SRI International

SDL Technical Report SRI-SDL-04-02 • April 6, 2004

Lightweight Key Management in Wireless Sensor Networks by Leveraging Initial Trust

Bruno Dutertre
Steven Cheung
Joshua Levy

This research is sponsored by DARPA under contract number F30602-02-C-0212. The views herein are those of the authors and do not necessarily reflect the views of the supporting agency.

System Design Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

We present a novel approach for key management in wireless sensor networks. Using initial trust built from a small set of shared keys, low-cost protocols enable neighboring sensors to authenticate and establish secure local links. As the risk of sensor compromise increases with time, the keys are used only for a limited period right after deployment. Once secure local links are established, other security services such as group-key refresh can be provided. The protocols we present require little memory and processing power, and require a small number of shared keys independent of the network size. Moreover, these protocols do not depend on a trusted server or base station. To validate the applicability of our approach to ad hoc wireless sensor networks, we have implemented our protocols on the TinyOS-based Mica platform and applied them to secure a perimeter monitoring application.

Contents

List of Figures

List of Tables

1 Introduction

Networks of wireless sensors present a cost-effective solution to a range of applications in critical domains such as detection of chemical or biological agents or tracking of enemy vehicles. In these critical applications, using incorrect or maliciously corrupted data can have disastrous consequences. Security services are essential to ensure the authenticity, confidentiality, freshness, and integrity of the critical information collected and processed by such networks. To support these security services, one needs entity authentication and key management that are resilient to external attacks against these networks and to failure or compromise of these sensors.

If all sensors have sufficient memory and processing power, approaches based on public-key cryptography or on the Diffie-Hellman key exchange protocol may be applicable, but the necessary cryptographic primitives are currently too expensive for the most resource-constrained devices. Less costly alternatives that employ trusted servers sharing a long-term secret with each client are available. However, such approaches have significant administrative overhead as clients must be registered and keys set up before deployment. Also, servers must have sufficient memory and computation power to ensure good performance, and connectivity must be maintained between clients and servers. Furthermore, unless additional costly measures are taken, attacks against a server may result in denial of service, or in the loss of a large set of long-term keys, compromising all security services. These disadvantages and constraints make server-based solutions unsuitable for sensor networks.

This report presents key-management services that enable a sensor network to set up cryptographic keys in an autonomous fashion, without relying on expensive cryptography or trusted servers, and with minimal administrative overhead.

The approach requires the sensors to share a small set of secret keys. These keys are loaded in each sensor before deployment, and, unlike other key predistribution schemes [4, 6], the number of keys required does not increase with the network size. The shared keys enable a pair of neighbors A and B — that is, two sensors that can communicate directly with each other — to mutually authenticate and securely exchange a key K_{ab}, unique to the pair (A, B). This key K_{ab} can then be used to secure local communication between A and B. We call the process of establishing these pairwise keys *bootstrapping* and call the corresponding links *secure local links*.

Since sensors are typically not tamperproof, we cannot assume that the initial keys used for bootstrapping can be kept secret forever. However, we can assume that it takes time for an adversary to physically compromise sensors and get the keys. Thus, sensors that are deployed at the same time can trust each other for a small time interval right after sensor deployment. Bootstrapping exploits this interval of trust to establish secure local links inexpensively. In particular, a sensor can authenticate and set up pairwise keys with its neighbors by using secrets that only recently deployed sensors possess. An extension of

the basic bootstrapping protocol supports multiphase deployment, in which secure links are established between sensors that are deployed in different phases.

Because of its low cost, this approach is well suited for key management in networks of resource-constrained sensors. The main benefits of the approach can be summarized as follows.

- Low memory and computation cost: Each sensor needs to store only a small set of (symmetric) keys, independent of network size, and no expensive operations such as those used in public-key cryptography are required.
- Low key setup overhead: Sensors deployed at the same time are preconfigured with the same set of keys. As a result, our approach has a small administrative key setup overhead.
- Self organizing: Sensors autonomously establish secure links without involving a trusted server that may become a bottleneck or a single point of failure.

The remainder of the report describes our key-management services in greater detail. Section 3 presents the basic bootstrapping protocol. An extended protocol for multiphase deployment of sensors is discussed in Section 4. An example network-level key-refresh service that builds upon the secure local links is described in Section 5. Our implementation of these protocols in the TinyOS framework is presented in Section 6, and related work is discussed in Section 7.

2 Notation and Cryptographic Primitives

The following table summarizes the notation used in this report.

