Implementation of Pair-Wise Key Pre-distribution Schemes on Wireless Sensor Devices

Independent Study Report

Instructor: Dr. James Minseok Kwon

Jeegar Brahmakshatriya

jvb3350@rit.edu

3/30/2009

Table of Contents

1. Introduction

Wireless sensor networks are one of the first real world examples of pervasive computing, the notion that small, smart and cheap sensing and computing devices will eventually permeate the environment [8]. Wireless sensor network (WSN) consists of a large number of ultra small sensor nodes. Each sensor node is an autonomous battery operated device with data processing capabilities, integrated sensors, limited memory and a short range radio communication capability. In application scenarios sensor nodes are randomly deployed over a region and collect data.

Wireless Sensor Networks are deployed for a wide variety of applications like military tracking, monitoring of environment, smart environments, patient tracking, etc. [1]. Security is extremely important when sensor nodes are deployed in hostile environments because they may be exchanging valuable or critical information about the environment and an adversary can use this information to his advantage or inject malicious information into the network. Apart from physical capture a malicious user can easily tap into the wireless communication and listen to the traffic, inject misleading data into the network or impersonate as a node of the network. To provide security, encrypted and authenticated communication is required. Active research is being pursued for efficient setup of secure keys in wireless sensor networks. Setting up of keys for secure communication is a part of the Key Management problem.

In general network environments there are three types of key agreement schemes: trusted server scheme, self enforced scheme and pre-distribution scheme. The trusted server scheme has a trusted server between two nodes to negotiate a shared key between the nodes. This scheme is not feasible in sensor networks because there is no central server in most WSN. Self enforcing scheme uses public key algorithms such as Diffie-Hellman key agreement or RSA. It is not desirable to use these schemes due to limited computation capabilities and energy constraints. The last scheme is pre-distribution scheme in which keys are distributed to the nodes prior to deployment. This is the most feasible scheme for WSN because of its lower resource utilization [2].

Pre-distribution scheme uses secret keys to establish pairwise keys after they are deployed. Using a master key between all nodes is not a good idea because if a malicious user obtains the key entire network is compromised. Another approach is to pre deploy the nodes with secret keys for nodes. This makes the network more resilient to node capture but it is not scalable because if a node is to be inserted in the network all the nodes that are already deployed have to be provided with a new set of keys. Also it would require lot of memory because in a network with *n* nodes each node would be required to store (n-1) keys even if they might not communicate with most of other nodes thorough out their life time. To have a scalable and resilient network, nodes should be able to establish secret keys using the information that is assigned to the node prior to the deployment. In this report, we discuss implementation of two schemes to establish secret keys: "*Lightweight Key Management in Wireless Sensor Networks by Leveraging Initial Trust*" [3] and "*q-composite Random Key pre-distribution Scheme*" [4].

Authentication and key setup requires initial trust between the sensor nodes. A public key is considered valid only if it is signed by a trusted authority. For symmetric key setup, the nodes setting up the key should somehow acquire a common secret that will enable them to identify each other in order to ensure secure communication between them. In traditional networks these secrets are setup manually. For example, when a central authentication server is used, an administrator distributes a shared key when a client registers with the server. To ensure security the secret key is communicated offline [3].

Setting up a large number of keys is impractical in case of Wireless Sensor Networks because they are wide spread and typically deployed at remote locations. For example, sensor nodes can be air-dropped in regions that are inaccessible. In such scenarios setup information has to be configured prior to deployment and after deployment. The network should also be self organizing and self-sufficient. The key management should then be managed by the sensor nodes without human involvement [3].

2. Lightweight Key management in Wireless Sensor Networks by Leveraging Initial Trust

Typically sensor nodes are deployed in a set of *S* nodes. Post deployment the nodes should be able to discover their neighbors and organize an ad-hoc network. During setup phase the primary concern is to setup a secret key without intervention of a malicious node. Initially the nodes from set *S* can be assumed to be trustworthy, as it takes some time for an adversary to compromise the nodes [3]. A quick establishment of trust is important because the risk of compromising a node increases with time. In this scheme a bootstrapping algorithm is used which builds initial trusted links between neighboring sensor nodes. Once the nodes have been deployed, the bootstrap algorithm is executed in a short time.

To setup a secure link between neighbors we do not need a very strong authentication protocol. A weaker authentication like ability of a node to prove that it belongs to set *S* should be sufficient, because all nodes of set *S* are assumed to be initially trustworthy. Hence node *A* and *B* can setup a secure link if they can ascertain that they belong to the same set of nodes *(S).* This type of authentication can be performed inexpensively. Each node in a set is provided with a group authentication key $bk_{1,}$ and a key-generation key denoted by $bk_{2.}$ These two keys are distributed in the nodes before deployment with negligible overheads.

2.1 Protocol Description

The authentication protocol is simple. A sensor node, say *A*, initiates the protocol. It generates a random nonce N_a and broadcasts a *hello* message. The format of the message is as follows:

$$Hello\ Message: \{Hello, A, N_a, MAC_{bk1}(\ Hello, A, N_a\ )\ \}$$

The hello message contains node *A*'s identity, nonce generated by *A* (N_a), and a Message Authentication Code (MAC) generated using key bk_1. On receiving a hello message any node from the set *S* can check if the MAC is valid. To check the validity, the node would decrypt the MAC using the group authentication key bk_1 that it has. On decrypting the message it should be able to verify that node *A*'s identity and N_a that were not encrypted using bk_1 are the same as the node *A*'s identity and N_a that it got after decrypting. If they are the same, then it can be sure that both the nodes are from the same set *S.* Let node *B* be one such node. By this exchange, node *B* has ascertained that node *A* is from the set *S,* and it will generate a random nonce N_b and send an acknowledgement to node *A*. The acknowledgement message is as follows:

