Key Refreshing in Wireless Sensor Networks

January 14, 2008

Abstract

The problem of establishing symmetric keys in wireless sensor networks has been extensively studied, but other aspects of key management have received comparatively little attention. In this paper we consider the problem of refreshing keys that are shared among several nodes in a WSN, in order to provide forward security. We discuss several applications that lead to sensor networks with very different properties, and we propose key refreshing schemes that are useful in each of these environments, together with resynchronisation methods that allow nodes possessing different versions of a key to arrive at a common version.

1 Introduction

A wireless sensor network (WSN) consists of a number of small, battery-powered sensing devices (known as sensor nodes) that employ wireless communication to form a network in order to distribute and manipulate the sensed data. As public-key cryptographic techniques are regarded as being undesirably costly for these highly constrained devices it is necessary for nodes to share symmetric keys for the purposes of providing authentication, data integrity or confidentiality. Much research has been done on the problem of establishing shared keys in such networks (see [3, 8, 14] for surveys of this area); less attention has been paid to the ongoing key management requirements that arise after a network has been deployed. One such requirement, recognised in the cryptographic community since the 1980s (see [5]), is *forward security*: if a node is captured and its secret material compromised, an adversary should not be able to decrypt messages that were intercepted by the adversary in the past. In a network environment, we may weaken this requirement to insisting that the adversary cannot decrypt messages that were broadcast more than a very short time ago. Forward security is of particular significance in a WSN, as the nodes operate in an uncontrolled environment and lack tamper proof hardware, and hence are vulnerable to adversarial compromise. Moreover, the difficulty of distinguishing node compromise from routine node failure adds to the security challenges of such an environment.

Schemes for *refreshing* keys (updating keys using a one-way function) in order to provide forward security in a WSN setting have so far been restricted to networks in which there are no group keys and where nodes are capable of storing a separate key for each of their neighbours (this is the case for the schemes proposed by Klonowski *et al.* [6] and Mauw *et al.* [9]). But these restrictions are often not valid: group keys (which could be shared by many nodes) are needed in many applications; even when keys are only used to secure pairwise links, a node might use the same key to communicate with more than one of its neighbours because of limited storage capabilities. (For example, this will often be the case if key predistribution techniques such as those of Eschenauer and Gligor [4]

or Lee and Stinson [7] are used.) Indeed, when a network is dense, the number of secure links a node might want to establish might exceed the number of keys it is able to store. In this paper we consider how to provide forward security by key refreshing in networks with these more general patterns of key sharing.

In Section 2 we consider definitions of forward security appearing in the literature, and examine standard techniques for refreshing pairwise keys, as well as those that have been proposed for a sensor network context.

In Section 3 we discuss several applications for sensor networks that give rise to five distinct categories of networks with differing properties. As suggested by the examples given in [13] by Römer and Mattern, the properties of sensors and their communication patterns can vary much more widely than is acknowledged in much of the sensor network literature. The network environments we describe encompass a wide range of possible WSNs: sensors may be fixed or mobile, the network may be dense or sparse, the amount of communication within the network may be steady, or it may fluctuate. In fact, the schemes we propose are not restricted to sensor networks, but may find application in any network in which symmetric keys are shared by more than two entities. We assume that each node stores a number of symmetric keys from some key pool, and that each key is potentially stored by a number of different nodes. We further suppose that a node is able to determine its *neighbours:* nodes with which it shares at least one key, and which are within its communication range.

In Section 4 we propose schemes for updating keys to provide forward security in the first two environments we have identified.

In Section 5 we discuss a scheme appropriate for the remaining three environments, and provide schemes for resynchronising the versions of keys possessed by nodes in cases where the nodes hold differing versions of the same key. We conclude with a discussion of some further issues relating to key refreshing in WSNs.

2 Forward Security through Pairwise Key Refreshing

In this section we describe standard techniques for achieving forward security for a single pairwise key, as well as two schemes that have been proposed in a sensor network context. We then point out the problems of extending pairwise schemes to the setting where a key is shared by more than one node in a network.

