

A Fault-tolerant Key-Distribution Scheme for Securing Wireless Ad-hoc Networks

Arno Wacker, Timo Heiber, Holger Cerman, and Pedro José Marrón

Institute for Parallel and Distributed Systems (IPVS)
Universität Stuttgart, Stuttgart, Germany

{wacker | heiber | cermanhr | marron}@informatik.uni-stuttgart.de

Abstract. We propose a novel solution for securing wireless ad-hoc networks. Our goal is to provide secure key exchange in the presence of device failures and denial-of-service attacks. The proposed solution relies solely on symmetric cryptography and therefore is applicable for highly resource-limited devices. In order to avoid a single point of trust, no master device or base station is used. We achieve this by enhancing our previously published approach with redundancy and algorithms for recovery on device failures.

1 Introduction

As the vision of Pervasive Computing becomes reality, many daily life devices will have computational power and wireless communication capabilities that allow the formation of ad-hoc networks. One scenario for such wireless ad-hoc networks is home automation. Here a private home is equipped with a multitude of sensors and actuators to enhance the lifestyle of individuals. For instance, the heating is turned on automatically when the owner of the house comes home; the light is switched on in rooms where motion is detected, etc. Security is a crucial factor for such systems as they introduce many new ways to invade an individual's personal life. For example, a thief could gather information about when somebody is at home before breaking into the house.

Encryption is an elementary technique for securing communications. Encryption schemes, however, require keys to be exchanged before secret communications can take place. Our goal is to provide a secure key exchange scheme for wireless ad-hoc networks. As we show below, this is a challenging task in such environments.

Our approach is suitable for resource-limited devices like sensors. Devices can exchange keys without referring to a central authority, thus avoiding a single point of trust. Unique keys are exchanged between device-pairs providing authenticity. Even when a device is subverted by an attacker, the key exchange for the remainder of the network remains functional. In [1] we have presented a key-distribution scheme that guarantees the secrecy of a key exchange as long as there are less than s subverted devices, where s can be chosen according to the actual security requirements. Device failures and denial-of-service attacks are not handled by this approach. In this paper, we extend our previous work to additionally cope with less than r device failures or denial-of-service attacks. This parameter can also be chosen according to the actual requirements.

The remainder of this paper is organized as follows: In Sec. 2 we describe the system model. Our previously published key-distribution scheme is introduced in Sec. 3. A first

extension to this scheme, based on redundancy, is presented in Sec. 4. Thereafter in Sec. 5, we incrementally design a scheme able to recover from multiple device failures even in the presence of attackers. Finally, we give an overview of existing approaches in wireless ad hoc and sensor networks in Sec. 6 and conclude the paper with a short summary and future work.

2 System Model

Our network consists of independent devices, each with its own processor and memory, communicating over a wireless channel. The channel itself is insecure, i.e. anyone can listen and send to the channel. We assume a non-partitioned network, that is, communication between two arbitrary devices is always possible¹. The number of devices is not predetermined or constrained in any way, as it may change due to the introduction of new devices to the network or device failures. The devices of such a network have to be inexpensive, and therefore they will only have limited resources. We assume that an attacker will subvert a number of devices since this is hard to prevent [2]. Considering these properties, we require that:

- The key distribution scheme must be decentralized — it still must be functional even when some devices are subverted.
- Symmetric cryptography is used in order to deal with resource limited devices — on such devices asymmetric cryptography is problematic [3].

We assume the existence of the following three types of attackers:

1. The *eavesdropping attacker (Eve)*. This attacker is only interested in learning about secrets of other devices, e.g. newly established keys. The objective of the attacker is to eavesdrop on communication between devices.
2. The *fail-stop denial-of-service attacker (fsDoS)*. This attacker silently halts all functions of a device. This class includes device failures (e.g. due to low power).
3. The *byzantine denial-of-service attacker (bDoS)*. This attacker may act arbitrary in order to prevent the correct execution of the key-distribution protocols. This class of attacker subsumes the fsDoS attacker class.

We have thoroughly analyzed Eve in [1]. In this work we analyze the other two types of attackers and provide countermeasures. Note that we do not consider DoS on the physical layer (e.g. jamming the frequency).

3 Basic Key-Distribution Scheme

The work presented in this paper is based on the scheme proposed and analyzed in [1]. We refer to this scheme as the basic key-distribution scheme (bKDS). In this section, we give a short overview of bKDS. Its overall objective is to establish a shared key between each pair of devices in the network. bKDS is robust against eavesdropping (Eve), but not against device failures (fsDoS) or denial-of-service (bDoS).

¹ See Sec. 4.3 for further discussion of this assumption

In order to set up the wireless ad-hoc network (and introduce new devices to it), bKDS requires an initial set of keys to be exchanged between devices via a secure channel. Following [4], this can e.g. be achieved by using physical contact between two devices to establish a unique shared key. However, it is impractical to establish physical contact between each pair of devices in the network, and the number of keys is limited by the memory of the devices. Hence, it is necessary to limit the number of physically exchanged keys and establish additional shared keys between devices on demand.

In the next subsection we define the formal representation of our network, which will be used throughout the rest of this paper. Subsections 3.2 through 3.4 describe the original algorithms while their properties and restrictions are discussed in 3.5.

3.1 Network Model

We represent our network as an undirected graph $G = (V, E)$, where V is the set of devices in the network, and E represents the set of shared keys between devices where $\{v_1, v_2\} \in E$ iff the nodes v_1 and v_2 share a symmetric key. We will use the term *device* to indicate the physical device and the term *node* to indicate the representation of that device in the graph.

3.2 Network Setup

Whenever a new device is added to the network, a certain number of keys needs to be exchanged. We require that each device that is added to the network shares a securely exchanged key with at least s devices that are already part of the network and refer to s as the *security level* of the network. The actual procedure is given in Alg. 1, using the formal representation of the network.