$$Acknowledge\ Message: \{Ack, A, B, N_b, Mac_{bk1}(Ack, A, B, N_b, N_a\ )\ \}$$

By this acknowledgement node *A* can ascertain if node *B* is from the same set *S* after decoding the MAC using the group authentication key bk_1. Node *A* can also verify the nonce N_a which it had sent to node *B*. Now at this point both the nodes *A* and *B* have nonces N_a and N_b. They can now create a pairwise symmetric key as follows:

$$Pairwise\ symmetric\ Key\ K_{ab} = G_{bk2}(N_a, N_b)$$

Both nodes use the key-generation key bk_2 along with a keyed one-way hash function *(G)* to generate a pairwise key. This pairwise key is used by nodes *A* and *B* for communicating securely.

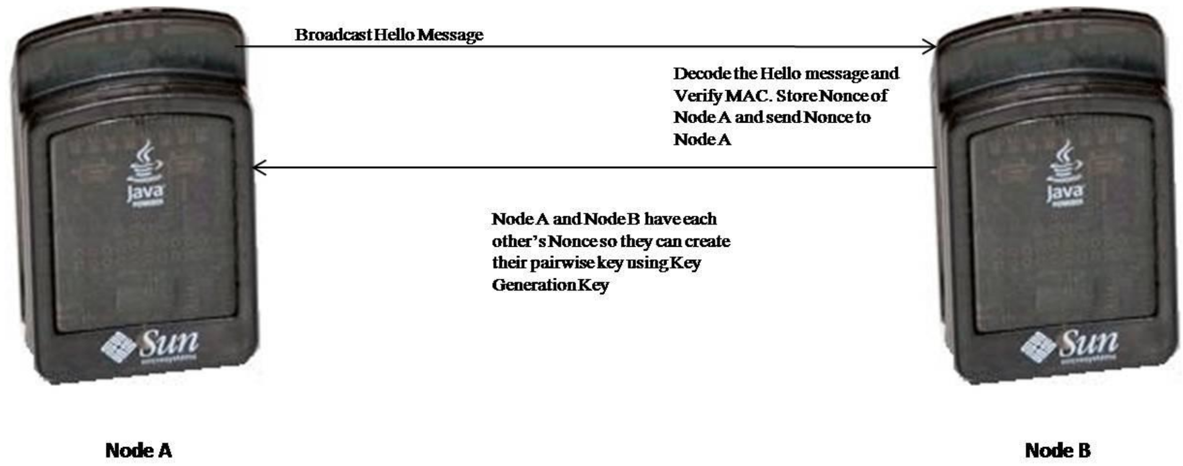

Figure 1 : Working of Light Weight Key management Scheme

2.2 Security

It can be proven that this bootstrapping protocol is secure against an intruder who does not know the bk_1 and bk_2 keys using models and techniques introduced by Bellare and Rogaway in [5]. The essence of the proof lies in the assumption that *G* and *MAC* are of pseudorandom function families. By this assumption we can show that for any intruder *E*, initiator node *A*, and responder node *B*, the following properties are satisfied:

- The probability of node *B* accepting a hello message that appears to be initiated by node *A* but was not sent by node *A* is negligible.
- The probability of node *A* accepting an acknowledgement message that appears to be sent by node *B* but was not from node *B* is negligible.
- Intruder *E* is unable to differentiate between the key $K_{a,b}$ and a random bit string of the same length.

This shows that the protocol is resilient against an intruder *E* who can listen to communication and inject malicious messages as long as *E* does not know the bootstrapping keys bk_1 and bk_2. The drawback of this protocol is that the nodes are not tamperproof. An intruder can possibly obtain the keys by physically compromising a node. If an intruder obtains the keys bk_1 and bk_2 during the bootstrap time window, it can easily compute the pairwise keys K_{ab} from the messages that are exchanged.

To counter this drawback, the bootstrapping window should be kept short. But if the intruder can record messages communicated between node *A* and *B* when bootstrap is being executed, it can later discover key bk_2. On acquiring the bk_2 key compromise of a large number of pairwise keys is risked because all nodes use the same key-generation key. To counter this kind of attacks, keys bk_1 and bk_2 are erased as soon as the bootstrap time window is up.

3. q-Composite keys random key pre-distribution Scheme

Eschenaur and Gilgor have proposed a random key pre-distribution scheme addressing the bootstrap problem [6]. Working of this scheme is as follows. An initialization phase is performed on each sensor node before its deployed. In this phase a random set of keys S is picked out of all the possible key space. Each sensor node is assigned with m random keys selected randomly from the key set S. We will call this set of m keys the node's key ring. The number of keys in the key pool, |S| is chosen so that two random subsets of size m in S will share at least one key with probability p.

Once the sensor nodes are deployed key-setup phase is performed. Key-discovery is performed by each node to find out with which of their neighbors it shares a key. To facility the key-discovery phase each key is assigned a short identifier to prior to deployment. On discovering that a node shares a key with its neighbor, the nodes use challenge-response protocol to verify that its neighbor actually holds the key. On confirming the shared key it becomes the key for that link [6].

In the above scheme any two neighbors need to find only one common key from their key rings to establish a secure link. This makes the network less resilient to node capture. To increase the network's resilience against node capture, the authors of [4] suggest that instead of just one key q common keys would be required for key-setup. Higher network resilience is achieved by increasing the amount of overlapping keys required. As the amount of overlapping keys required increases, it becomes exponentially harder for an attacker to compromise a link. But to preserve the given probability p of two nodes sharing sufficient keys for establishing a secure link, it is necessary to reduce the size of the key pool $|S|$. This would allow an adversary to gain a larger sample of S by compromising fewer nodes. Tradeoffs have to be made among these two opposing factors. We would like to have an optimal amount of overlapping keys to provide the greatest obstacle for an adversary and some desired probability of eavesdropping on a link.