$A, B, C, \ldots$	Node identities
N_a, N_b	Random numbers (nonces) generated by A or B
R_a	Random number stored in node A before deployment
$G_k(m)$	Keyed one-way hash function applied to string m using key k
$MAC_k(m)$	Message authentication code for message m, generated using key k
bk_1	Group authentication key used for bootstrapping
bk_2	Key generation key used for bootstrapping
gk_i	Key shared by all sensors of generation i and used for authentication with previous generations
K_{ab}	Pairwise key established by neighbors A and B

G is a keyed one-way hash function. It has the property that, given a random quantity r and a data string m, it is computationally infeasible to find the key k such that $r = G_k(m)$. Moreover, given m and k, one can compute $G_k(m)$ efficiently, but one cannot learn anything about $G_k(m)$ without knowing k. More formally, G is assumed to form a pseudo-random function family. That is, a polynomial time adversary cannot distinguish between the function G_k for a randomly chosen key k, and a true random function f of same domain and range as G_k. The notion of undistinguishability is defined rigorously in [1], for example.

MAC is an algorithm for constructing secure message-authentication codes using k. Given k and a message m, $MAC_k(m)$ can be efficiently computed, but one cannot efficiently construct $MAC_k(m)$ given m but not k. We also assume that the MAC is collision resistant. Knowing m and $MAC_k(m)$, it is computationally intractable to construct a message m' such that $MAC_k(m') = MAC_k(m)$. Like G, such a MAC can be constructed from a pseudo-random function family.

3 Bootstrapping Service

Authentication and key management require initial trust between some of the parties involved. For example, a public-key certificate is accepted as valid if signed by an authority

one trusts. If only symmetric-key cryptography is used, the parties that trust each other must somehow acquire a common shared secret that will enable them to communicate securely. In traditional networks, the initial secrets that are necessary to bootstrap the authentication services are typically set up by hand. For example, if a central authentication server is used, an initial shared key is distributed by an administrator when the client is registered with the server. This initial key is typically communicated offline to ensure secrecy.

In the case of large networks of embedded devices, manually setting a large number of keys is not practical. In many scenarios, access to the devices for administration is impossible once the devices are deployed. For example, sensors could be dropped from a plane over an inaccessible region or deployed in a toxic environment [7]. In such cases, device configuration is possible only before deployment, and there are no secure offline channels. Once deployed, the network must be autonomous and self-organizing. The initial keys should then be set up securely by the devices themselves, without manual intervention.

The typical scenario is for a set S of wireless sensors to be deployed or dropped in the environment. At this point, the devices must discover their neighbors and self-organize in an ad hoc network. During this initial phase, the main security concerns are external attacks and possibly malicious devices already present in the environment. The sensors from S themselves may be assumed initially trustworthy, as it takes time for an adversary to compromise them. As the risk of device compromise increases with time, it is crucial to very quickly establish the initial secure links. This calls for an efficient localized algorithm with minimal communication overhead. Our bootstrapping protocol is a localized algorithm that builds initial trusted links between sensors that are within direct communication range of each other. It is executed in a short time window after the sensors have been deployed.

3.1 Protocol Description

Since all the sensors of S are assumed initially trustworthy, two neighbors A and B can trust each other and establish a secure link if they can make sure that both of them belong to S. Hence, a fairly weak form of authentication is sufficient, namely, the ability for a sensor to prove that it belongs to S. This is implemented cheaply by loading a secret *group authentication key* bk_1 into all the members of S. Another secret key, the *key generation key* denoted by bk_2, is also stored in all sensors of S. It is used by neighbors A and B to generate a pairwise key K_{ab} after they have authenticated. Loading these two keys in all devices can be done easily when the sensors are programmed, and has very minimal administration overhead.

The protocol is straightforward. A sensor, say A, initiates the protocol by generating a random nonce N_a and broadcasting a *hello* message of the following form:

$$\langle \mathsf{Hello}, A, N_a, MAC_{bk_1}(\mathsf{Hello}, A, N_a) \rangle.$$

The message contains A's identity and the nonce N_a, and a message authentication code

(MAC) generated using bk_1. On reception of such a message, any sensor of S can check whether the MAC is valid, thus establishing that the sender possesses the secret key bk_1. Let B be such a sensor. Once B has verified the MAC, it generates a random nonce N_b and sends the following reply to A:

$$\langle \mathsf{Ack}, A, B, N_b, MAC_{bk_1}(\mathsf{Ack}, A, B, N_b, N_a) \rangle.$$

This acknowledgment communicates to A the nonce N_b, and proves to A that B knows bk_1 and has received N_a. When A receives the message, it can check whether the MAC is valid, and if so, extract the nonce N_b.

After this exchange, A and B have proven to each other that they know the group authentication key, and they are also both in possession of the nonces N_a and N_b. They construct a pairwise symmetric key as follows:

$$K_{ab} \quad = \quad G_{bk_2}(N_a, N_b),$$

where G is a keyed one-way hash function. This pairwise key enables them to communicate securely in the future. The key K_{ab} is actually split into two subkeys, K_{ab}^1 and K_{ab}^2, used for encryption and authentication of future messages, respectively.