A graph constructed with Alg. 1 with less than $(s+1)$ nodes will be fully connected. For each additional node introduced, s new edges from the new node to previously existing nodes will be added. This algorithm guarantees the existence of s node-disjoint paths between two arbitrary nodes in the graph. Proof of this and further properties of the algorithm can be found in [1]. The security of the scheme presented in the next subsection is dependent on this property.

3.3 Establishing a New Shared Key

Two devices that do not share a key yet can establish a new one using the following approach: To establish an l -bit key k , a device randomly generates s l -bit shares k_1, \dots, k_s , and sends them over s device-disjoint paths (i.e. paths that do not share common devices) to the destination device (Fig. 1(a)). On each link of a path, the key share is encrypted with the existing shared key for this link. The final key k is then calculated by $k = k_1 \oplus k_2 \oplus \dots \oplus k_s$, where \oplus is the bitwise XOR operation. This approach is also used in [5–7].

It should be clear that without having access to all key shares, an Eve-type attacker does not stand a chance to recover the key. If it can be assured that the key shares are communicated over s node-disjoint paths of the network graph, Eve will need to subvert at least s nodes (one on each path) to compromise the newly established key. Our construction per Alg. 1 guarantees this property.

Algorithm 1 Introducing a new node to the graph

```
1: Given a graph  $G = (V, E)$  with  $n = |V|$  with nodes  $v_i \in V$  and a new node  $v_{n+1}$ 
2:  $V := V \cup \{v_{n+1}\}$ ;
3: if  $s \geq n$  then
4:   // device corresponding to  $v_{n+1}$  creates  $n$  new keys
5:   for  $i = 1$  to  $n$  do
6:      $E = E \cup \{v_{n+1}, v_i\}$ ; // establish a new key through physical contact
7:   end for
8: else
9:    $V' :=$  random subset of  $V - \{v_{n+1}\}$  with  $|V'| = s$ ;
10:  // device corresponding to  $v_{n+1}$  creates  $s$  new keys
11:  for  $i = 1$  to  $s$  do
12:     $E = E \cup \{v_{n+1}, v_i\}$  with  $v_i \in V'$ ; // establish a new key through physical contact
13:  end for
14: end if
```

3.4 Controlled Removal of Devices

Removing a device from the network can destroy the s -connected property of the corresponding graph. In [1], we also proposed an algorithm for removing a device in a controlled shutdown, i.e. a device has the time to announce its impending departure from the network and make all necessary arrangements. Algorithms 2 and 3 describe this procedure for removing a device from the network.

The presented solution is based on pretending that the device that is to be removed had never been there in the first place and replacing existing shared keys accordingly. If the device is still present when this procedure is performed, keys can be replaced automatically, using the procedure described in Sec. 3.3, thus re-establishing the s -connected property of the underlying graph.

To see why these algorithms work, let the graph under consideration be $G = (V, E)$ with $V = \{v_1, v_2, \dots, v_n\}$ where a node v_i was the i -th node added to the graph according to our construction. Let the node that is to be removed from the graph be v_j .

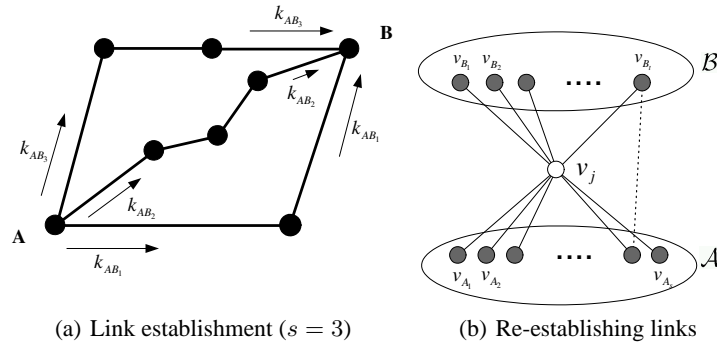


Fig. 1.

Algorithm 2 Controlled removal from the network

```
1:  $FirstDevices$  := first  $s$  known devices (set  $\mathcal{A}$ );
2:  $MyDeviceList$  := the set of devices for which we have a shared key;
3: for all  $Device \in MyDeviceList$  do
4:    $sendto(Device, LeavingIntention\{FirstDevices\})$ ;
5: end for
6: wait for all ACKs;
```

Algorithm 3 Action of a neighboring device

```
1:  $onReceiveLeavingIntention(Sender, DeviceList)$ ;
2:  $MyDeviceList$  := the set of devices for which we have a shared key;
3:  $Candidates := DeviceList - MyDeviceList$ ;
4: if  $Candidates \neq \emptyset$  then
5:    $Device :=$  pick randomly one from  $Candidates$ ;
6:   establish a new shared key with  $Device$ ;
7:   update  $MyDeviceList$  by replacing  $Sender$  with  $Device$ ;
8: else
9:    $MyDeviceList := MyDeviceList - \{Sender\}$ ;
10: end if
11:  $sendto(Sender, "ACK")$ ;
```

Then we can define two sets, \mathcal{A} and \mathcal{B} (see Fig. 1(b)). The set \mathcal{A} contains all s nodes to which v_j had its first s edges during the construction, i.e. the devices, v_j established the first s keys ($FirstDevices$ in Alg. 2). The set \mathcal{B} contains all nodes which established an edge to v_j during their insertion into the graph, i.e. the devices being inserted *after* v_j . When v_j leaves the network, all nodes in set \mathcal{B} need to establish a new edge to some node in set \mathcal{A} which did not exist before. Doing this will result in the same situation as if the nodes in set \mathcal{B} were introduced into the graph when node v_j was not a node of the graph. Thus, the s -connected property of the graph is again guaranteed by the construction algorithm.

3.5 Properties of the Basic Key-Distribution Scheme

bKDS guarantees the *secrecy* of a newly established key as long as Eve controls less than s devices. However, the s -connected property and therefore new key establishment can only be guaranteed in the absence of DoS attacker devices:

A device subverted by a fsDoS attacker will not execute any algorithms anymore.