2.1 Notions of Forward Security

Provably secure refreshing In [1], Bellare and Yee describe how symmetric keys can be refreshed using a *stateful generator:* a pseudorandom bit generator that takes a state as input, then produces an output block and a new state, which is used as the input for the next iteration of the generator. Such a generator is defined to be *forward secure* if an adversary who is given access to the state of the generator at a time of its choice cannot feasibly distinguish the sequence of bits previously output by the generator from a random sequence. A stateful generator can be used for key refreshing in the following manner. Let $g : \{0,1\}^s \to \{0,1\}^{b+s}$ be a pseudorandom generator (such as the Blum-Blum-Shub generator [2], for example) and let s_0 be a randomly chosen s-bit initial state. The first b bits of $g(s_0)$ are output as an initial key k_1, and the remaining s bits are stored as the state s_1. A sequence of keys k_i can then be produced by applying g to the state s_{i-1}, updating the

state using the output of g each time. Bellare and Yee prove that this stateful generator is forward secure, provided that g is pseudorandom. As this process is deterministic, two entities who share an initial common state can use this method to produce a forward-secure sequence of shared keys without any communication overheads.

In the constrained environment of a WSN, however, the use of a provably secure generator is likely to prove too computationally expensive. Also, the need to store the generator's internal state as well as the current key represents an additional overhead. In order to realise a significant gain in efficiency, it may therefore be deemed acceptable in a sensor network context to consider a weaker form of forward security, namely: given a version of a key, it should be computationally infeasible to decrypt any ciphertexts produced with prior versions of the key. This can be achieved by the use of a one-way function. This is a standard technique that can be described as follows.

Standard refreshing Suppose that nodes Alice and Bob share a symmetric key k taken from a key space $\mathcal{K}$. Let $f : \mathcal{K} \to \mathcal{K}$ be a public one-way function (so f can be efficiently computed, but it is difficult to find an inverse image under f). In practice, we may build a one-way function f from a secure hash function.

Define the ith *version* k_i of the key k by $k_0 = k$ and $k_i = f(k_{i-1})$ for $i \geq 1$. Initially, both Alice and Bob store version 0 of the key. Whenever they exchange an encrypted message, they use the current version of the key. After exchanging the message, they replace their key k_i by $k_{i+1} = f(k_i)$, and destroy the original key k_i. This process is known as *refreshing* the key. Note that if Alice or Bob are compromised, the adversary only comes into possession of the most up-to-date version k_i of the key. The adversary is unable to compute any previous version k_j of the key, where $j < i$, because f is one-way, and so cannot decrypt any ciphertexts intercepted before the node compromise. This method therefore satisfies the restricted notion of forward security (although it does not acheive Bellare and Yee's indistinguishability property, as the function f can be used to distinguish a sequence of keys from a random sequence). Both standard and provably secure refreshing require no communication overhead. The former technique is a little more efficient, but the latter technique has the advantage of a more precisely defined security model. In our schemes, which we describe in Sections 4 and 5, either technique can be used.

2.2 Forward Security in Sensor Networks

The literature contains examples of schemes for refreshing pairwise keys that have been proposed specifically for WSNs. In [9], Mauw, van Vessem and Bos consider a network in which each node communicates directly with a base station. Each node n shares a unique initial key x_n^0 with the base station, and the standard refreshing technique described in Subsection 2.1 is used, with key x_n^i being generated as $\mathcal{H}(x_n^{i-1})$, where $\mathcal{H}$ is a one-way hash function (the authors suggest the use of SHA-1).

Klonowski, Kutyłowski, Ren and Rybarczyk consider the scenario in which every node "shares a separate pairwise key with each neighbour" [6]. Their scheme also employs a one-way function F but, based on a key distribution mechanism in [11], incorporates an element of randomness. In their scheme, if nodes A and B share key k_{AB} and A wants to send a message to B, then A encrypts the message using a key $k' = F(k_{AB}, i)$, where i is chosen uniformly at random from the set $\{0, 1, \ldots, l\}$ for some small l. Node B then has to perform several trial decryptions in order to determine the precise value of i and hence k' that was used. This is more computationally expensive than the standard pairwise refreshing technique, and has the complication that B must succeed in receiving and decrypting the message in order for key refreshing to occur successfully. The presence of the

randomness does, however, provide the additional property that an adversary that possesses an old version of a key will eventually be unable to determine newer versions of the key unless it has continued to monitor all the messages sent using intermediate versions of that key. Note that real randomness, rather than pseudorandomness, must be used for this additional property to hold (as we may assume that node compromise reveals the state of any pseudorandom generator used by the node). This limits the applicability of the scheme. Note also that the computational burden of trial decryptions may be eliminated from this scheme at the expense of a little more communication complexity by appending the random bits used in key refresh to the message from A to B.