3.2 Security

This bootstrapping protocol is a variant of the implicit key exchange protocol AKEP2 of [2]. It can be proven to be secure against an adversary who does not know the keys bk_1 and bk_2 using the models and techniques introduced by Bellare and Rogaway in [2]. The proof relies on the assumption that MAC and G are pseudorandom function families. Under this assumption, one can show that the following properties are satisfied for any adversary E, initiator sensor A, and responder B.

- The probability that B accepts a hello message that appears to be from A but was not sent by A is negligible.
- The probability that A accepts an acknowledgment message that appears to be from B but was not sent by B is negligible.
- E cannot distinguish between the key $K_{a,b}$ and a random bit string of the same length.

These properties can be stated precisely and proven rigorously as shown in [2]. This proof shows that the protocol is secure against an adversary E who can listen to traffic and inject messages, as long as E does not know the bootstrapping keys bk_1 and bk_2.

Since sensors are typically not tamperproof, an adversary could potentially obtain the keys by physically compromising a sensor. Clearly, if an adversary obtains bk_2 and bk_1

during the bootstrapping time window, then it can compute the pairwise keys K_{ab} from the messages it intercepts, or interfere with bootstrapping by forging hello and acknowledgment messages. This risk is small if the bootstrapping window is kept short. However, an additional risk exists if the adversary can record the messages exchanged between A and B during bootstrapping and later discover the key bk_2. Since all sensors use the same key-generation key, compromise of bk_2 can lead to the compromise of a large number of pairwise keys. Our countermeasure to this attack is to erase both bk_1 and bk_2 as soon as possible, after the bootstrapping window has elapsed.

3.3 Robustness and Cost

The unreliability of the communication link is a major issue in designing protocols for wireless sensor networks. We use several mechanisms to make the bootstrapping protocol robust to message loss.

First, all the sensors that are deployed together will play both the initiator and responder role. All of them will initiate the bootstrapping protocol at least once by broadcasting a hello message. Two neighbors A and B have then at least two chances to establish a secure link: once with B and once with A as the initiator. Optionally, the sensors can be programmed to send more than one hello message, thus executing the bootstrapping protocol more than once. This increases the probability that bootstrapping succeeds between two neighbors even if some messages are lost.

In addition, several timing mechanisms are employed to reduce the probability of message collisions. Randomization is used to prevent all sensors from sending a hello at the same time. When first started, a sensor will wait for a random period of time after deployment before sending its hello message. A similar technique is used to reduce the risk of collisions between several acknowledgments to a hello. When a sensor A broadcasts a hello message, neighbors of A that already share a pairwise key with A do not respond. Such sensors either already responded to a previous hello from A, or they have sent a hello to which A responded. Except for these sensors, every neighbor of A that received the hello is expected to respond. To reduce the probability of collisions between acknowledgments from different responders, replies to A are sent after a randomized wait time.

A final mechanism reduces the risk of collisions between hello messages and acknowledgments. When a hello message is transmitted at time t, then a time interval $[t, t + \Delta]$ is reserved for acknowledgments to this hello. Transmissions of hellos are triggered by a timer. If a sensor B receives a hello at time t, it will not broadcast its own hello until after $t+\Delta$. If B's timer expires in the interval, then B will not send its hello but restart the timer, with a randomized delay, to retry later.

All these mechanisms are necessary to make the protocol robust in a network where radio links are unreliable. Since the protocol requires message exchanges only between neighbors, it is inexpensive in terms of communication. For a sensor A, the cost is one

broadcast message per hello, and at most one reply from A to each of its neighbors. A more economical approach could be envisaged that requires only one hello message per node. A protocol that relies on this approach to exchange session keys is discussed in [12]. In such protocols, the key K_{ab} must be constructed from nonces attached to the hellos from A and B. This is very cheap in terms of communication, but also very unreliable if hellos are lost because of collisions or radio noise.

A main benefit of our bootstrapping protocol over other approaches [4, 6] is its low memory requirement. Only two secret keys are necessary for bootstrapping, irrespective of the network size. The computational cost is also relatively small as all the cryptographic primitives required can be implemented using block ciphers.

4 Multiphase Deployment

Sensors may be deployed in different phases. For example, new sensors may be added when previously deployed sensors fail or when the capability of the existing network is determined to be insufficient. We assume that sensors are deployed in successive generations. The bootstrapping protocol of Section 3 applies to sensors of a single generation. This section presents an extension of bootstrapping that enables a sensor A of generation i to establish a secure link with a sensor B of a later generation $j > i$.