Thus it will not perform the procedure for controlled removal, which might destroy the s -connected property of the underlying graph.

A device subverted by a bDoS attacker will interact with the protocols with the deliberate intention to destroy the s -connected property, i.e. preventing the establishment of new keys. Imagine the following situations:

1. *During key establishment*: The device might forward an altered key-share (see Sec. 3.3). As a result, the two participating devices will not share a common key. Thus, establishing a new key cannot be guaranteed anymore.

2. *Initiating the procedure for controlled removal:* The device might initiate the removal protocol, but send a false list of *FirstDevices* to its neighbors. As a result, the neighbors would make connections to the wrong devices, possibly destroying the s -connected property.
3. *Responding to the procedure for controlled removal:* The device might decide not to respond in a correct manner when the device gets a “LeavingIntention” message. This also might result in the destruction of the s -connected property of the underlying graph.

In order to become robust against fsDoS or even bDoS, these issues have to be addressed. Therefore in the remainder of this paper, we enhance bKDS with redundancy and recovery mechanisms to cope with these attacker types.

4 Basic Redundancy for Coping with DoS Devices

A direct solution for dealing with DoS is the introduction of redundancy: In order to deal with r bDoS attacker devices we need to establish a $(s + r)$ -connected graph. We refer to r as the *redundancy level* of our network. Thus, between any pair of nodes, there exist $(s + r)$ node-disjoint paths, of which s paths are needed in order to establish a new key.

4.1 Properties of the Basic Redundancy Approach

Having a $(s + r)$ -connected graph, provides room to tolerate up to r bDoS devices. Consider the following situations where bDoS attacker devices can interfere:

Attacks on the key establishment protocol. In order to establish a new key, a device chooses s paths (out of a total of $(s + r)$) to the destination and invokes the key establishment protocol as described in Sec. 3.3. After the key establishment has succeeded, both parties have to check whether they really share the same key. This can be done with a challenge/response protocol using the newly established key. In case the key-establishment fails (or the key is invalid), the device chooses another set of s paths to the destination device, until it eventually establishes a functional key.

Attacks with respect to Alg. 2 When a device removes itself, it is expected to execute Alg. 2. Three cases are possible:

1. The leaving device has failed (e.g. low power), or it has been subverted by a DoS attacker, i.e. it is part of the fsDoS class. In this case it will not initiate the removal protocol. Thus the network will not be repaired, i.e. the $(s + r)$ -connected property will not be restored, leaving the graph only $(s + r - 1)$ -connected in the worst case. For an analysis see Sec. 4.2.
2. The device is subverted by a bDoS attacker and therefore tries to prevent new key establishments in the network. It does so by providing a false list of devices (*FirstDevices*) when sending its “LeavingIntention” (see Alg. 2). This leads to a false reconstruction of the graph, which also in worst case decreases the connectivity of the graph by one.

3. The leaving device is supposed to wait until all neighboring devices have acknowledged (see Alg. 2) before it can eventually leave the network. Due to failure of a neighboring device, however, this message may never be received. In order not to wait forever, we must introduce a timeout mechanism. For the underlying graph, this means that one of the neighbors did not execute the removal protocol properly, therefore the original connectivity of the graph cannot be guaranteed anymore. However, also this situation would as a result decrease the connectivity of the graph by one.

Attacks with respect to Alg. 3 When a device receives a “LeavingIntention” from a neighboring device, it must act in order to repair the $(s + r)$ -connected property of the graph. It must (if necessary) establish new keys in order to bypass the leaving device and when done, send an acknowledgement to the leaving device. Two situations are possible:

1. The device does not send the necessary acknowledgement at the end of Alg. 3, since it is subverted by an fsDoS or even bDoS attacker. When subverted by a bDoS attacker it may establish its new keys to arbitrarily chosen devices. This situation is already described above leading to the decrease of the connectivity by one.
2. The device is subverted by a bDoS attacker, and therefore does not repair the network as it should (i.e. establish keys to a device from the *FirstDevices* list), but sends the acknowledgement message anyway. Also here, in the worst case, the connectivity of the graph is reduced by one.

Therefore we can conclude that given an $(s + r)$ -connected network, the network will stay at least s -connected as long as the attacker (fsDoS or bDoS) does not subvert more than r devices.

4.2 Analysis

From a formal point of view, all attacks described in the previous section can be divided into two categories:

1. The removal of a node, without “repairing” this section of the graph, i.e. constructing new edges in order to bypass the removed node.
2. The removal of a node while constructing additional edges to some arbitrary nodes, which do not bypass the removed node.

Clearly, the additional construction of edges in a k -connected graph cannot decrease the connectivity. In the worst case, however, the removal of nodes can decrease the connectivity by the number of removed nodes. In order to show this we need the following definition from graph theory:

Definition 1 (k-connected graph). A graph $G = (V, E)$ is said to be k -connected iff for any set $W \subseteq V$ with $|W| < k$, the subgraph induced by $V - W$ is still connected.

Theorem 1. In a k -connected graph $G = (V, E)$, the removal of t nodes (and all corresponding edges), will result in a graph, which will be at least $(k - t)$ -connected.

Proof. By contradiction. Assume that the graph G' induced by $V - Y$ with $Y \subseteq V$ and $|Y| = t$ is not at least $(k - t)$ -connected. Then it suffices to remove h nodes from G' , with $h < (k - t)$ in order to disconnect G' . But this means we disconnected G by removing $t + h$ nodes with $t + h < k$, which is a contradiction to G being k -connected since the removal of less than k nodes cannot disconnect G .

In this section, we have seen that an $(s + r)$ -connected graph can tolerate up to r bDoS devices while retaining a security level of s . However, with the Basic Redundancy approach the connectivity of the graph will monotonously decrease with every additional devices being in fsDoS or bDoS. In Sec. 5 we will present an approach that allows the graph to recover from device failures, i.e. devices from the fsDoS class.

4.3 Coping with Denial of Service at the Routing Layer

Up to now, we have assumed that devices can always communicate with each other. However, dealing with denial of service attacks also requires dealing with attacks on the communication system.