Since randomness is not needed for forward security, and a security model where the randomness has benefits must involve a weakening of the standard model considered for sensor networks (in which an adversary is capable of intercepting all communication), we do not consider randomness in the schemes we present later in the paper.

The schemes we have considered so far all involve refreshing keys that are shared by exactly two entities. As discussed in the introduction, however, many sensor network applications involve keys that are shared by more than two participants. Refreshing keys in this situation becomes more complicated; in the following subsection we discuss some of the issues that arise.

2.3 Problems that Arise when Widely Shared Keys are Refreshed

When seeking to maintain forward security when a key k is shared by more than two nodes, a pairwise key refreshing scheme cannot be used without some modifications. If user X is currently storing version i of the key k, we write $\mathrm{vn}_k(X) = i$. In the standard two node schemes discussed in Subsection 2.1, it is clear that $\mathrm{vn}_k(\text{Alice}) = \mathrm{vn}_k(\text{Bob})$ at all times, whereas this will not usually be the case if more than two nodes use the same key. If communicating nodes simply refresh their keys after each message, other nodes using the same key will not necessarily be aware that a message has been transmitted and so will not refresh their key appropriately. This causes two problems:

- (*Undecipherable messages*) If users X and Y are such that $\mathrm{vn}_k(X) < \mathrm{vn}_k(Y)$, we have a problem if X sends a message to Y using version $\mathrm{vn}_k(X)$ of k (since Y cannot decrypt). So we need to have a mechanism to ensure that the version numbers of X and Y are synchronised.

- (*Degradation of forward security*) Suppose some node Z has refreshed its key less than communicating nodes X and Y, so $\mathrm{vn}_k(Z) < \mathrm{vn}_k(X) = \mathrm{vn}_k(Y)$. Then the compromise of Z allows an adversary to decipher any messages exchanged by X and Y using versions of the key lying between $\mathrm{vn}_k(Z)$ and $\mathrm{vn}_k(X)$. So we need to have a mechanism to ensure that no node stores a "very old" version of a key.

The first problem could be solved by requiring nodes to use a different version number of the key for each pairwise communication link they maintain. A node would have to store a set of version numbers (one for each link) together with the version of the key corresponding to the lowest of these version numbers. But this causes a proliferation of version numbers and so this solution is often unrealistic because of storage constraints in the WSN model. Moreover, the second problem becomes worse.

In this paper we propose two alternative classes of solutions. In Section 4 we address the problems of undecipherable messages and degradation of forward security by describing mechanisms to ensure that all nodes update their copy of k at essentially the same time (*synchronised key refreshing*). However, in some applications this approach is unrealistic, and so in Section 5 we describe a

method whereby a pair of communicating nodes determines which version number of the key to use (*asynchronous key refreshing*, addressing the first problem) and then describe several mechanisms to ensure that no node stores a low version number of a key (*key resynchronisation*, addressing the second problem). First, however, we discuss several applications for sensor networks. These give rise to five categories of network environment, in which our different schemes are appropriate.

3 Sensor Network Environments and Applications

The vast array of applications that have been proposed for WSNs leads to networks with widely varying properties. In order to provide a context for the key refreshing schemes we propose in this paper, we consider five distinct sensor network application environments. The differing characteristics of these situations mean that the most appropriate method of key refreshing varies between examples. Here we describe these environments, and give examples of possible applications for which they are appropriate.

1. **(Synchronised clocks)** In many applications, the nodes in the network have synchronised clocks. As discussed in Römer *et al.* [12], clock synchronisation comes at a cost. However, in networks where it is provided for the purposes of the application, we can exploit clock synchronisation for performing key refreshing. Examples of applications for which clock synchronisation is necessary include an intruder detection system in which records of events are timestamped by individual sensors, or a system for monitoring volcanic activity in which the network is used to provide a global picture of a volcano's behaviour at a given time.

2. **(Frequent flooding)** Many environments do not require nodes to have synchronised clocks, but frequent flooding of messages through the network should take place. This might be the case, for example, in a disaster recovery scenario in which sensors attached to medical personnel flood real-time updates on their status to others in the area.