The basic idea is for A to store a random quantity, r_a, and a secret $S_{a,j}$ derived from R_a. The secret has the property that no other sensor of generation i, or earlier generation, can efficiently compute $S_{a,j}$ from R_a. On the other hand, a sensor of generation j can efficiently compute $S_{a,j}$ from R_a. The secret is used to establish a secure link between A and sensors of generation j. The construction of $S_{a,j}$ relies on a keyed one-way hash function such as the function G used previously. For authentication across multiple generations, we add an extra key gk_j that is shared by all sensors of generation j, and the secret $S_{a,j}$ is constructed by

$$S_{a,j} \quad = \quad G_{gk_j}(R_a).$$

Thus, under the assumption that G is a secure one-way function, only sensors of generation j can construct $S_{a,j}$ from R_a. Sensor A itself knows $S_{a,j}$ and R_a, but it does not possess gk_j. Several secrets such as $S_{a,j}$ must be stored in A before deployment; each corresponds to one generation between $i+1$ and $i+n$, where $n > 0$ is the number of future generations with which A can establish secure links.

Sensor A of generation i and B of generation j use the following protocol, called cross-generation bootstrapping (XGB). When B is first deployed, it adverstises the event by broadcasting a *hello* message:

$$\langle \mathsf{Hello}, B, j, N_b \rangle$$

The *hello* message consists of B's identity and generation, and a randomly generated nonce N_B. Upon receiving the message, A extracts the generation number j and extracts corresponding secret $S_{a,j}$. Then A sends the following acknowledgment to B:

$$\langle \mathsf{Ack}, A, B, R_a, MAC_{S_{a,j}}(\mathsf{Ack}, A, B, R_a, N_b)\rangle$$

When B receives this message, it can compute $S_{a,j}$ using gk_j and R_a. Then B will verify whether the MAC is valid to establish that x possesses the secret $S_{a,j}$. If the MAC is determined to be valid, B completes the protocol by sending a second acknowledgment that B can authenticate using the secret:

$$\langle \mathsf{Ack2}, B, A, MAC_{S_{a,j}}(\mathsf{Ack2}, B, A)\rangle.$$

After XGB, A and B will derive a new session key based on $S_{a,j}$, R_A, and N_b for securing their communication in a way similar to that of the bootstrapping protocol.

Because of the one-way property of function G, A cannot obtain gk_j. Thus, A may not tamper with the communication between a sensor of generation j and another sensor other than itself. Also, A cannot masquerade as another sensor Z of generation i, of an earlier generation, or of a later generation when communicating with a sensor of generation j because A cannot efficiently compute $G_{gk_j}(R_z)$.

As previously, the security of the XBG protocol relies critically on the assumption that sensors of generation j are trustworthy when deployed and remain trustworthy for a long enough time to complete the protocol. It is also crucial for all sensors of generation j to erase the key gk_j as soon as the cross-generation protocol is over.

Using the secure local links established by the XGB protocol, one can securely transmit a group key, K_g, from generation i and pre-generation i sensors to generation $(i+1)$ sensors. In other words, we have a set of old sensors of generations smaller than $i+1$ that share a secret group key K_g. This set may be strictly smaller than the set of all generation i and pre-generation i sensors. For example, some sensors may be excluded because they are detected to be compromised or misbehaving. When generation $i+1$ is deployed, we want them to obtain the group key K_g so that all sensors can participate in a common application.

Again, we assume that sensors are not compromised shortly after they are deployed. After generation $(i+1)$ sensors are deployed, there exists a time window during which all generation $(i+1)$ sensors can be trusted to behave correctly and no adversary can obtain the secrets stored in these sensors. During this time window, old sensors can establish secure local links with new sensors of generation $i+1$ using XBG, and they can transmit K_g to them using the secure local links. To prevent a misbehaving old sensor from causing generation $(i+1)$ sensors to use an incorrect group key, generation $(i+1)$ sensors can exchange the group keys they receive among themselves to filter out incorrect group key(s), assuming the majority of the group keys obtained from distinct (based on the R_a values)

pre-generation-$(i+1)$ sensors are correct. Moreover, thanks to its inability to obtain $S_{z,i+1}$ for another sensor Z, a misbehaving pre-generation-$(i+1)$ sensor cannot masquerade as Z in this process. Thus the misbehaving sensor cannot perform a Sybil attack [5] to outnumber the correct sensors by presenting itself as multiple pre-generation-$(i+1)$ sensors.

5 Using Secure Local Links

Once neighbors can communicate via secure local links, other security services can be built inexpensively. As a simple example, chaining can be used to secure communication between distant nodes. We present a group-key distribution protocol built on top of the secure local links.

An inexpensive way of adding security to a sensor network is to rely on a common group key known by all the sensors. For example, this approach is supported by TinySec [11], a link-layer encryption service for TinyOS. Using a global key, messages between sensors can be encrypted for confidentiality, or protected against corruption by using a MAC. An important advantage of this approach is that secure multicast is very efficient. The sender of a multicast message encrypts the message and computes the MAC once using the group key. Every recipient decrypts the message and checks the MAC only once.