As mentioned earlier, countermeasures against attacks on the physical layer are beyond the scope of this paper. What remains to be considered, though, are routing attacks, i.e. attacks on layer 3 of the ISO/OSI model.

If all devices are in transmission range of each other, our protocols can be built directly on layer 2 of the ISO/OSI model. There is no need for routing, so the problem is solved trivially.

If routing is used for communication, we need to consider the *communication graph* of the network. The communication graph is a graph whose nodes represent the devices and whose edges describe a direct communication link between two devices. In order to fully prevent denial-of-service with up to r devices exhibiting bDoS behavior, we need to build an $(r + 1)$ -connected communication graph. Similar to Alg. 1, it is sufficient if every device is in the communication range of at least $(r + 1)$ other devices when adding it to the network. With an $(r + 1)$ -connected graph and an appropriate routing protocol (e.g. flooding), a message will always arrive at the destination.

For the rest of this paper we assume that we either have a $(r + 1)$ -connected communication graph or that the devices are in range of each other — thus, as stated in Sec. 2, communication between two arbitrary devices is always guaranteed.

5 Recovery from Device Failure

While the number of attacker devices is a parameter which depends on the security aspects of the networks, the number of probable failures (fsDoS) is not as easy to pre-determine. Therefore we propose an enhancement which enables the network to recover from fsDoS attacks, thus making the initially needed connectivity independent of the number of probable device failures.

We introduce a new parameter c which holds the maximum number of device failures (fsDoS) occurring *concurrently* while the recovery algorithm can still recover the connectivity of the underlying graph.

Algorithm 4 Detecting a failed device

```
1: onDeviceFailureDetect(FailedDevice);
2: if FailedDevice  $\in$  FirstDevices then
3:   // we are in set  $\mathcal{B}$ , therefore we must act
4:   broadcast(FailingNotification(FailedDevice));
5: end if
```

Algorithm 5 Checking upon a *FailingNotification*

```
1: onReceiveFailingNotification(Sender, FailedDevice);
2: if LinkPartnerKeyPosition(FailedDevice)  $\leq$  ( $s + r$ ) and FailedDevice not responding then
3:   // we are in FirstDevices (set  $\mathcal{A}$ ) of FailedDevice and the device really failed
4:   broadcast(HealingNotification(FailedDevice));
5: end if
```

We are going to build our final algorithms incrementally. In Sec. 5.1, we show the basic principle for dealing with device failures. We then modify this approach in Sec. 5.2 to rule out some DoS attacks. Our final algorithm, which handles DoS and also multiple failures at the same time, is presented in 5.3.

5.1 Basic Approach for Recovery from Device Failures

When a device fails, it is not able to perform the controlled removal protocol (as described in Sec. 3.4). Even worse, in the basic removal approach, the departing device still has to be in place while executing the protocol, since the underlying graph does not provide any redundancy (it is exactly s -connected).

We will now introduce the *passive* removal protocol. This protocol is based on the controlled protocol, but in contrast it is initiated by a neighboring device. Note that by having an underlying $(s + r)$ -connected graph, any removal protocol can be executed without the aid of the device which is leaving or has just failed. The basic idea behind the passive removal protocol is, that if some device in the network detects that another device has failed, it tries to recover this part of the underlying graph. In order to do so, it first has to gather the information about which nodes are in the set \mathcal{A} (see Sec. 3.4) of the failed device. It does so by broadcasting a request (Alg. 4) in order to get an answer from any device being in the *FirstDevices* set of the *FailedDevice* (Alg. 5). After receiving a “HealingNotification” from any device, which it does *not* share a key with, it can create a new key to this device to substitute for the failed one. Since all messages are sent using broadcasts, other devices — having the failed device in their *FirstDevices* set — can also make use of the sent “HealingNotification”, even if they did not send a “FailingNotification” themselves. Algorithms 4, 5 and 6 describe our approach in detail.

There is still an open question in the above algorithm: How does a device know if it belongs to the set *FirstDevices* of another device? Clearly, asking the device in question is not possible, since it failed and therefore will not be able to respond to any requests. This information can be saved together with a new key, when integrating a device into the network. Keys are stored in the order in which they were established, making it easy to determine the set *FirstDevices* for a certain device. Additionally, in order to

Algorithm 6 Action of a healing device

```
1: onReceiveHealingNotification(Sender, FailedDevice);
2: if FailedDevice not responding and Sender  $\in$  FirstDevices then
3:   establish a new shared key with Sender;
4:   update MyDeviceList by replacing Sender with FailedDevice;
5: end if
```

determine whether a device is among the set *FirstDevices* of another device, for each key we store the position of this key at the other device. For device x this information is *LinkPartnerKeyPosition*(x). When this position changes, also the remote stored information has to be updated. Note that this still scales with respect to the network size — it just increases the needed storage for a key on the device by a constant amount. The basic key distribution scales with on average $2s$ keys per device.

To see why these algorithms work, consider the following: In 3.4 we stated that on failure of a node v_j any node $v_{B_i} \in \mathcal{B}$ has to replace its missing edge to v_j with an edge to a node $v_{A_i} \in \mathcal{A}$ (see Fig. 1(b)). In the above algorithms, a node in the set \mathcal{B} will detect the failure of the corresponding device and will therefore send a “FailingNotification”. All nodes $v_{A_i} \in \mathcal{A}$ will receive and answer to this request with a “HealingNotification”, thereby giving all nodes $v_{B_i} \in \mathcal{B}$ the information needed to replace the missing edge.

As the above algorithms are based on the controlled removal protocol, they provide the same properties, i.e. after a device failure it is ensured that the underlying graph has still the same connectivity as before. However, this will not work under all circumstances:

If a bDoS device is present the following can happen: In case of device failure, some other device will detect this failure and per Alg. 4 will ask for the set of *FirstDevices* (set \mathcal{A}) of the failed device. The attacker, who wants to prevent the network from establishing new keys, will announce himself as part of this set. In this case, other devices will establish links to the attacker in order to repair the network, possibly decreasing the connectivity of the underlying graph by one. Therefore doing this $r + 1$ times (on $r + 1$ device failures), the connectivity of the underlying graph may drop below s , although there was just a single bDoS attacker device present.