3. **(Infrequent network-wide events)** Some applications call for networks in which synchronised clocks and regular flooding are not present, but in which there is an occasional event that can be detected by the entire network. For example, the data sink could consist of a helicopter that flies over the network occasionally and broadcasts a request to retrieve data to the entire network. In some applications an infrequent flooding of the network might take place (for example, an intruder detection system in a warehouse might be armed or disarmed by a flooded message that is triggered by the locking or unlocking of a door).

4. **(Infrequent local events)** Our fourth category consists of networks in which no global events occur with sufficient frequency or regularity and no regular flooding takes place, but whose communication capacity can support an occasional flooded message. This is the case in networks measuring events that occur locally, and in which there is a low amount of (mostly local) communication between nodes.

5. **(Regular disconnection)** The final network environment that we address consists of networks that have a high likelihood of becoming disconnected, but in which the separate components continue functioning independently until the network is later reconnected. This might occur in sparse networks in which nodes are sited at the very edge of their communication capacities, or networks in which clusters of nodes are associated with moving objects, such as vehicles.

4 Schemes to Synchronise Key Refreshing

This section contains two schemes that can be used to synchronise key refreshing throughout a network; they can be applied in the first two application environments respectively. The schemes can either be used to refresh a fixed key from the keypool, or a subset of keys.

4.1 Synchronous Event-Driven Key Refreshing

The simplest means of maintaining sychronicity of key version numbers is:

Scheme 1. (Event-driven refreshing) *Nodes refresh their keys in response to some event that can be observed by the whole network.*

In our first application scenario, in which nodes have synchronised clocks, the network can simply refresh their keys every five minutes, say, thus providing forward security for messages more than five minutes old. Alternatively, if nodes are capable of detecting some network-wide event that happens with sufficient frequency, then they can refresh their keys every time such an event is detected, thus removing the requirement that their internal clocks be strictly synchronised. Finally, in networks possessing a base station capable of broadcasting directly to each node, the base station can simply send regular messages prompting the nodes to refresh their keys.

This scheme is very desirable in that there are no communication overheads. The existence of a suitable network-wide event is a strong (but widely satisfied) requirement: the more complex schemes discussed in subsequent sections are intended to be used when this requirement is not met.

4.2 Flooded Refreshing

Another solution to the problem of version number synchronisation is for a node to flood a key refresh signal throughout the network each time a key needs to be refreshed. The resulting communication overhead makes this infeasible in many instances; however, in our second scenario where much of the traffic involves messages being flooded throughout the entire network, the refresh signal can be 'piggy-backed' onto a flooded message. Each such flood then acts as a signal for all keys to be updated (hence the same version number is maintained for each key). The following scheme illustrates how this can be carried out, taking into account the fact that the flooding of separate messages may be simultaneously initiated at differing points of the network. The only communication overhead associated with refreshing in this manner is the need to append the version number to the encrypted message. (Even this overhead could be eliminated at the cost of nodes potentially having to perform several trial decryptions to determine the correct version number.)

Scheme 2. (Flooded refreshing)

1. *Before initiating the flooding of a message, a node first updates all its keys. It then encrypts the message under the new version of its keys before broadcasting it.*

2. *A node X receiving a flooded message encrypted with a version $i > \mathrm{vn}_k(X)$ of key k must update k in order to decrypt the message; it similarly updates the rest of its keys, then encrypts the message under these new versions before forwarding it. (Note that a node only forwards each message once; if it receives additional copies of the same message it simply ignores them.)*

3. *A node keeps a particular version of its keys until after it has broadcast a message using a higher version number. If a node receives several messages encrypted with different version numbers before it is able to forward them, it encrypts all the messages using the highest of these version numbers before rebroadcasting them. Once the messages have been sent it deletes all older versions of its keys.*

This scheme ensures that nodes only have to store multiple versions of the same key for the brief time between receiving a message and rebroadcasting it. If we assume that the media access control employed by the WSN prevents two neighbouring nodes from broadcasting simultaneously, then this manner of key updating prevents problems arising from nodes needing to use old versions of keys that they have already deleted. (Note that because of the small distances involved, we suppose that a message sent directly to a node by its neighbour is received instantaneously.)

