stored by signature schemes do not change often, so they can be stored in the node's FLASH memory, which is also used for program data. The hash values stored by recovery change often and must reside in the node's much more limited RAM, which must be either powered or written to more permanent storage (writing to FLASH memory usually uses much more power than reading it) when a sleep cycle starts. Thus, if nodes do not coordinate their wake cycles, the signature based Tier Plus scheme is preferred. This is one of the few cases where Group Signatures could be useful, as nodes could stay awake doing the required computations. However, the energy cost of this would still tip the scales toward Tier Plus unless storage was really at a premium.

Data Generation Model: Different applications generate different patterns of data. In some cases, events may cause “bursts” of data from areas of the network, while in other cases there may be a relatively constant stream of events. Combinations of those two extremes are possible. As the lowest overhead system, Recovery works best for the burst data model. The one concern is the degree to which the recovery protocol would cause interference if triggered. Thus, the signature protocols and their immediate detection and attribution aspect have an advantage.

In general, we find there to be no single best algorithm, but the recovery protocol offers the best performance by far if its minor weaknesses in detection and attribution can be tolerated. If these are a problem, a variant of the Tier Plus algorithm provides robust detection and attribute of malicious data alteration with only modest overheads, as well as not being affected by factors such node or base station mobility. Finally, the zone hybrid algorithm provides a way of managing the tradeoff between detection time, computational overhead and memory usage. In the future we would like to enhance the Tier Plus algorithm with a more quantitative analysis of the risks of multiple adversary nodes and key combinations while examining the idea of reducing these by controlling the spacial distribution of keys and combinations in the network. We note that the simple numeric approach we used provides good security unless the number of nodes is close to the number of combinations and that the risk can be reduced by increasing the number of combinations. We would also like to further cutting edge cryptographic techniques and further examine the turning parameters for the hybrid system to see it can provide tradeoff’s along different axies.

BIBLIOGRAPHY

[1] AKYILDIZ, I. F., SU, W., SANKARASUBRAMANIAM, Y., AND CAYIRCI, E. Wireless sensor networks: a survey. *Computer Networks 38* (2002), 393–422.

[2] AL-KARAKI, J. N., AND KAMAL, A. E. Routing techniques in wireless sensor networks: a survey. *Wireless communications, IEEE 11*, 6 (2004), 6–28.

[3] ATAKLI, I. M., HU, H., CHEN, Y., KU, W. S., AND SU, Z. Malicious node detection in wireless sensor networks using weighted trust evaluation. In *Proceedings of the 2008 Spring simulation multiconference* (San Diego, CA, USA, 2008), SpringSim '08, Society for Computer Simulation International, pp. 836–843.

[4] ATMEL_CORPERATION. Atmega128. http://www.atmel.com/devices/atmega128.aspx. visted on 2014.02.27.

[5] BARRETO, P. S. L. M. The pairing-based crypto lounge. http://www.larc.usp.br/ pbarreto/pblounge.html. Vistied on 2-24-14.

[6] BECHER, E., BENENSON, Z., AND DORNSEIF, M. Tampering with motes: Real-world physical attacks on wireless sensor networks. In *in 3rd International Conference on Security in Pervasive Computing (SPC* (2006).

[7] BONEH, D., BOYEN, X., AND SHACHAM, H. Short group signatures. In *Advances in Cryptology–CRYPTO 2004* (2004), Springer, pp. 41–55.

[8] CERTICOM_RESEARCH. Standards for efficient cryptography, sec 1: Elliptic curve cryptography. version 1.0, 2000. http://www.secg.org/collateral/sec1 final.pdf, 2000. Visited on 2014.03.02.

[9] CERTICOM_RESEARCH. Standards for efficient cryptography (sec) 2: Recommended elliptic curve domain parameters, version 1.0 edition. http://www.secg.org/collateral/sec2_final.pdf, 2000. Visited on 2014.03.01.

[10] CHEN, X., MAKKI, K., YEN, K., AND PISSINOU, N. Sensor network security: a survey. *Communications Surveys & Tutorials, IEEE 11*, 2 (2009), 52–73.

[11] CROSSBOW_TECHNOLOGY_INC. Micaz mote. http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf. Visited on 2014.02.27.