Multiple failures at the same time also pose a problem: Imagine two devices failing, where one device is in the *FirstDevices* set of the other. Not all required devices (those of set \mathcal{A}) will respond to a “FailingNotification” and therefore it cannot be guaranteed that every node from set \mathcal{B} will find a suitable node from set \mathcal{A} to replace the “dead” key. Note that a fundamental property for the correctness of the protocol is the fact that the size of set \mathcal{A} is equal to the connectivity of G — see [1] for details.

There are several more possibilities for a single DoS attacker device. All these attacks are possible due to the fact that the complete set *FirstDevices* is only known to one device and are lost if this device fails.

5.2 Recovery with Replicated Data in Set \mathcal{B}

As pointed out in the last section, the major weakness in algorithms 4-6 is that the *FirstDevices* set of the failed device is not known to other devices. Therefore, an attacker

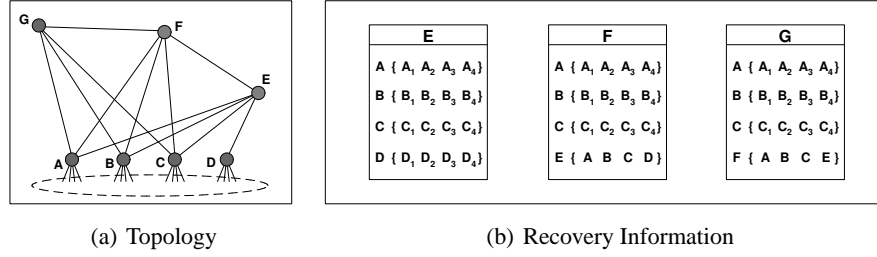


Fig. 2. Recovery Information

can forge this information. To overcome this, we need to replicate this information to all devices that may need it. By Sec. 3.4 these are the nodes which are in the set \mathcal{B} of a node. As an example consider Fig. 2. The *FirstDevices* set of device A is $\{A_1, A_2, A_3, A_4\}$ (analogously for B, C, D) — in Fig. 2(a) the area below A, B, C and D . Fig. 2(b) shows the information stored on nodes E, F and G . For instance, node E is in the set \mathcal{B} of device A, B, C and D , therefore it stores the *FirstDevices* sets for these nodes (first column). The *FirstDevices* set of node E is $\{A, B, C, D\}$. This information is replicated only on node F since F is in the set \mathcal{B} of node E (column 2, line 4).

The replication of this list is done during the initialization phase of a device. Whenever a new device is introduced into the network, a number of keys are shared with other devices. Immediately after these keys are shared, the device knows its own *FirstDevices* set, and can replicate this information to every *new* device in *MyDeviceList* — this corresponds to the set \mathcal{B} of the underlying graph.

Having stored this information, every device can locally execute the function `getFirstDeviceList(Device)`, which delivers the corresponding list for each device in its own *FirstDevices* set. That way, failures can be repaired based only on local information.

Note that by executing Alg. 7 on every device, there is no need for additional communication anymore — Alg. 7 replaces the original algorithms for passive removal. The controlled removal algorithm can still be used as an optimization but due to the additional data, an attacker device cannot cheat about any list sent.

Properties of the Approach with Replication in set \mathcal{B} Our modified approach uses additional memory on each device. For a $z := s + r$ -connected graph, we need on average

Algorithm 7 Action on a device failure on every node

- 1: `onDeviceFailureDetect(FailedDevice)`;
 - 2: `FailedDeviceFirstDevices := getFirstDeviceList(FailedDevice)`
 - 3: `Candidates := FailedDeviceFirstDevices – MyDeviceList`;
 - 4: **if** `Candidates` $\neq \emptyset$ **then**
 - 5: `Device := pick randomly one from Candidates`;
 - 6: establish a new shared key with `Device`;
 - 7: update `MyDeviceList` by **replacing** `FailedDevice` with `Device`;
 - 8: **end if**
-

$2z \cdot s_k$ bytes memory storage for the keys on each device, where s_k is the size of a key in bytes. On top of that, the following amount of storage is needed for the *FirstDevices* sets: Each set contains z device ids. If s_i is the size of a device id in bytes, we need $z \cdot s_i$ bytes for a single set, of which z sets are stored per device. Thus the additional memory requirements for storing the replicated lists are $z^2 \cdot s_i$ bytes per device. The total required memory per device is $2z \cdot s_k + z^2 \cdot s_i$ bytes, which is not dependent on the network size n . This means that it scales with respect to the network size n , but the memory overhead with respect to the security parameters is $\mathcal{O}((s+r)^2)$.

Alg. 7 prevents some of the DoS attacks described in Sec. 5.1. The reasons why this algorithm can recover from non-simultaneous failures remain the same as in Sec. 5.1. The problem of multiple device failures, however, is still unresolved: Alg. 7 cannot recover when multiple devices which share a key fail simultaneously. The reasons are the same as in Sec. 5.1. Therefore, although this approach prevents some DoS attacks, it is still vulnerable and does not tolerate arbitrary device failures.

5.3 Recovery with Replicated Data in Set \mathcal{A}

In the last section we have proposed to replicate the *FirstDevices* set on all devices which are in the set \mathcal{B} of a particular node. We showed that this does not prevent all weaknesses described in 5.1. The main reason is that the size of set \mathcal{B} is not predetermined in any way. Thus we instead propose to replicate the *FirstDevices* set of a device on all devices which are part of the *FirstDevices* set, i.e. in set \mathcal{A} . This set always contains z elements when the underlying graph is z -connected.