Theorem 4.1. *If synchronous key refreshing is performed using Scheme 2 then no node receives a message encrypted with a version of a key that it has already deleted.*

Proof. Suppose a node A receives a message m rebroadcast by a neighbouring node B encrypted with version $\mathrm{vn}_k(B)$ of a key k possessed by A. Then A has version $\mathrm{vn}_k(A)$, and $\mathrm{vn}_k(A) \leq \mathrm{vn}_k(B)$, unless A has already rebroadcast some message using a version number higher than $\mathrm{vn}_k(B)$. However, in that case, A's neighbour B would have received that message prior to sending m, and thus $\mathrm{vn}_k(B) \geq \mathrm{vn}_k(A)$, which is a contradiction. □

In environments where a significant proportion of communication is local, Scheme 2 would incur an undesirable communication overhead. So we need to find schemes that flood the network less frequently.

5 The Asynchronous Case

The synchronous schemes discussed in Section 4 all have the advantage of ensuring that nodes sharing a given key maintain the same numbered version of that key. In our last three network environments, however, there are no sufficiently frequent network-wide events that would enable these schemes to be employed. In Subsection 5.1, we discuss an asynchronous scheme that can be used in these environments. The nature of the scheme means that we need to resynchronise the version numbers across the network occasionally, to prevent undue degradation of forward security. Subsections 5.2, 5.3 and 5.4 discuss methods for resynchronisation appropriate in environments 3, 4 and 5 respectively.

5.1 Asynchronous key refreshing

A simple method of *asynchronous key refreshing*, in which different nodes refresh their keys at different rates, is described as follows:

Scheme 3. (Message-driven refreshing)

1. *When two neighbouring nodes X and Y want to communicate using key k, X sends $\mathrm{vn}_k(X)$ to Y and Y sends $\mathrm{vn}_k(Y)$ to X.*

2. *X and Y each compute*
$$\mathrm{newvn} = 1 + \max\{\mathrm{vn}_k(X), \mathrm{vn}_k(Y)\}.$$

3. X and Y each update their copy of k by applying f an appropriate number of times, so that

$$\mathrm{vn}_k(X) = \mathrm{vn}_k(Y) = \mathrm{newvn}.$$

Then they use the updated key k to encrypt any information they wish to send to each other.

This scheme works well if all the nodes are more-or-less equally active, and hence update k at similar rate[1]. Even so, it is still possible that relatively inactive nodes do not update k very often. Thus, to avoid the degradation of forward security, a resynchronisation scheme must be deployed. Again, the method employed will depend on the network environment: we now discuss some possible methods.

5.2 Periodic Resynchronisation

The third category of networks discussed in Section 3 consists of those that experience regular events (such as a helicopter fly-past) that would be suitable for event-based key refreshing except that they do not happen with sufficient frequency. In such a context, the asynchronous refreshing Scheme 3 can be applied, but with the version numbers held by nodes being resynchronised each time the infrequent event is observed. A simple resynchronisation scheme requires all nodes to update their keys to a pre-specified version number upon detection of the event. For example, the j^{th} occurrence of the regular event could trigger each node to update their version number to the value $100j$ (assuming that no node will transmit more than 100 times between events). Thus less active nodes will "catch up" with highly active nodes once a day, maintaining some level of sychronicity on a regular basis. This technique is suitable as long as the amount of traffic likely to occur between consecutive occurrences of the event in question does not vary greatly and can be reasonably estimated. It has the advantage of incurring no communication overheads.

5.3 Resynchronisation by a flood

In applications where there are no network-wide events and the network can only support occasional flooding (see our fourth environment), a flooding technique could be used for resynchronisation rather than key refresh. So whenever a node has refreshed its key 100 times (say), it uses the flooded key refresh scheme from Section 4 to flood the network with a message requiring all nodes to update their keys to its version number. Flooding places an extra communication burden on the network, but this can be made manageable since the frequency of the floods is much lower than the frequency of key refresh operations. This scheme trades a degradation of forward security for an improvement in communication complexity.

5.4 Resynchronisation via a Leader Election

A third approach towards resynchronising keys in the absence of an appropriate network-wide event would be to periodically execute a protocol to resynchronise the network, by determining which node has the highest version number of a key k. (This is similar to the *leader election problem* that

[1] Due to the broadcast nature of wireless communication, it is also possible for any neighbours of the nodes involved in this exchange to learn the version number reached and refresh their own keys if necessary.

is studied in distributed systems.) Then every node would update their keys to this version[2]. This technique is useful in the fifth application environment of Section 3, in which the network may be temporarily disconnected. If the amount of traffic in each component varies then the key versions possessed by nodes in different components will differ. In order to resynchronise these versions once the components are reconnected, it will be necessary to excute a protocol of this nature.