A limitation of using a shared group key is that compromise of a single sensor is sufficient to obtain the key, which gives an adversary access to all network traffic. To recover from such an attack, one needs the means to distribute a new group key to all group members except those that are considered compromised. This can be easily implemented by exploiting the secure local links.

Our key refresh protocol provides this service. It can be initiated by any member of a group, although it is typically done by a base station. The initiator generates a new, random group key and optionally constructs a list of sensors to be excluded from the group. The new key together with the exclusion list, a sequence number, and the initiator's identity is distributed via the secure local links to all sensors, except those on the exclusion list. First the initiator securely sends a copy of the key and exclusion list to its good neighbors (i.e., those not on the list), using the pairwise key it shares with each of these neighbors.

A key-refresh message sent by A to B is of the following form:

$$\langle \mathsf{KeyRefresh}, B, A, O, N, \{K_g\}_{K^1_{ab}}, L, MAC_{K^2_{ab}}(\ldots)\rangle.$$

In this message, O is the originator of the new key, that is, the sensor that initiated the key refresh, N is the group key's sequence number, K_g the new group key, and L the exclusion list. The message is protected by using the pairwise key K_{ab} that A and B set up during bootstrapping. More precisely, the subkey K^1_{ab} is used to guarantee confidentiality of K_g, while K^2_{ab} is used for authentication and integrity.

When B receives such a key-refresh message, it checks the message integrity using K^2_{ab}, and it checks whether the message is fresh, based on the sequence number N and the originator identity O. If both checks succeed, B accepts the new group key carried by the message, and forwards it to all its neighbors except A and any sensor on the exclusion list. This requires a re-encryption and MAC computation for each of B's good neighbors.

This protocol distributes the new group key securely and robustly. As long as the good group members are connected, the flooding-like procedure distributes the new key to all good members in a robust manner. However, this procedure is expensive in terms of communication and computation. The key-refresh message is decrypted once but encrypted multiple times by each sensor, and sent in separate messages to each neighbor. This may not be a significant issue if the group key is not changed very often, but more efficient solutions may be desirable.

Including the identity of the originator and a sequence number provides the means to arbitrate between conflicting key-refresh messages, which can occur if multiple nodes initiate the protocol at roughly the same time. Key-refresh messages are totally ordered using the lexicographic order on the pair (N, O). When a key-refresh message is received by B, it is accepted and forwarded only if it is higher in the lexicographic order than all key-refresh messages seen by B in the past.

This protocol is secure as long as the originator and all nonexcluded sensors are not compromised. If one of the relaying nodes or the originator is compromised it could exploit the protocol to effect denial of service. A possible protection against such compromises is to require that the originator of all key refresh messages come from a trusted node, such as a base station. This could be done using a protocol such as μ-Tesla [14]. We are currently investigating extensions of our protocols for authenticating distant nodes that could also be used in this context. We are also examining monitoring mechanisms to detect misbehaving nodes in a timely manner.

6 Implementation

We have implemented and experimented with the bootstrapping and key-refresh protocols using Mica devices [9]. The Mica platform is based on an Atmel ATmega 103L or Atmega 128 microcontroller and the RF Monolithics TR100 radio transceiver. The microcontroller is an 8-bit processor that runs at 4 MHz, and includes 4 kB of RAM and 128 kB of flash program memory. Mica supports a variety of sensor boards with photo-diode, thermistor, microphone and sounder, and magnetic and acceleration sensors. The radio has a fixed frequency of 916.5 MHz and a range that can be varied from inches to hundreds of feets, depending on power.

The Mica platform runs UC Berkeley's TinyOS operating system [8]. TinyOS is a modular operating system designed for small sensor platforms. In the TinyOS model, an

application consists of a set of software components that interact using event passing and a simple tasking mechanism. The TinyOS infrastructure provides a collection of low-level components for interaction with sensor hardware, which can be flexibly assembled and integrated with application components. Since version 1.0, TinyOS and application components can be written in NesC, an extension of the C programming language that supports the TinyOS component and composition model. All our implementation was done with TinyOS 1.0.

6.1 Radio Stack

Implementing our security protocols in TinyOS required significant extensions to the TinyOS radio stack. In version 1.0, TinyOS actually provided two different radio stacks for the Mica platform. One was the standard radio implementation that does not include any security. In this implementation, radio messages consist of a header, a payload, and a cyclic redundancy check (CRC) that is used to detect message corruption. The header includes fields such as destination address, message type, and length. This version of the radio stack was not suitable for our protocols because the message formats they require do not match TinyOS messages very well. For example, all our protocols use cryptographic MAC for authentication and integrity, which means that a CRC is unnecessary. Also, some of the header fields required by TinyOS are not used by our protocols.