3.1 Protocol Description

The q composite keys scheme authentication protocol is divided into three phases: Initialization phase, key discovery and key setup.

In the initialization phase out of the total key space, a set S of random keys is selected. Each sensor node is provided with m random keys selected from the set S (m is the number of keys in the key ring of each node). These m keys are stored in the sensor node. In the key discovery phase each node discovers all common keys it shares with each of its neighbors. This can be done by a simple broadcast. Broadcast based key discovery is simple and straightforward to implement but it has a huge drawback that an eavesdropper can identify the key sets of all the nodes in a network and can easily compromise sensor nodes. To make key discovery more secure key discovery phase uses client puzzles such as Merkle puzzle [7].

For securing key discovery phase, each sensor node has m client puzzles (one puzzle for each of the m keys). Sensor node A would send a broadcast message with all its client puzzles. Node B would know how to solve the puzzles as they are from the same group. Node B randomly selects one puzzle from the m puzzles and solves it. Node B acknowledges node A's broadcast with the solution to the puzzle and the index of the puzzle. Node A verifies the solution of the puzzle and if the solution is correct, the solution is used as a key for exchanging the shared keys. Using client puzzles is secure because solving the puzzles is computationally hard but not impossible. If an adversary tries brute force on the puzzles, it is computationally very difficult because he does not know what operations are to be done on the puzzle. Also node B would randomly choose a puzzle out of the m puzzles, so to compromise the link an adversary has to use brute force on all the keys. This being computationally difficult will take an adversary a long time.

Once a temporary key has been created this key is used for key setup. Using the temporary key node *A* sends its key ring to node *B*. Node *B* checks if it has at least *q* common keys with node *A*. If there are at least *q* keys common between them node *B* sends its key ring to node *A*. Node *A* again verifies for common keys. Once it is established that node *A* and node *B* share *q* common keys a new communication link key *K* is generated. To generate *K* all the common keys are hashed. The keys can be hashed in a canonical order, for example based on the original order in the key set *S*. If the number of common keys is less than *q* key-setup is not performed.

$$K = hash\ (k_1||k_2||\ \ldots\ k_{q'})$$

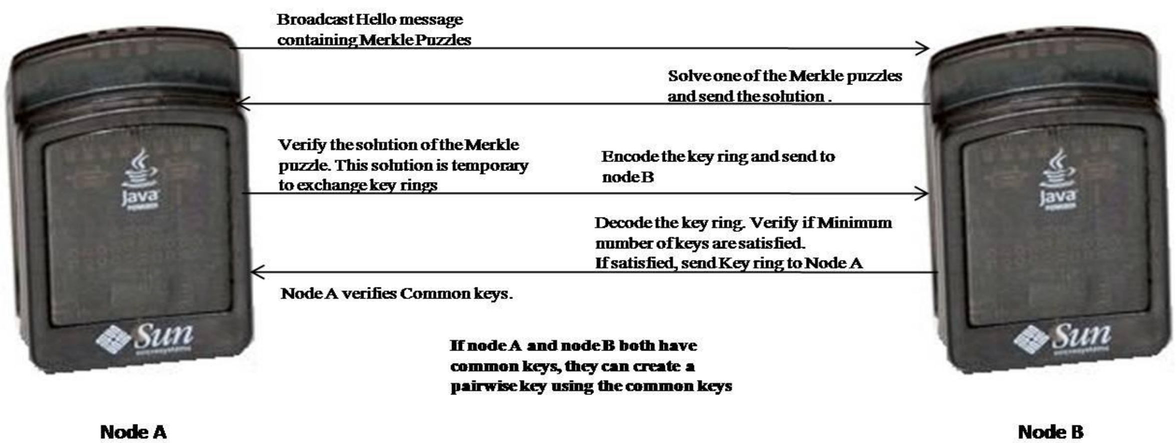

Figure 2: Working of q-Composite keys random key pre-distribution Scheme

3.2 Security

The *q*-composite keys scheme's resilience to node capture depends on the number of shared keys and the number of nodes being captured. If the number of shared keys is high an adversary can get access to higher number of shared keys by compromising fewer sensor nodes. Where as if the number of shared keys is very less an adversary can easily break a link with a given key set [4]. Hence it is very important to choose the number of shared keys such that it poses greatest obstacle for an adversary while keeping some desired probability of eavesdropping on a link.

4. Implementation

The above mentioned key management schemes are divided into two phase: Pre-deployment phase and post deployment phase. The pre-deployment phase is performed on a platform with high resources (like a desktop computer). The machine should have enough memory to store the keys that are generated. It should also have enough processing capacity to be able to generate keys with required probability so that two nodes share sufficient keys such that the probability of network graph being connected is high. The post-deployment phase is performed on wireless sensor devices (Sun SPOT devices) which have limited resources. Typically they are battery operated, have limited memory and processing capacity and use wireless radio for communication.

4.1 System Architecture:

Both the above mentioned key management schemes use same system architecture.