[12] GANESAN, P., VENUGOPALAN, R., PEDDABACHAGARI, P., DEAN, A., MUELLER, F., AND SICHITIU, M. Analyzing and modeling encryption overhead for sensor network nodes. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications* (2003), ACM, pp. 151–159.

[13] GÖLOGLU, F., GRANGER, R., MCGUIRE, G., AND ZUMBRÄGEL, J. On the function field sieve and the impact of higher splitting probabilities. In *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 109–128.

[14] HWANG, J. Y., LEE, S., CHUNG, B.-H., CHO, H. S., AND NYANG, D. Short group signatures with controllable linkability. In *Lightweight Security & Privacy: Devices, Protocols and Applications (LightSec), 2011 Workshop on* (2011), IEEE, pp. 44–52.

[15] JOUX, A. A new index calculus algorithm with complexity l (1/4+ o (1)) in very small characteristic. *IACR Cryptology ePrint Archive 2013* (2013), 95.

[16] KARLOF, C., SASTRY, N., AND WAGNER, D. Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems* (2004), ACM, pp. 162–175.

[17] KRISHNAMACHARI, L., ESTRIN, D., AND WICKER, S. The impact of data aggregation in wireless sensor networks. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on* (2002), IEEE, pp. 575–578.

[18] LIU, A., AND NING, P. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on* (2008), IEEE, pp. 245–256.

[19] MARTI, S., GIULI, T. J., LAI, K., BAKER, M., ET AL. Mitigating routing misbehavior in mobile ad hoc networks. In *International Conference on Mobile Computing and Networking: Proceedings of the 6 th annual international conference on Mobile computing and networking* (2000), vol. 6, pp. 255–265.

[20] MATLAB. *version 8.3.0 (R2014a)*. The MathWorks Inc., Natick, Massachusetts, 2014.

[21] MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of applied cryptography*. CRC press, 2010.

[22] MHATRE, V., AND ROSENBERG, C. Design guidelines for wireless sensor networks: communication, clustering and aggregation. *Ad Hoc Networks 2*, 1 (2004), 45–63.

[23] MISHRA, A., NADKARNI, K., AND PATCHA, A. Intrusion detection in wireless ad hoc networks. *Wireless Communications, IEEE 11*, 1 (Feb 2004), 48–60.

[24] OLSZYNA, J., AND WINIECKI, W. Low-power cryptographic coprocessor for autonomous wireless sensor networks. In *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2013* (2013), International Society for Optics and Photonics, pp. 890327–890327.

[25] PERRIG, A., STANKOVIC, J., AND WAGNER, D. Security in wireless sensor networks. *Communications of the ACM 47*, 6 (2004), 53–57.

[26] RAGHAVENDRA, C. S., SIVALINGAM, K. M., AND ZNATI, T. *Wireless sensor networks*. Springer, 2004.

[27] RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126.

[28] SOHRABI, K., GAO, J., AILAWADHI, V., AND POTTIE, G. J. Protocols for self-organization of a wireless sensor network. *IEEE personal communications 7*, 5 (2000), 16–27.

[29] SPREITZER, R., AND SCHMIDT, J.-M. Group-signature schemes on constrained devices: the gap between theory and practice. In *Proceedings of the First Workshop on Cryptography and Security in Computing Systems* (2014), ACM, pp. 31–36.

[30] SZCZECHOWIAK, P., OLIVEIRA, L. B., SCOTT, M., COLLIER, M., AND DAHAB, R. Nanoecc: Testing the limits of elliptic curve cryptography in sensor networks. In *Wireless sensor networks*. Springer, 2008, pp. 305–320.

[31] TANENBAUM, A. S. *Computer Networks 4th Edition*. Prentice-Hall Englewood Cliffs (NY), 2003.

[32] WANDER, A. S., GURA, N., EBERLE, H., GUPTA, V., AND SHANTZ, S. C. Energy analysis of public-key cryptography for wireless sensor networks. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on* (2005), IEEE, pp. 324–328.

[33] WOO, A., TONG, T., AND CULLER, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems* (2003), ACM, pp. 14–27.

[34] ZHANG, W., SONG, H., ZHU, S., AND CAO, G. Least privilege and privilege deprivation: towards tolerating mobile sink compromises in wireless sensor networks. In *Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing* (2005), ACM, pp. 378–389.