In this scheme we set $r := d + c + m$ with

$$m := \begin{cases} d + 1 - s & : s < d + 1 \\ 0 & : otherwise \end{cases}, \quad (1)$$

where c stands for the maximum number of *concurrent* device failures and d for the maximum tolerated number of bDoS attacker devices. Thus the graph G will be z -connected with

$$z := s + r = \begin{cases} s + d + c & : s < d + 1 \\ 2d + c + 1 & : otherwise \end{cases}. \quad (2)$$

Since $s + d + c \geq 2d + c + 1$, G will be at least $(2d + c + 1)$ -connected. The basic idea behind this approach is that having the *FirstDevices* set replicated $(2d + c + 1)$ -times, even when there are d subverted devices, acting as bDoS attacker devices and additionally c concurrent failures in the network, there are still a majority of $(d + 1)$ devices with the replicated information available, which will cooperate and therefore enable the algorithms to proceed.

Thus, when building an z -connected graph as above, we tolerate s eavesdropping devices, d bDoS attacker devices and an arbitrary number of failures, of which at most c failures can occur concurrently.

A device failure will eventually be detected by a neighboring device. This device will request the *FirstDevices* set with a broadcast. A device which holds a replica of this set will first establish a new key with the requesting device (if necessary) and then

Algorithm 8 Detecting a failed device

```
1: onDeviceFailureDetect(FailedDevice);
2: if FailedDevice ∈ FirstDevices then
3:   // we are in set  $\mathcal{B}$  of the failed device
4:   FirstFailedDevice := FailedDevice; // remember the device we started the recovery with
5:   execute recoverDeviceFailure(FailedDevice);
6: end if
```

Algorithm 9 Broadcasting the Recovery Request

```
1: function recoverDeviceFailure(FailedDevice);
2:   DeviceToRecover := FailedDevice;
3:   VoterSet :=  $\emptyset$ ; // empty set
4:   FirstDevicesLists := []; // empty list
5:   broadcast(RecoveryRequest(FailedDevice));
```

send the set to the requesting device over this secure channel. Note that with at most d bDoS attacker devices (and c concurrent failures), in the worst case the requesting device will still receive $(2d + c + 1) - c - d = d + 1$ sets from non-subverted devices. When $(d + 1)$ times the same *FirstDevices* set has been submitted by different senders, the receiving device can be sure that this is the correct set and use it for determining the device to which it must create a new key in order to recover from the failure. If this device is not available either (i.e. it has also failed or it is a DoS attacker device), the scheme is applied recursively. Algorithms 8 through 11 describe this behavior in detail.

When a device detects the failure of another device belonging to its *FirstDevices* set, it saves the corresponding device id in the variable *FirstFailedDevice* (see Alg. 8). Then the recovery is started by executing the function `recoverDeviceFailure()` (Alg. 9). This function will be called recursively in case of multiple device failures.

Consider Fig. 3. The parameters used in this example are $s = 2$, $d = 1$ and $c = 1$, which results in $r = 2$.

In Fig. 3 devices F and E have just failed simultaneously. Note that since bDoS behaviour subsumes fsDoS behaviour, it is possible to tolerate these failures. The failure of F is eventually detected by G , which needs to replace the key to F with a key to some device in the *FirstDevices* set of F . In order to gather this information, it broadcasts the request for *FirstDevices* of F (Fig. 3(a)).

A device receiving “RecoveryRequest(*FailedDevice*)” must first check if it belongs to the *FirstDevices* set of the failed device. This is done using the function `LinkPartnerKeyPosition` introduced in Sec. 5.1. Note that the information needed for `LinkPartnerKeyPosition` can be gathered from the locally replicated *FirstDevices* sets — there is no need to store additional key positions as proposed in Sec. 5.1.

In the next step, every device in the *FirstDevices* set of the failed device will send the local copy of this set to the requesting device. This is done over a secure connection, establishing a new key with the requesting device if necessary. Fig. 3(b) shows devices A , B and C sending this information to G — device E should also send this information, but since it failed it will not do so. Device G collects the responses, i.e. 3 times

Algorithm 10 Actions upon a receiving a RecoveryRequest

```
1: onReceiveRecoveryRequest(Sender, FailedDevice);
2: if LinkPartnerKeyPosition(FailedDevice)  $\leq (s + r)$  then
3:   // we are in set  $\mathcal{A}$  of FailedDevice, therefore we hold a replica of the FirstDevices-set
4:   if Sender  $\notin$  MyDeviceList then
5:     // we don't share a key with the requesting device, thus we have to establish one
6:     establish a new shared key with Sender;
7:     // send the replicated FirstDevices of FailedDevice to the sender
8:     sendto(Sender, RecoveryAnswer(FailedDevice, FirstDevices(FailedDevice)));
9:     release shared key with Sender;
10:  else
11:    // send the replicated FirstDevices of FailedDevice to the sender
12:    sendto(Sender, RecoveryAnswer(FailedDevice, FirstDevices(FailedDevice)));
13:  end if
14: end if
```

the *FirstDevices* set of device F : $\{A, B, C, E\}$. A response is counted only once per responding device.

Device G , having received the same set at least twice ($d + 1 = 2$), tries to recover from the failure of device F by establishing a new key to device E (Fig. 3(c)). However, E has also failed, thus the key establishment will fail. Device G will therefore recursively try to recover from the failure of E by broadcasting the request for *FirstDevices* of E (Fig. 3(d)). In this case devices A, B, C and D will answer, providing G with the needed information (Fig. 3(e)). After the second identical *FirstDevices* set ($d + 1 = 2$), G can be confident that it has the correct set, namely $\{A, B, C, D\}$, and will choose D — the only one with which G does not share a key yet. In order to recover the connectivity of the underlying graph, device G will now replace the key for device F with the newly established one for device D (Fig. 3(f)).