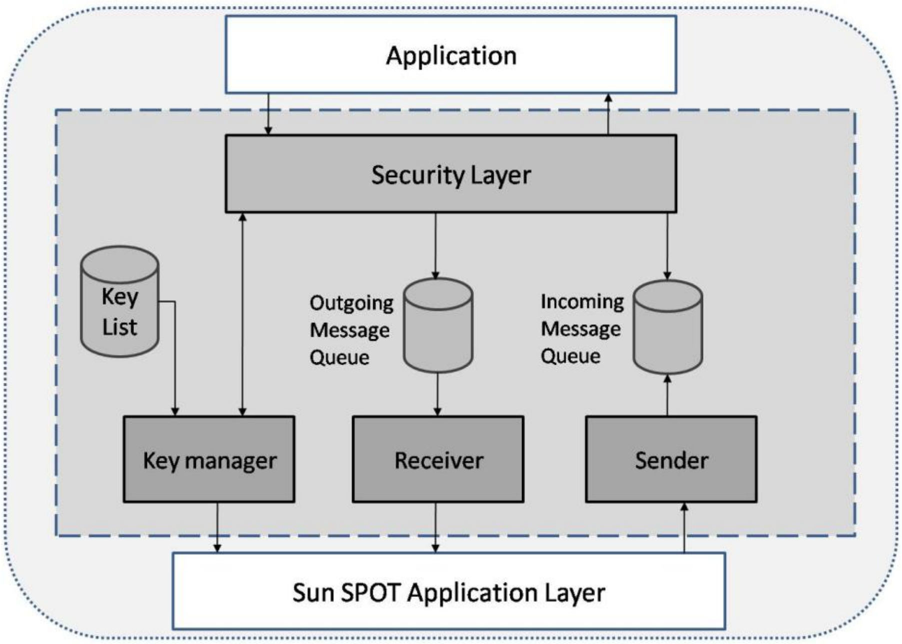

Figure 3: System Architecture for Key Management Schemes.

The application on the Sun SPOT device interacts with the key manager layer and the key manager layer interacts with the application layer of the Sun SPOT device. When one sensor node (say *A*) wants to send a message to another sensor node (say *B*), a message is sent from the application to the key manager layer. The key manager layer in node *A* checks if it already has a key setup for node *B*. If it has the key for node *B* it encrypts the message using the secret key and sends the message to outgoing message queue. When the Sun SPOT application layer is ready to send a message it picks up the next message from the outgoing queue and sends it to node *B*. If key manager of node *A* does not have a key for node *B*, key setup is initialized.

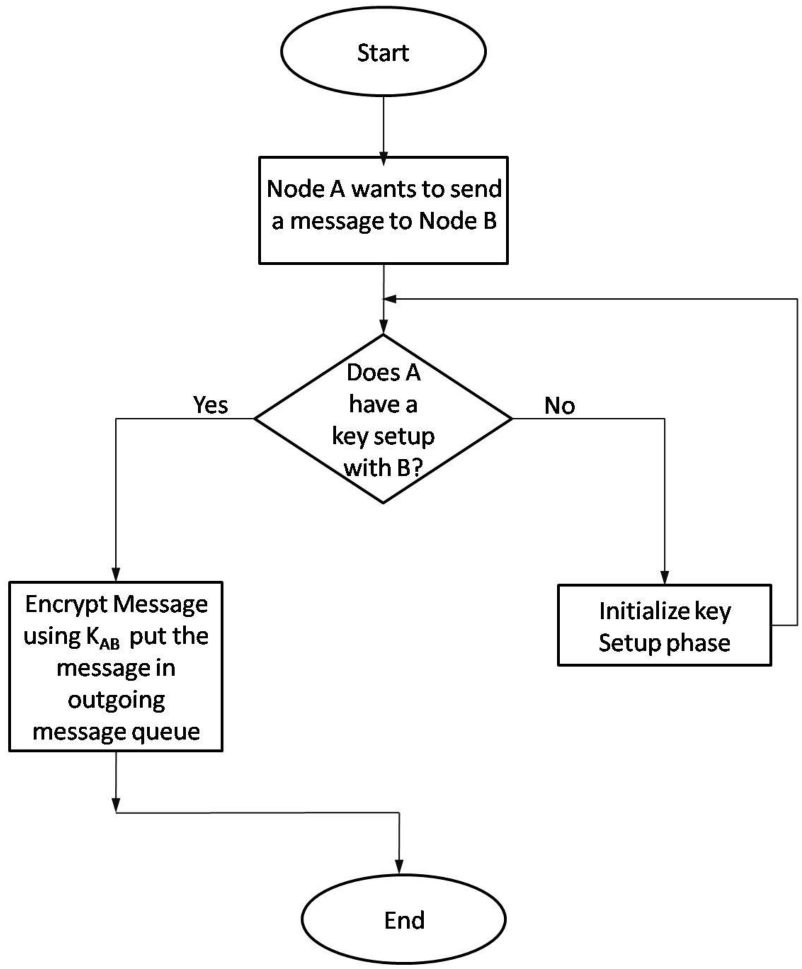

Figure 4: Outgoing Message Flow Chart

The incoming message queue is filled up by the Sun SPOT application layer. When a message is received from another Sun Spot the application layer puts the message in the incoming message queue. The key manager layer picks up a message from the incoming message queue and checks what kind of message it is. If the message is for the application it checks if it has a key setup with the sender node, if it finds the key, the message is decrypted using the key and sent to the application message, otherwise the message is discarded as it can possibly be from an adversary who is trying to infiltrate the network. If the message is for key setup the message is sent to key manager layer and it sends an appropriate acknowledgement to the sender.