The second radio stack available with TinyOS is TinySec [10]. It provides link-layer security based on a fixed network-wide key. In TinySec, the CRC is replaced by a MAC and the payload is encrypted. This use of cryptography for securing radio communication could address some of our needs but it is not sufficiently flexible for our protocols. TinySec relies on a fixed key that is used for all messages and provides no interface for changing the key. In our protocols, several keys are maintained for each neighbor of a sensor. Some messages require different keys depending on the destination. Conversely, checking a received message requires identifying the sender to find the correct pairwise keys to use. Furthermore, some MAC computations that our protocols use require information that is not included in the messages sent (e.g., the acknowledgments to a *hello* message during bootstrapping). For these reasons, we need a radio stack that provides flexible per-message formatting and encryption.

We have developed a new radio stack for TinyOS that provides these services. This stack is an extension and combination of the standard TinyOS stack and TinySec. It provides four communication services that use the following four types of messages:

- Plain messages in a format similar to that used by the standard TinyOS stack. Messages are sent in clear and a CRC is added for error detection.
- Encrypted messages, similar to the format used by TinySec. The message payload is encrypted, and a MAC is added for integrity and authentication.

Stack	bytes in ROM	bytes in RAM
TinyOS	9440	356
TinySec	14630	1078
Our Stack	11818	914

Table 1: Code Size with Different Stacks

- Authenticated messages: a variant of the TinySec format in which the payload is sent in clear and a MAC is added.
- Raw messages: intended to be formatted by the application. A raw message consists of a single header byte that specifies the message length and a payload.

Thus, two of the communication services provided by our radio stack are the same as what the TinyOS stack and TinySec provide. Authenticated messages are a simple variant of TinySec messages. The raw-message interface gives the application full responsibility for formatting and error checking. All four types of communication services are available within the same radio stack, and can be accessed via different interfaces. By default, the encrypted and authenticated message services use group keys that are fixed at compilation time, but our radio stack provides an interface for changing these keys at runtime.[1]

An application that sends a message via the raw-message interface is free to format the payload in any way. Conversely, when a raw-message is received, the radio stack forwards it to the application without performing any check. This interface gives the most flexibility and it is the one we use for the bootstrapping and key refresh protocols.

Our radio stack reuses many components of TinyOS and TinySec, and attempts to remain compatible with them. For example, we use the same MAC algorithm as TinySec, and we encrypt the payload in CBC mode using the cipher stealing technique also employed by TinySec. The block ciphers we use for computing MACs and for encryption are also inherited from TinySec.

The size and performance of our radio stack are similar to the TinySec stack. Table 1 shows the code size and RAM usage of the same example application compiled with the TinyOS stack, the TinySec stack, and our new stack. The data was obtained with the TinyOS 1.0 distribution. In this example, both TinySec and our stack used the SkipJack block cipher. The application is one of the demo applications distributed with TinyOS; it periodically increments a counter and sends its value on the radio. As could be expected, using cryptography increases the code size and RAM usage of both our stack and TinySec, compared with the nonsecure TinyOS stack, but the code size fits easily within the Mica

[1]This implementation was done using version 1.0 of TinyOS. A more recent version of TinySec [11] includes some of the same extensions as our radio stack.

program memory. On the other hand, the RAM consumption is close to 25% of the total Mica RAM, which may be a lot for certain applications. Several optimizations are possible to reduce the memory used by the block cipher. For example, the SkipJack implementation stores a constant table of 256 bytes in RAM. It is possible to move this table into ROM, at the cost of a slight reduction in performance. With the table stored in ROM, SkipJack is about 7% slower than with the table in RAM.

6.2 Protocol Implementation

Our bootstrapping protocol is intended for authentication and key distribution between neighbor sensors in a network. We have implemented this protocol on the Mica platform using the radio stack described previously. The *hellos* and acknowledgment messages are sent and received via the raw-message interfaces since they require special formatting and MAC construction.

The bootstrapping protocol is implemented by a NesC component called `SecureLinkManager`. The main role of the `SecureLinkManager` module is to build a table of authenticated neighbors. At the end of bootstrapping, the table contains the identity of each authenticated neighbor and the two pairwise keys (i.e., K_{ab}^1 and K_{ab}^2) established with this neighbor, and other bookkeeping data.

The bootstrapping protocol uses a different block cipher than those available with the TinySec distribution, namely, AES. The main reason for developing a new cipher implementation was to reduce the memory space needed to store the pairwise keys. TinySec provides implementation of two block ciphers — RC5 and SkipJack — but these implementations are optimized for speed. They use buffers to store intermediate data derived from the cryptographic keys to speed up encryption and decryption. Storing this data requires 128 bytes of memory for SkipJack and 104 bytes for RC5. This is too much if one needs to store cryptographic material equivalent to two keys per neighbor. We have developed an AES implementation that requires less RAM. This implementation uses 128-bit keys, has a block size of 128 bits, and is optimized for space. Using this implementation, the neighbor table requires only 48 bytes per neighbor for storing cryptographic material.