There is a large literature describing algorithms for leader election in different settings. For our purposes, a variation of the algorithms described in Peleg [10] is appropriate, and we describe this algorithm below (Algorithm 1). This approach to resynchronisation is appropriate in situations when the network needs to run a protocol to establish some of its global properties (such as the shortest path to a sink node) in cases when the network is dynamic.

The algorithm has time complexity $O(D)$ and message complexity $O(DE)$, where D is the diameter of the network and E is the number of edges in the network. We describe an algorithm for leader election that can be initiated by any node x. The algorithm does not require that message transmissions be synchronised. The number of rounds (or *pulses*) is determined by the maximum distance of a node from the initiating node x, which we denote by dmax. The value of dmax does not have to be known ahead of time; indeed, the algorithm will compute it. We do not require that nodes have any knowledge of the structure of the network, except for the requirement that every node is assumed to know who all of its neighbours are. Note that if two nodes initiate the protocol simultaneously, it is easy to avoid any resulting conflicts by enforcing a standard rule for deciding which algorithm to drop.

Every node i has a value v_i; at the end of the algorithm, every node should know the value

$$\mathsf{vmax} = \max\{v_i\}.$$

In this algorithm, nodes broadcast tuples of the form (s, y, d, v), whose components are defined as follows:

- s is the node who is broadcasting the tuple, ($s = 0$ denotes a termination condition for the algorithm)
- y is the node at maximum distance (which is denoted by d) from the initiating node x, according to the current knowledge of s,
- v is the value of vmax, according to the current knowledge of s.

Algorithm 1.

1. *The first time node s receives a broadcast from any of its neighbours, it increments d by one and specifies itself as the node of maximum distance from x. It sets the value of v to be the maximum of v_s and the received value of v, then it broadcasts the tuple (s, s, d, v). This represents the first pulse for node s.*
2. *In subsequent pulses, the node s waits until it receives broadcasts from all of its neighbours (subsequent to its last broadcast). Then it updates d and v (and y, if necessary) based on the most recent set of tuples received, and broadcasts an updated tuple.*

[2]In general, it is not necessary for the refreshing of two distinct keys to be synchronised. For the sort of applications we are considering, however, it simplifies matters if all keys are refreshed at the same time. In particular this avoids any problems arising when the set of nodes that share a given key is disconnected.

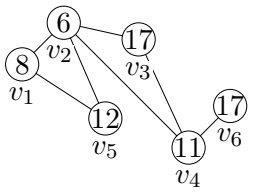

Figure 1: A network in which the nodes possess different versions of a key

3. *The initialising node x terminates the algorithm once there are two consecutive pulses in which the maximum received d-value does not change. This allows x to conclude that it has received information from every other node. It broadcasts the terminating condition ($s = 0$) in the form of the tuple $(0, y, d, v)$ in which $d = \mathsf{dmax}$, $v = \mathsf{vmax}$, and y has distance dmax from x.*

4. *Whenever a node receives a broadcasted tuple with $s = 0$, it rebroadcasts this tuple and terminates.*

Algorithm 1 can be used in conjunction with our asynchronous key refreshing scheme (Scheme 3). As it requires a substantial amount of communication between nodes it is perhaps most useful when performed occasionally, in response to a change in network conditions. For example, in the context of our fifth application environment, if the network becomes disconnected then Algorithm 1 can be applied in order to resynchronise key version numbers once connectivity is restored. We now give an example that demonstrates its behaviour.

Example 1. We present an example illustrating Algorithm 1. We use the graph in vertex set $\{1, \ldots, 6\}$ with edges 12, 15, 23, 24, 25, 34, 46 (Figure 1). The values stored in the nodes are $v_1 = 8$, $v_2 = 6$, $v_3 = 17$, $v_4 = 11$, $v_1 = 12$, $v_1 = 17$, and the initialising node is $x = 1$.