Properties of the Approach Just as in the previous approach, additional memory is required. The additional memory requirements are the same as in Sec. 5.2, since we also replicate the *FirstDevices* set z times, with $z := s + r$. However, during the execution of the recovery, additional temporary memory is needed to store the received sets. A maximum of z additional sets will be received, thus the temporary overhead is $z \cdot z \cdot s_i = z^2 s_i$. This approach again is independent of the network size n , but has quadratic overhead with respect to the security parameters s and r .

This final approach, however, can cope with up to c concurrent device failures and uncooperative devices, i.e. DoS attackers. Building the graph with a redundancy of $r = c + d$ gives room to tolerate d attacker devices and c concurrent failures. This is due to the fact that even in the worst case, with d DoS attackers and c failing devices, there will be $d + 1$ devices left which will cooperate. After a successful execution of the recovery algorithms, the connectivity of the graph will be restored, i.e. the graph will again be $(s + r)$ -connected. To see why this approach works, consider the possibilities a DoS attacker device has:

Algorithm 11 Actions due to requested answers

```
1: onReceiveRecoveryAnswer(Sender, FailedDevice, FailedDeviceFirstDevices);
2: if FailedDevice = DeviceToRecover and Sender  $\notin$  VoterSet then
3:   // first answer from this sender
4:   VoterSet := VoterSet  $\cup$  Sender; // just one answer per sender
5:   FirstDevicesLists := FirstDevicesLists + (FailedDeviceFirstDevices);
6:   Candidates := select a FirstDevicesLists-entry, if included at least  $(d + 1)$  times, else  $\emptyset$ ;
7:   Candidates := Candidates - FirstDevices;
8:   if Candidates  $\neq$   $\emptyset$  then
9:     Device := pick randomly one from Candidates;
10:    if establish a new shared key with Device then
11:      update MyDeviceList by replacing FirstFailedDevice with Device;
12:      DeviceToRecover := none;
13:    else
14:      execute recoverDeviceFailure(Device);
15:    end if
16:  end if
17: end if
```

Denying to forward key-shares or send altered key-shares This cannot harm, since there are always redundant paths available.

Refusing to establish a new key with some device The attacker device will be considered as a failed device, and therefore removed from the network. Due to the recovery mechanism, the connectivity will be restored without this device. Thus only the attacker device itself is affected.

Incorrectly initiating a recovery process Three cases are possible:

1. The attacker device initiates the recovery process for a non-existing device. On receiving the broadcast every device will locally determine that it is not part of the corresponding *FirstDevices* set. Therefore no further action will be taken.
2. The attacker device initiates the recovery process for an existing device which is not part of its *FirstDevices* set. With this action the attacker gains information about the network topology, i.e. the *FirstDevices* set of the announced *FailedDevice* and it causes some unnecessary communication. The connectivity of the underlying graph will not be decreased by such messages, since every device has to initiate the recovery process on its own.
3. The attacker device initiates the recovery process, but forges its own id — this is possible since group communication (i.e. broadcast) is not authenticated. This attack yields the same communication overhead as described before. Devices receiving answers (the real devices to which the forged id belongs) will discard these messages, as long as they did not broadcast the same request.

Incorrectly responding to a recovery request There are two cases possible: The responding device might not answer at all, or it replies with a bogus *FirstDevices* set. Both cases will not do harm since there will be enough correct answers (at least $d + 1$).

Incorrect processing of recovery messages Upon a failure, an attacker device might decide to announce this failure correctly, thus gathering the needed information

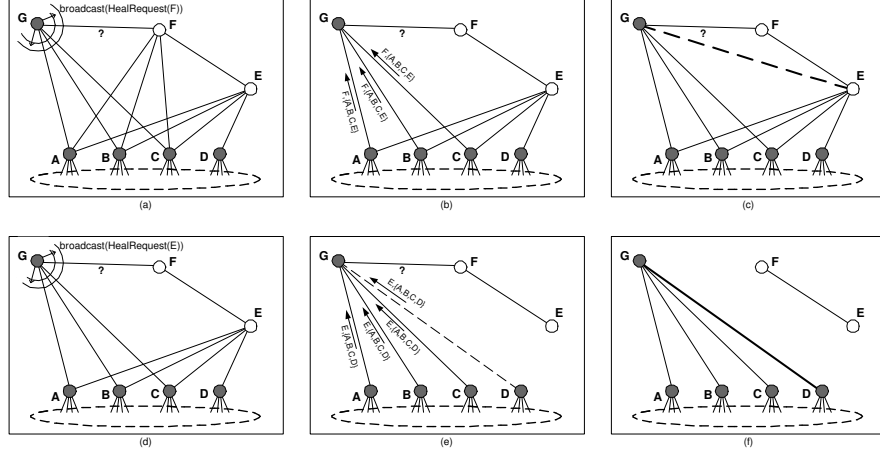


Fig. 3. Recovery from multiple device failures, ($s = 2, r = 2$)

about the *FirstDevices* set, but behave arbitrarily afterwards. This decreases the connectivity of the graph by at most one. However, this happens per attacker device and not per failed device and is therefore covered by the introduced redundancy.

The extensions to our algorithms guarantee the secrecy of the newly established keys, as long as there are less than s eavesdropping attackers in the network, and also guarantee the key establishment functionality of the network as long as there are at most r DoS attacker devices present. Devices belonging to the fsDoS class are fully covered in these algorithms, i.e. the number of fsDoS devices is only limited by the network size n and such failures can always be recovered from as long as there are less than c simultaneous failures. In general, in order to deal with g attackers of *any kind* while recovering up to c simultaneous failures, we have $s := g + 1$ and $d := g$, hence we need to initially establish a z -connected graph with $z := 2d + c + 1 = 2g + c + 1$ (since $s \not\leq d + 1$, see (2) in Sec. 5.3).

6 Related Work

A number of solutions for key-distribution in wireless ad hoc or sensor networks have been proposed. Most of these approaches do not address the issue of easy addition or removal of devices.

The straightforward solution is to use certificates issued by a central certification authority. A central server thereby stores and signs the public keys of the individual devices. The major problems with this approach are the limited resources of the devices and the need for the central (potentially vulnerable) authority itself.