All the cryptographic operations performed by the `SecureLinkManager` module rely on this AES implementation. This includes MAC computation and generation of the pairwise keys as discussed in Section 3. In addition, we use the AES cipher for implementing a secure pseudo-random generator for generating nonces. This generator is initialized with a random AES key, that must be different for each sensor, and that is constructed when the Mica nodes are programmed.

We have also developed a prototype implementation of the key refresh protocol of Section 5. This protocol is used to change the group keys used by the TinySec-like services of our radio stack. The implementation of this key-refresh protocol relies on the neighbor table constructed by bootstrapping to flood key-refresh messages. These key-refresh messages

are formatted at the application level and are transmitted via the raw-message interface of the stack. The encryption and MAC applied to these messages use the pairwise key stored in the neighbor table and thus employ our implementation of AES.

The whole code for bootstrapping and key refresh together with the radio stack occupies around 17,000 bytes of program memory. The total RAM usage depends on the size of the neighbor table. Assuming a table of as many as 10 nodes, an application requires 1753 bytes, which includes the neighbor table and the data structures and buffers used by the radio stack.

6.3 An Example Application

We have tested our bootstrapping and key refresh implementations in a demonstration application: a perimeter monitoring scenario in which sensors along a perimeter communicate sensor readings (in our case, light levels) via an ad hoc network of other nodes. The routing layer is an implementation of destination-sequenced distance-vector (DSDV) routing [13] written for TinyOS by Intel Research's heterogeneous sensor networks project [15].

During normal operation, sensor readings are sent along dynamically updated multihop paths to a base station. However, the routing protocol is vulnerable to malicious route update messages. For instance, a compromised "black hole" node can falsely advertise that it is close to the base station, and then not forward sensor readings. Even in the case where messages are signed with a group key (as in TinySec), all sensor measurements can be thwarted by a single malicious node that knows the group key. However, with the fallback of pairwise keys obtained via bootstrapping, we can — if we know the identity of the malicious node — refresh the group key to trusted nodes only. After a straightforward assembly of the TinyOS components for bootstrapping, key refresh, and routing, our implementation successfully demonstrated this capability. A screenshot of this application is shown in Figure 1.

7 Related Work

Because of resource constraints, most of the key-management and distribution protocols developed for standard networks are not applicable to large-scale sensor networks. Readers are referred to [3] for a more detailed discussion of how the resource constraints impact security. We review recent work on key management for sensor networks.

Eschenauer and Gligor [6] and Chan et al. [4] have proposed key-management schemes based on random key predistribution. A subset of keys is randomly selected from a large key pool and distributed to each sensor before deployment. Secure communication channels can be established by using common keys shared by neighbor nodes. Through random graph analysis and simulations, the authors show that random key predistribution can ensure

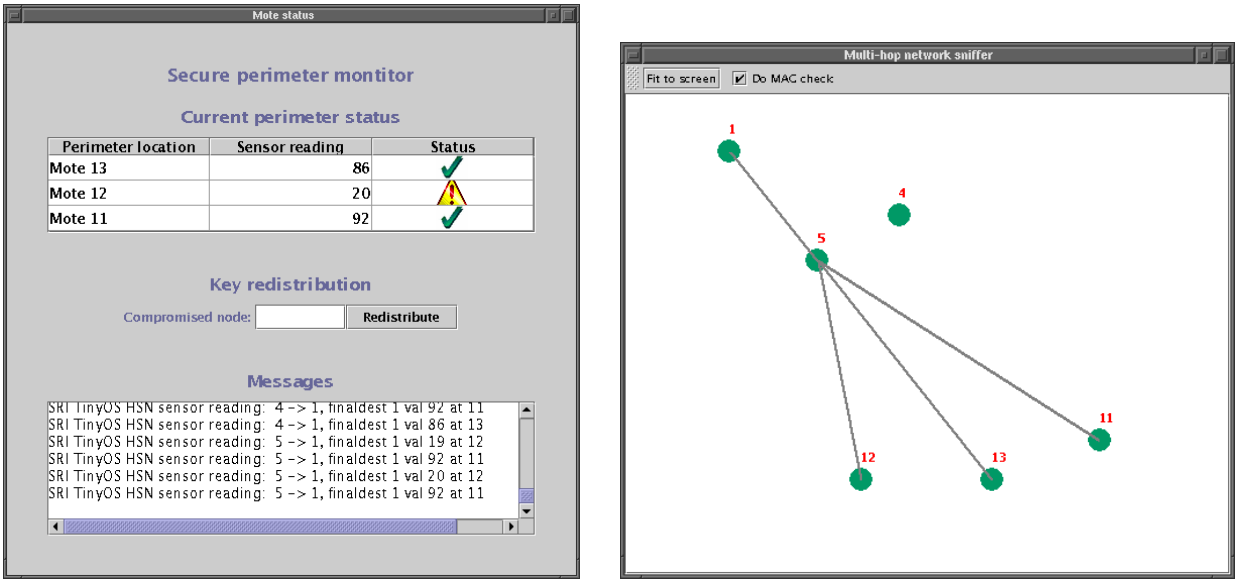

Figure 1: Perimeter monitoring demonstration.