The tuples that will be broadcast during the execution of the algorithm are shown in Table 1. During the first pulse node 1 broadcasts the tuple $(1, 1, 0, 8)$ to initiate the algorithm. This is received by its neighbours, nodes 2 and 5. Node 2 has a lower version number than node 1, so it broadcasts the tuple $(2, 2, 1, 8)$. The first 2 denotes that the tuple is being sent by node 2, the second 2 and the 1 indicate that node 2 is at distance 1 from the initiating node, and that as yet it does not know of any nodes located further away. The 8 is the highest version number that node 2 has encountered so far. Similarly, during this second pulse node 5 broadcasts $(5, 5, 1, 12)$ to indicate that it is at distance 1 from node 1, including its own value for v as it is higher than that of node 1. This process continues until node 1 has received tuples with $d = 3$ in two consecutive pulses. Node 1 now knows that node 6 is the farthest node, and that the highest version number in the network is 17. It thus broadcasts the termination message $(0, 6, 3, 17)$, which is then rebroadcast by the other nodes in the network, until all nodes have received and rebroadcast this message.

6 Discussion

We have seen that the behaviour of a key refreshing scheme depends on the network environment in which it is to be applied. In Table 2 we summarise the properties of the schemes we have proposed for key refreshing and resynchronisation, as well as prior schemes appearing in the literature. The

Table 1: Example of the Leader Election Algorithm

	1	2	3	4	5	6
send	$(1,1,0,8)$					
receive		$(1,1,0,8)$			$(1,1,0,8)$	
send		$(2,2,1,8)$			$(5,5,1,12)$	
receive	$(2,2,1,8)$	$(5,5,1,12)$	$(2,2,1,8)$	$(2,2,1,8)$	$(2,2,1,8)$	
	$(5,5,1,12)$					
send	$(1,2,1,12)$		$(3,3,2,17)$	$(4,4,2,11)$		
receive		$(1,2,1,12)$	$(4,4,2,11)$	$(3,3,2,17)$	$(1,2,1,12)$	$(4,4,2,11)$
		$(3,3,2,17)$				
		$(4,4,2,11)$				
send		$(2,3,2,17)$			$(5,2,1,12)$	$(6,6,3,11)$
receive	$(2,3,2,17)$	$(5,2,1,12)$	$(2,3,2,17)$	$(2,3,2,17)$	$(2,3,2,17)$	
	$(5,2,1,12)$			$(6,6,3,11)$		
send	$(1,3,2,17)$		$(3,3,2,17)$	$(4,6,3,17)$		
receive		$(1,3,2,17)$	$(4,6,3,17)$	$(3,3,2,17)$	$(1,3,2,17)$	$(4,6,3,17)$
		$(3,3,2,17)$				
		$(4,6,3,17)$				
send		$(2,6,3,17)$			$(5,3,2,17)$	$(6,6,3,17)$
receive	$(2,6,3,17)$	$(5,3,2,17)$	$(2,6,3,17)$	$(2,6,3,17)$	$(2,6,3,17)$	
	$(5,3,2,17)$			$(6,6,3,17)$		
send	$(1,6,3,17)$		$(3,6,3,17)$	$(4,6,3,17)$		
receive		$(1,6,3,17)$	$(4,6,3,17)$	$(3,6,3,17)$	$(1,6,3,17)$	$(4,6,3,17)$
		$(3,6,3,17)$				
		$(4,6,3,17)$				
send		$(2,6,3,17)$			$(5,6,3,17)$	$(6,6,3,17)$
receive	$(2,6,3,17)$	$(5,6,3,17)$	$(2,6,3,17)$	$(2,6,3,17)$	$(2,6,3,17)$	
	$(5,6,3,17)$			$(6,6,3,17)$		
send	$(0,6,3,17)$		$(3,6,3,17)$	$(4,6,3,17)$		
receive		$(0,6,3,17)$	$(4,6,3,17)$	$(3,6,3,17)$	$(0,6,3,17)$	$(4,6,3,17)$
send		$(0,6,3,17)$			$(0,6,3,17)$	
receive			$(0,6,3,17)$	$(0,6,3,17)$		$(0,6,3,17)$
send			$(0,6,3,17)$	$(0,6,3,17)$		
receive						$(0,6,3,17)$

Scheme	Communication Overhead	Required Network Properties	Suitable Application Environments
Key Refreshing			
[9]	-	nodes communicate directly with the base station	
[6]	-	keys are shared by pairs of nodes	
1. Event-driven	-	frequent occurrence of a network-wide event	synchronised clocks
2. Flooded	+vn	frequent flooding of messages	frequent flooding
3. Message-driven	$2 \times$ vn	-	any
Resynchronisation			
Periodic	-	occasional network-wide event	infrequent network-wide events
Flooded	$O(n)$	capable of supporting occasional flooded messages	infrequent local events
Leader Election	$O(DE)$	-	regular disconnection

Table 2: A comparison of key refreshing and resynchronisation schemes. *+vn=key version number appended to each message; $x \times$ vn=x additional transmissions of vn per message; n=number of nodes; D=diameter of network; E=number of edges in network graph; for description of applications, see Section* 3

first four schemes have the advantage of incurring no communication overheads, although the scheme of [6] does involve a slight computational overhead, due to the need for trial decryptions. In the case of a network where there is pairwise communication with a base station, our event-driven scheme essentially reduces to the scheme of [9]; however, it is applicable in a wider range of environments, particularly any network where the nodes have synchronised clocks.