Since a centralized authoritative device is a single point of failure, asymmetric decentralized approaches have been proposed. Such designs can be achieved by distributing the certification authority over the participating devices [8, 6].

Asymmetric approaches share a common problem: If small sensor devices are used — even with the use of supporting high performance nodes [9] — asymmetric cryptography is often not possible due to delay and energy constraints [10, 3].

Perrig et al. [11] proposed SPINS, a centralized approach using symmetric cryptography. They address device failures due to energy loss or destruction. If a device fails, the base station will remove all references on other devices. The establishment of new session keys is done solely by the base station. Therefore, device failure cannot affect this functionality or communication. However, the failure of the base station will disable the whole key-distribution. Also, the base station is a single point of trust, thus the hostile take over of the base station will subvert the whole network at once.

To avoid these problems, a few approaches using symmetric cryptography without the use of a base station have been proposed. Secure Pebblenets [12] represent a simple approach proposed by Basagni et al. using only one key — known by all devices — for the whole network. A link creation is not required since all messages are encrypted with the same key, independent of the receiver. As a drawback, no device to device authenticity can be achieved, and additionally, the tampering of one single device reveals the data sent by all devices to the attacker.

Eschenauer and Gligor [13] proposed a solution using a random key predistribution. Before deployment, every device is supplied with a probabilistic set of keys from a key pool. After deployment, the devices try to establish connections by finding a commonly shared key or by creating a new key through a secure path including other devices. Since their approach is probabilistic, no clear assumptions about network connectivity can be made afterwards. After the initial network establishment, every device creates a secure link to all of his neighbors in communication range, so the connectivity of the network will not be threatened by single device failures. The removal of subverted devices is handled by a centralized revocation scheme using a trusted base station.

Extensions have been introduced by Chan et al. [5]. They present three different approaches. In one of their approaches, a set of secure and authenticated links can be established after deployment of the sensors. Due to the random predistribution, two arbitrary devices might not be able to establish a secure link without relying on other devices, since only some *randomly chosen* nodes can communicate directly and authenticated with each other. Due to this fact, real authenticated communication between arbitrary devices is not always possible. A device failure decreases the probability that two devices may communicate.

The fourth decentralized symmetric approach by Zhu et al. [14] also uses a initially distributed set of random keys. Additionally to [5], Zhu et al. propose a pairwise key establishment protocol using *multiple* paths. This way, splitting of a pairwise key over multiple untrusted paths, as proposed in [7], can be used to improve attacker resistance. Due to the random key predistribution, the actual existence of different paths in the network is not assured in any way. It follows that — in case of device failures — no presumption about the real number of the device disjunct paths is possible.

Previous work on fault-tolerance has been done for instance in [15]. The focus of this work is the interplay of network connectivity and secure communication in a general way. The issue of easy addition or removal of devices is not within the scope of these approaches.

7 Conclusion and Future Work

In this paper, we presented a fault-tolerant approach for key distribution in wireless ad-hoc networks based on symmetric cryptography. In contrast to the related work, we have proposed a parametrized algorithm which *guarantees* the ability for an arbitrary pair of devices to exchange a key in a secure fashion, provided that the number of subverted devices remains below certain threshold parameters. We proposed a recovery algorithm which ensures the key-exchange functionality in case of device failures.

We have implemented our approach on the Atmel ATMega16 microcontroller. This implementation will be used to conduct a performance evaluation of our approach.

Furthermore, we are working on new mechanisms to discover the device-disjoint paths. Additional work includes key revocation schemes combined with intrusion detection mechanisms.

References

1. Wacker, A., Heiber, T., Cermann, H.: A key-distribution scheme for wireless home automation networks. In: Proceedings of IEEE CCNC 2004, Las Vegas, Nevada, USA, IEEE Communications Society, IEEE (2004)
2. Anderson, R., Kuhn, M.: Tamper resistance - a cautionary note. In: Proceedings of the Second Usenix Workshop on Electronic Commerce. (1996) 1–11
3. Brown, M., Cheung, D.: PGP in constrained wireless devices. In: Proceedings of the 9th USENIX Security Symposium. (2000)
4. Stajano, F., Anderson, R.: The resurrecting duckling: Security issues for ad-hoc wireless networks. In: 7th International Workshop on Security Protocols. LNCS (1999) 172–194
5. Chan, H., Perrig, A., Song, D.: Random key predistribution schemes for sensor networks. In: IEEE Symposium on Security and Privacy. (2003)
6. Zhou, L., Haas, Z.J.: Securing ad hoc networks. *IEEE Network* **13** (1999) 24–30
7. Gong, L.: Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications* **11** (1993) 657–662
8. Hubaux, J.P., Buttyan, L., Capkun, S.: The quest for security in mobile ad hoc networks. In: Proceeding of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC). (2001) 146–155
9. Modadugu, N., Boneh, D., Kim, M.: Generating RSA keys on a handheld using an untrusted server. In: Cryptographer's Track RSA Conference. (2000)
10. Carman, D., Kruus, P., Matt, B.: Constraints and approaches for distributed sensor network security. Technical Report #00-010, NAI Labs (2000)
11. Perrig, A., Szewczyk, R., Wen, V., Culler, D.E., Tygar, J.D.: SPINS: Security protocols for sensor networks. In: Mobile Computing and Networking. (2001) 189–199
12. Basagni, S., Herrin, K., Bruschi, D., Rosti, E.: Secure pebblenets. In: Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing. (2001) 156–163
13. Eschenauer, L., Gligor, V.D.: A key-management scheme for distributed sensor networks. In: Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS-02). (2002) 41–47
14. Zhu, S., Xu, S., Setia, S., Jajodia, S.: Establishing pair-wise keys for secure communication in ad hoc networks: A probabilistic approach. Technical Report ISE-TR-03-01, George Mason University (2003)
15. Dolev, D., Dwork, C., Waarts, O., Yung, M.: Perfectly secure message transmission. *J. ACM* **40** (1993) 17–47