with high probability that the network is connected via secure links. For example, given a network with 10,000 sensors, where each sensor can directly communicate with 40 sensors and stores a random key set of size 250 obtained from a key pool of size 100,000; then, the network is almost certainly connected. However, it is not clear whether such schemes can be used for sensors with very limited memory such as the Mica platform. As network size increases, one must either increase the number of keys given to each sensor or decrease the size of the key pool to ensure with high probability that the whole network is connected. Assigning too many keys to each node is impractical for sensors with limited memory, and reducing the size of the key pool has an impact on security. With a small key pool, access to a few sensors may be sufficient to compromise a large number of communication links. On the other hand, a random predistribution scheme can be combined with our bootstrapping protocol. Instead of assuming that all nodes share common bootstrapping keys bk_1 and bk_2, one could predistribute a small number of keys randomly chosen from a key pool. This would make the protocol partially resilient to compromise of a node during the bootstrapping window.

Other approaches such as SPINS [14] rely on a trusted base station for distributing keys between sensors. A major part of the SPINS protocols is a very efficient approach for authenticating multicast messages that originate from the base station. SPINS also introduced a link-layer encryption and authentication algorithm called SNEP. This algorithm adds very little overhead over unencrypted messages but it requires both ends of a communication

link to maintain consistent counters. This may be difficult to ensure if the radio link is unreliable. TinySec [11] is an alternative security service, developed in the TinyOS framework, to add security in sensor networks. TinySec assumes that all sensors share common cryptographic materials, and as discussed in this report can be enhanced using our bootstrapping and key-refresh protocols. Our implementation borrows many of its components from TinySec.

8 Conclusion

We have presented a collection of lightweight protocols for authentication and key distribution in resource-constrained sensor networks. These protocols have been implemented on a representative sensor platform. They require only inexpensive cryptographic primitives and use little memory. Security is achieved by taking advantage of bounded periods of trust, just after sensors have been deployed, to quickly and cheaply establish pairwise keys. Bootstrapping keys that enable sensors to authenticate during this trust period are used only within that time, and erased after pairwise keys have been exchanged.

In future work, we are planning to extend these protocols to support authentication and key exchange between distant nodes. The challenge is to develop protocols for this purpose that are as economical as possible, while ensuring security even if some of the nodes in a network have been compromised.

Bibliography

[1] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceeedings of the 38th Symposium on Foundations of Computer Science*. IEEE, 1997. 3

[2] Mihir Bellare and Phillip Rogaway. Entity Authentication and Key Distribution. In *Advances in Cryptology – Crypto'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer-Verlag, 1994. 5

[3] David W. Carman, Peter S. Kruss, and Brian J. Matt. Constraints and approaches for distributed sensor networks. Technical Report 00-010, NAI Labs, September 2000. 14

[4] Haowen Chan, Adrian Perrig, and Dawn Song. Random key predistribution schemes for sensor networks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 197–213, May 2003. 1, 7, 14

[5] John R. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002. 9

[6] Laurent Eschenauer and Virgil Gligor. A key-management scheme for distributed sensor networks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 41–47, November 2002. 1, 7, 14

[7] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 263–270, Seattle, WA, August 1999. 4

[8] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the* 9^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 93–104, Cambridge, Massachusetts, November 2000. 10

[9] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002. 10

[10] Chris Karlof, Naveen Sastry, and David Wagner. TinySec 0.91: User manual. Manuscript, February 11, 2003. 11

[11] Chris Karlof, Naveen Sastry, and David Wagner. TinySec: User manual. Manuscript, September 12, 2003. 9, 12, 16

[12] Bocheng Lai, Sungha Kim, and Ingrid Verbauwhede. Scalable session key construction protocol for wireless sensor networks. In *IEEE Workshop on Large Scale Real-Time and Embedded Systems (LARTES)*, Austin, Texas, December 2002. 7

[13] Charles Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994. 14

[14] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J.D. Tygar. SPINS: Security protocols for sensor networks. In *Proceedings of the* 7^{th} *Annual International Conference on Mobile Computing and Networks (MOBICOM 2001)*, pages 189–199, Rome, Italy, July 2001. 10, 15

[15] Intel Research. Heterogenous sensor networks. `http://www.intel.com/research/exploratory/heterogeneous.htm`. 14