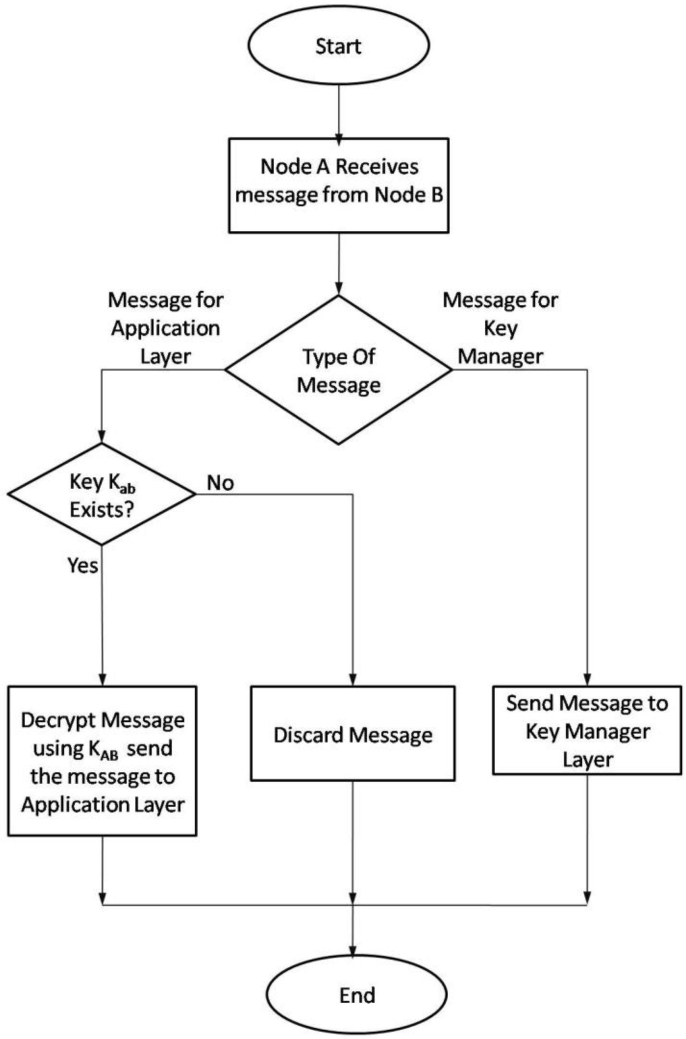

Figure 5: Incoming Message Flow Chart

4.2 Implementation of Lightweight key management scheme.

Lightweight key management scheme uses a group authentication key bk_1 and a key-generation key bk_2. Each node in set *S* has the same authentication and key-generation key. The authentication key is used to prove that node *A* is from the same set *S* as node *B*. There is a small time window for the nodes to setup a pairwise key with their neighbors, so as soon as the nodes are deployed the bootstrapping procedure is started. Suppose node *A* initiates the protocol. Node *A* sends a broadcast message containing a *hello* message. The message contains node *A*'s id, a nonce generated by node *A* (N_a), and a message authentication code (MAC). MAC is generated by encoding *Hello*, node *A*'s id and N_a with authentication key bk_1. On receiving this message, node *B* decodes the message using its own bk_1 and compares the decoded information with the non encoded information. If the information matches, node *B* can sure that node *A* is from the same set *S*. Node *B* responds with an acknowledgement message containing *Acknowledgement*, node *A*'s id, node *B*'s id and node *B*'s nonce (N_b) along with MAC. Again the MAC is generated by encoding *Acknowledgement,* node *A*'s id, node *B*'s id and N_b with bk_1. On receiving an acknowledgement message, node *A* decodes the MAC using bk_1 and compares the decoded information with the non encoded information. If the information matches node *A* can be sure that node *B* is from the same set S. Now both node A and node B have nonce N_a and N_b. node A and node B can now generate a pairwise key (K_{ab}) using these nonce. To generate K_{ab} a keyed one-way hash function is used. They use the key-generation key bk_2 for this purpose. For secure communication node A and node B use key K_{ab}.

Once the time window for bootstrap is over all the nodes in set S delete the authentication and key-generation key. This ensures that in an event of node capture an adversary cannot get the authentication and key-generation key. After the time widow elapses no more key setup requests are accepted.