The remaining schemes do require extra communication, but are applicable in environments in which the first four schemes cannot be used. In the case of the flooded scheme this overhead is slight, as it is only necessary to append a key version number to each message that is flooded through the network. The final refreshing scheme (message-driven refreshing) is more costly, as two version numbers have to be transmitted before each message is sent. However, it can be used in any network environment, and hence can be employed in networks that do not have the necessary properties for the other schemes to be applied. Similar observations can be made regarding the resynchronisation schemes.

There are several issues concerning key refreshing in a WSN context that merit further research. In some WSNs it is customary to deploy an excess of nodes that then spend part of their time in a 'sleep' state. Such nodes have the potential to degrade forward security if they are asleep through several key refresh events. One solution might be to mandate that nodes refresh their keys numerous times before entering the sleep state, however overall network-wide management of this process requires further investigation. Also, nodes in a WSN have relatively high failure probabilities, whether due to battery exhaustion, destruction, or simple malfunction. It would be interesting to investigate ways of limiting the degradation of forward security due to the results of node failure. Finally, many WSNs have specific topologies (such as hierarchal networks) for which it may be possible to devise dedicated key refreshing schemes that perform more efficiently than the general ones proposed in this paper.

References

[1] M. Bellare and B. Yee. Forward-Security in Private-Key Cryptography. *Topics in Cryptology – CT-RSA '03*, LNCS 2612, Springer-Verlag, (2003), 1–18.

[2] L. Blum, M. Blum, M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, **15(2)** (1986), 364–383.

[3] S.A. Çamtepe and B. Yener. Key distribution mechanisms for wireless sensor networks: a survey. *Rensselaer Polytechnic Inst. Tech. Rep. TR-05-07*, (2005).

[4] L. Eschenauer and V.D. Gligor. A key-management scheme for distributed sensor networks, *Proceedings of the 9th ACM conference on Computer and Communications Security*, (2002), 41–47.

[5] C.G. Günter. An identity-based key-exchange protocol, *Advances in Cryptology — Eurocrypt '89*, LNCS 435, Springer-Verlag, (1990) 29–37.

[6] M. Klonowski, M. Kutyłowski, M. Ren and K. Rybarczyk. Forward-secure key evolution in wireless sensor networks. *Cryptology and Network Security 6th International Conferences — CANS 2007*, LNCS 4856, Springer-Verlag, (2007), 102–120.

[7] J. Lee and D.R. Stinson. On the construction of practical key predistribution schemes for distributed sensor networks using combinatorial designs. To appear in *ACM Transactions on Information and System Security*.

[8] K. M. Martin and M. B. Paterson. An application-oriented framework for wireless sensor network key establishment. *WCAN 2007*, to appear in *ENTCS* (2007).

[9] S. Mauw, I. van Vessen and B. Bos. Forward secure communication in wireless sensor networks. *SPC 2006*, (2006), 32–43.

[10] D. Peleg. Time-optimal leader election in general networks, *Journal of Parallel and Distributed Computing*, **8** (1990), 96–99.

[11] M. Ren, T.K. Das and Jianying. Diverging keys in wireless sensor networks. *ISC 2006*, LNCS 4176, Springer-Verlag, (2006), 257–269.

[12] K. Römer, P. Blum and L. Meier. Time synchronization and calibration in wireless sensor networks. In I. Stojmenovic (Ed.): *Handbook of Sensor Networks: Algorithms and Architectures*, Wiley and Sons, (2005), 199–237.

[13] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications Magazine*, **11(6)** (2004), 54–61.

[14] Y. Xiao, V.K. Rayi, B. Sun, X. Du, F. Hu and M. Galloway. A survey of key management schemes in wireless sensor networks. *Computer Communications*, **30** (2007), 2314–2341.