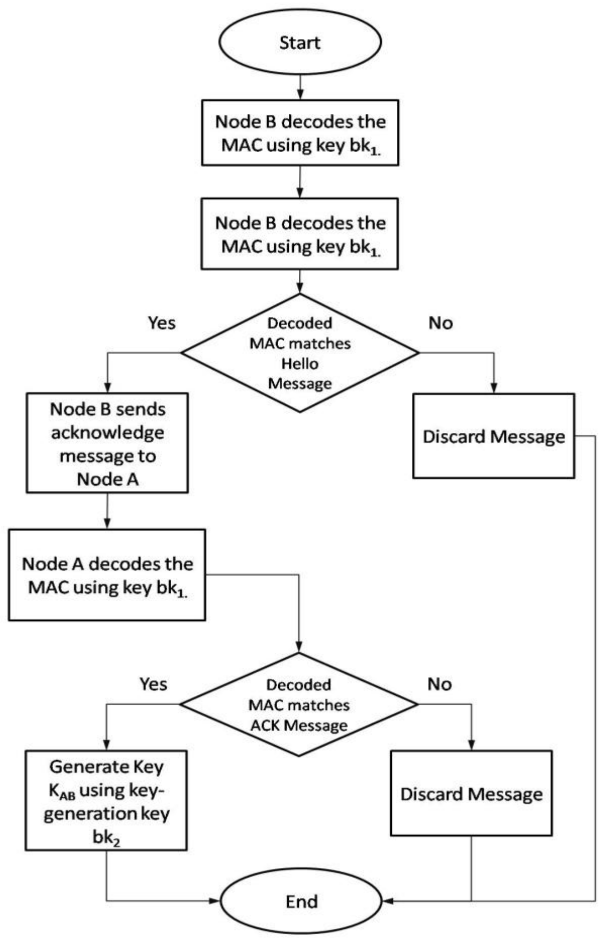

Figure 6: Key setup Flow Chart

4.3 Results of Light weight key management scheme

Output for Node *A.* (A Initiates Key setup)

```
[java] Initializing Application...
[java] Waiting for response from neighbors...
[java] SECURE_LAYER :: Initializing...
[java]
[java] KEY_MANAGER :: Initializing...
[java] KEY_MANAGER :: Node ID = 1                                  Node Startup Processes
[java]
[java] SENDER :: Initializing...
[java] RECEIVER :: Initializing...
[java] SECURE_LAYER :: Start Receiving thread...
[java] SECURE_LAYER :: Booting up for key discovery...
[java]
[java] KEY_MANAGER :: Booting up...
[java]
[java] KEY_MANAGER :: Broadcasting to neighbors (Round 1): 1-8061533852060732353 1-8061533852060732353    Broadcast Key Setup Request
[java]
[java] RECEIVER :: Receiving: 1 2-5716790770726554523 1 2-5716790770726554523-806153385206073235,
              from Node at address: 0014.4F01.0000.129D
[java] RECEIVER :: Adding a message to incoming messages queue...     Received Key Setup Reply
[java] RECEIVER :: Handling incoming messages...
[java] SECURE_LAYER :: Received message...
[java]
[java] SECURE_LAYER :: Key Setup Reply received...
[java]
[java] SECURE_LAYER :: Received message...
[java]
[java] SECURE_LAYER :: Message received for key setup...
[java] SECURE_LAYER :: Source ID = 2
[java] SECURE_LAYER :: Nonce A = 571679077072655452               Processing Key Setup Reply. Verify Mac and Setup key
[java] SECURE_LAYER :: MAC Verified. Setting up new pairwise key...
[java] KEY_MANAGER :: Setting up Key for Node = 2 at Address  = 0014.4F01.0000.129D
[java] KEY_MANAGER :: Key setup established...
[java] RECEIVER :: Waiting for incoming messages...
[java] RECEIVER :: No incoming messages...
[java] RECEIVER :: Waiting for incoming messages...
[java] RECEIVER :: got something...
[java] RECEIVER :: Receiving: 2-5716790770726554523 2-5716790770726554523,     Received Key Setup Request
              from Node at address: 0014.4F01.0000.129D
[java] RECEIVER :: Adding a message to incoming messages queue...
[java] RECEIVER :: Handling incoming messages...
[java] KEY_MANAGER :: Sending Key Setup Reply to 0014.4F01.0000.129D    Send Key Setup Reply
[java]
[java] KEY_MANAGER :: Active region, neighbors found...
[java] SECURE_LAYER :: Boot up process accomplished, starting Sending thread...
[java] Application Initialized and ready...                       Boot process successful
[java] SECURE_LAYER :: Waiting for data reception notification...
```

Output for Node B. (B responds to A's Key Setup Request)

```
[java] Initializing Application...
    [java] Waiting for response from neighbors...
    [java] SECURE_LAYER :: Initializing...
    [java]
    [java] KEY_MANAGER :: Initializing...
    [java] KEY_MANAGER :: Node ID = 2
    [java]
    [java] SENDER :: Initializing...
    [java] RECEIVER :: Initializing...
    [java] SECURE_LAYER :: Start Receiving thread...
    [java] SECURE_LAYER :: Booting up for key discovery...
    [java]
    [java] KEY_MANAGER :: Booting up...
    [java]
    [java] RECEIVER :: got something...
    [java] RECEIVER :: Receiving: 1-806153385206073235 31-8061533852060732353,
                    from Node at address: 0014.4F01.0000.1271
    [java] RECEIVER :: Adding a message to incoming messages queue...
    [java] RECEIVER :: Handling incoming messages...
    [java] SECURE_LAYER :: Received message...
    [java]
    [java] SECURE_LAYER :: Message received for key setup...
    [java] SECURE_LAYER :: Source ID = 1
    [java] SECURE_LAYER :: Nonce A = 806153385206073235
    [java] SECURE_LAYER :: MAC Verified. Setting up new pairwise key...
    [java] KEY_MANAGER :: Setting up Key for Node = 1 at Address  = 0014.4F01.0000.1271
    [java] KEY_MANAGER :: Key setup established...
    [java] KEY_MANAGER: Sending KeysetupReply.
    [java] KEY_MANAGER :: Sending Key Setup Reply to 0014.4F01.0000.1271
    [java]
    [java] RECEIVER :: No incoming messages...
    [java] RECEIVER :: Waiting for incoming messages...
    [java] KEY_MANAGER :: Active region, neighbors found...
    [java] SECURE_LAYER :: Boot up process accomplished, starting Sending thread...
    [java] Application Initialized and ready...
    [java] SECURE_LAYER :: Waiting for data reception notification...
```

Node Startup Processes

Received Key Setup Request from Node A

Processing Key Setup Request

Verify Mac to ensure the node A is from same Set S

Send Key Setup Reply to Node A

Boot process successful

4.4 Implementation of q-composite keys random key pre-distribution scheme.

q-Composite keys random key pre-distribution scheme (as described above) consists of three phases: Initialization phase, key discovery and key setup. In the initialization phase a set (say *S*) of random keys is generated. Each node is provided with *m* keys selected randomly from set *S*.

To setup a pair-wise key two nodes (say *A* and *B*) need to have at least *q* common keys in their key ring. In key discovery phase nodes *A* and *B* exchange their key rings and check if they have at least *q* common keys. If key rings are exchanged over the air without encoding an adversary can easily obtain the key rings and compromise a large set of nodes. To overcome this problem each node is also provided with a set of client puzzles. The number of puzzles is equal to the number of keys in the key ring. Before key rings are exchanged a temporary key has to be setup. To setup this key node *A* sends a broadcast message with all its client puzzles. On receiving the client puzzles node *B* randomly selects one of the puzzles and solves it. Node *B* sends an acknowledgement message to node *A* which contains node *B*'s id, id of client puzzle that was solved and the solution to the puzzle. On receiving the message node *A* verifies the solution and if the solution is correct, the solution is used as a temporary key to exchange key rings. In the implementation, we have used a simple mathematical function for client puzzles. This is not computationally difficult yet is difficult for an adversary to brute force a solution.

Once the temporary key is setup node *A* encodes its key ring using the temporary key and sends it to node *B.* Node *B* decodes the key ring, extracts the keys and checks if it has at least *q* common keys with node *A*. If it does node *B* encodes its key ring and sends it to node *A*. On receiving the key ring node *A* decodes the key ring and checks for minimum common key requirement. Once it is ascertained that node *A* and node *B* share at least *q* common keys they proceed to create a pair-wise key. To generate a pair-wise key the common keys are hashed in a canonical order.

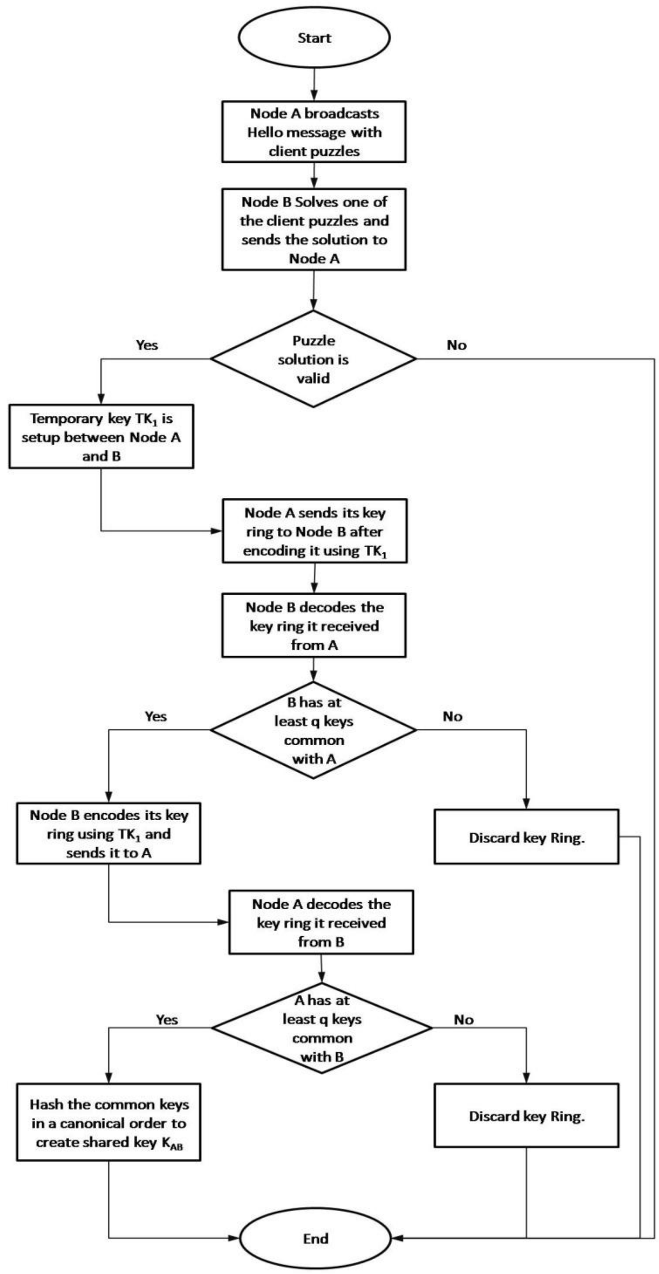

Figure 7: q-Composite flow chart.

4.5 Results of q-composite keys random key pre-distribution scheme.

Output for node *A* (*A* initiates Key Setup)

```
[java] Initializing Application...
[java] SECURE_LAYER :: Initializing...
[java]
[java] KEY_MANAGER :: Initializing...
[java]
[java] SENDER :: Initializing....
[java] RECEIVER :: Initializing...
[java] SECURE_LAYER :: Start Receiving thread...
[java] SECURE_LAYER :: Booting up for key discovery...
[java]
[java] KEY_MANAGER :: Booting up...
[java] RECEIVER :: Waiting for incoming messages...
[java] RECEIVER :: No incoming messages...
```

Node Startup Processes

```
[java]
[java] KEY_MANAGER :: Broadcasting to neighbors (Round 1): 2 0000000000123456789 0000000000912345678
                 0000000000891234567 0000000000789123456 0000000000678912345 0000000000567891234
```

Broadcasting Key Setup Request

```
[java]
[java] RECEIVER :: Receiving: 21061728394,  from Node at address: 0014.4F01.0000.1271
[java] RECEIVER :: Adding a message to incoming messages queue...
[java] RECEIVER :: Handling incoming messages...
[java] SECURE_LAYER :: Received message...
```

Process message received from Node B

```
[java]
[java] SECURE_LAYER :: Message Reply received for Merkle Key setup...
[java]
[java] Merkle Puzzle Solution is Correct
```

Merkle Key setup reply Received. Verify solution

```
[java]
[java] Adding Merkle Key
[java] Source ID = 1 || merkle result = 61728394
```

Save the merkle key

```
[java]
[java] Sending Q Keys:
[java] KeyString = 5646557890123456789 5464456789012345678 8912345678901234567
                 7891234567890123456 6789123456789012345 5678912345678901234
[java]
[java] KEY_MANAGER :: Sending Key Setup Reply to 0014.4F01.0000.1271
```

Encode Q keys in Merkle Key and send it to Node B

```
[java]
[java] RECEIVER :: No incoming messages...
[java] RECEIVER :: got something...
[java] RECEIVER :: Receiving: 2 1 1234567890123456789 9123456789012345678 8912345678901234567 7891234567890123456
                 6789123456789012345 5678912345678901234,    from Node at address: 0014.4F01.0000.1271
[java] RECEIVER :: Adding a message to incoming messages queue...
[java] RECEIVER :: Handling incoming messages...
[java] SECURE_LAYER :: Received message...
```

Received Message from Node B

```
[java]
[java] SECURE_LAYER :: 2nd Qkeys Exchange ...
[java]
[java] Length of QKEYS = 6
[java] KEYMANAGER: Number of Common Keys = 4
[java] Putting value in idKeyList
[java] SRCID :::::: 1 KEY VALUE :::::: 1744571290487985417
```

Create and save the key after verification of minimum common keys.

```
[java] KEY_MANAGER :: Key setup established...
[java] KEY_MANAGER :: Active region, neighbors found...
[java] SECURE_LAYER :: Boot up process accomplished, starting Sending thread...
[java] Application Initialized and ready...
```

Key setup successfully completed

Output For node *B* (*B* responds to *A*'s Key Setup Request)

```
[java] Initializing Application...
[java] SECURE_LAYER :: Initializing...
[java]
[java] KEY_MANAGER :: Initializing...
[java]
[java] SENDER :: Initializing....
[java] RECEIVER :: Initializing...
[java] SECURE_LAYER :: Start Receiving thread...
[java] SECURE_LAYER :: Booting up for key discovery...
[java]
[java] KEY_MANAGER :: Booting up...
[java] RECEIVER :: Waiting for incoming messages...
[java] RECEIVER :: No incoming messages...
```

Node Startup Processes

```
[java]
[java] RECEIVER :: Receiving: 2 00000000001234567890 000000000912345678 000000000891234567
000000000789123456 000000000678912345 000000000567891234,    from Node at address: 0014.4F01.0000.129D
```

Message received from Node A

```
[java]
[java] RECEIVER :: Adding a message to incoming messages queue...
[java] RECEIVER :: Handling incoming messages...
[java] SECURE_LAYER :: Received message...
[java]
[java] SECURE_LAYER :: Message received for Merkle Key setup...
```

Process message received from Node A

```
[java]
[java] Adding Merkle key Result
[java] Source ID = 2 || merkle result = 61728394
[java]
[java] KEY_MANAGER :: Sending Key Setup Reply to 0014.4F01.0000.129D
[java] Data : 21061728394
```

Solve the Merkle Puzzle and send reply to Node A

```
[java]
[java] RECEIVER :: Receiving: 1 2 564655789012345678 9 546445678901234567 8 891234567890123456 7
789123456789012345 6 678912345678901234 5 567891234567890123 4,    from Node at address: 0014.4F01.0000.129D
[java]
[java] RECEIVER :: Adding a message to incoming messages queue...
[java] RECEIVER :: Handling incoming messages...
[java] SECURE_LAYER :: Received message...
```

Reveive Q keys encoded with Merkle Key from Node A

```
[java]
[java] Length of QKEYS = 6
[java] KEYMANAGER: Number of Common Keys = 4
[java] Putting value in idKeyList
[java] SRCID :::::: 2 KEY VALUE :::::: 1744571290487985417
[java] KEY_MANAGER :: Key setup established...
```

Decode the keys and verify for minimum common keys. Save the key.

```
[java] ——————————————————
[java] SEND Q KEYS :: Value of key is :- 1744571290487985417
[java] ——————————————————
```

Encode and Send Q keys to Node A

```
[java] KEY_MANAGER :: Sending Key Setup Reply to 0014.4F01.0000.129D
[java]
[java] KEY_MANAGER :: Active region, neighbors found...
[java] SECURE_LAYER :: Boot up process accomplished, starting Sending thread...
[java] Application Initialized and ready...
```

Key setup successfully completed

5. Conclusion

Efficiently establishing secure keys is of critical importance for secure wireless sensor network applications. Local processing of sensor data requires secure node to node communication [4]. The schemes discussed above present efficient key distribution schemes for solving the security bootstrapping problem in resource-constrained wireless sensor networks.

The Lightweight Key Management in Wireless Networks by Leveraging Initial Trust scheme achieves security by exploiting the advantage of bounded period of trust, just after sensor nodes are deployed. Bootstrapping keys are used only in this trust period to establish a pairwise key and are erased after trust period has expired. Quick and inexpensive cryptographic primitives which require little memory are used in this scheme.

The q-composite random pre-distribution scheme achieves significant security under small scale attacks at the cost of greater vulnerability to large scale attacks. There is a tradeoff to be made between security and vulnerability that has to be considered based on the sensor network size and application.

6. References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “A survey on sensor networks,” *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–114, August 2002.

[2] A Key Management Scheme for Wireless Sensor Networks Using Deployment Knowledge

[3] Lightweight Key management in Wireless Sensor Networks by Leveraging Initial Trust

[4] Haowen Chang, Adrian Perrig and Dawn Song. *Random Key Pre-distribution Schemes for Sensor Networks.* Proceedings of the 2003 IEEE Symposium on Security and Privacy.

[5] 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.

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

[7] R. Merkle. Secure communication over insecure channels. *Communications of the ACM*, 21(4):294-299, 1978.

[8] Wireless Sensor Networks, A systems perspective By Nirupama Bulusu and Sanjay Jha.