A New Approach for Establishing Pairwise Keys for Securing Wireless Sensor Networks

Arno Wacker, Mirko Knoll, Timo Heiber and Kurt Rothermel
Dept. of Distributed Systems, Institute of Parallel and Distributed Systems, Universität Stuttgart
Stuttgart, Germany
{Wacker | Knoll | Heiber | Rothermel}@informatik.uni-stuttgart.de

ABSTRACT

Wireless sensor networks based on highly resource-constrained devices require symmetric cryptography in order to make them secure. Integral to this is the exchange of unique symmetric keys between two devices. In this paper, we propose a novel decentralized key exchange protocol that guarantees the confidentiality of a key exchange even if an attacker has compromised some of the devices in the network. A central objective of the protocol design was to minimize resource consumption on the individual devices. We evaluate the resource requirements of our protocol in terms of memory requirements, CPU usage and network traffic both through theoretical analysis and through simulations.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection, Cryptographic controls

General Terms

Algorithms, Security

Keywords

Wireless Sensor Network Security, Key Establishment

1. INTRODUCTION

With the ongoing miniaturization of sensors and actuators daily life devices already have computational power and wireless communication capabilities. One scenario for such wireless sensor or actuator 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.

SenSys'05, November 2–4, 2005, San Diego, California, USA.

Encryption is an elementary technique for securing communications. Encryption schemes, however, require keys to be exchanged before secret communications can take place. In this paper, we provide a secure key exchange scheme especially geared towards resource-constrained environments.

Our approach is suitable for resource-constrained 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 if a device is subverted by an attacker, the key exchange for the remainder of the network remains functional.

In [1], we 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. Our solution is based on using s node disjoint paths in an s-connected graph to distribute key shares. The nodes then use these to generate the session key. As we show below, finding node disjoint paths with only limited resources on each device is a non-trivial task. Thus we present here a new approach for establishing keys without having to explicitly find these paths.

The remainder of this paper is organized as follows: We start out in Sec. 2 giving our system model and requirements. In Sec. 3 we introduce the basic principle of decentralized key establishment on very resource-constrained devices. In Sec. 4, our approach is discussed in detail. We evaluate our key establishment protocol in Sec. 5. We conclude this paper with a discussion of related work (Sec. 6) and a summary with ideas for future work (Sec. 7).

2. SYSTEM MODEL & REQUIREMENTS

For the purpose of this paper we assume that a network consists of a set of independent devices, each with its own processor and memory, that communicate 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 so that communication between any two devices is always possible (direct or indirect via ad-hoc routing), and a transport layer mechanism that recovers from packet losses. The number of devices is not predetermined or constrained in any way, and may change due to the introduction of new devices to the network or device deactivation. The devices of such a network have to be inexpensive, and therefore they will only have limited resources.

For the purposes of this work, we assume only eavesdropping adversaries. An adversary may eavesdrop on the communication channel or have control of a subset of the devices in the network and learn any data that is transmitted through them. For the sake of simplicity we do not consider device failures or devices that exhibit byzantine behavior. However, a device may use a deactivation protocol to leave the network in a controlled fashion. Possible ex-

tensions to cope with other types of attackers and device failures are mentioned in Sec. 7.

Consideration of these properties leads to the following requirements:

- The key distribution scheme must be decentralized – it must remain functional even if some devices are subverted.
- Symmetric cryptography is used in order to deal with resource limited devices which do not have the computational power to perform asymmetric cryptographic algorithms [2].

3. MULTIPATH KEY ESTABLISHMENT

In the following discussion, we use the notion of a *key graph*: A key graph is 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 $(v_1, v_2) \in E$ if and only if the nodes v_1 and v_2 share a symmetric key. We use the term *device* to indicate the physical device and the term *node* to indicate the representation of that device in the key graph.

The fundamental approach for establishing a shared key between two devices in a decentralized way works as follows [3, 4, 5, 1]: If the key graph contains s node-disjoint paths between the corresponding source and target device (initiator and recipient of the exchange), the source device randomly generates s key shares $k_1 \ldots k_s$ of identical length and sends them over the s device-disjoint paths to the target device (Fig. 3). On each link of a path, the key share is encrypted and integrity-protected with the existing shared key for this link. The newly exchanged key k is calculated as $k = k_1 \oplus k_2 \oplus \ldots \oplus k_s$, where $\oplus$ is the bitwise XOR operation. We assume that once the key between a pair of devices is established, it cannot be compromised without subverting one of the devices.

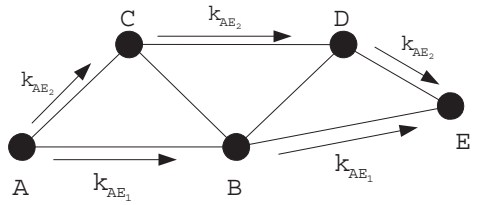


Figure 1: Key establishment ($s = 2$)

It is obvious that once the source or target device is subverted a secure key establishment cannot be guaranteed any more, since all newly established keys will be known to the attacker as well. This is a general fact and not limited to our key establishment protocol. Thus for the purpose of this paper we do not consider compromised source or target devices.

Without access to all key shares, an attacker cannot recover the correct key. Consequently, if it can be assured that the key shares are communicated over s node-disjoint paths of the key graph, the attacker will need to subvert at least s nodes (one on each path) to compromise the newly established key.

A problem shared by the above approaches is that actually finding node-disjoint paths is not trivial, especially on memory constrained devices: An adaptation of basic algorithms [6] would require each node to have complete knowledge of the key graph. The alternative is a reactive protocol, i. e. to run a path discovery protocol every time a key exchange takes place, e. g. by adapting the approaches of [7] and [8], which use distance vector routing algorithms to collect path information as needed. In this protocol, the source device floods the network with ROUTE REQUEST packets and receives multiple ROUTE REPLY packets indicating all available paths (source routing). Large networks will lead to long paths between nodes and require larger packet sizes to store the path.

The overall memory needed by all of the above algorithms exceeds $\Omega(n)$. For resource-limited devices, efficient distributed algorithms are necessary. In that area, Srinivas et al. [9] proposed an algorithm (based on fundamental work by [10]) that has a time complexity of $O(kn^2)$. However, this algorithm benefits from the Wireless Multicast Advantage, which is not applicable in our scenario since we need hop-by-hop encryption to guarantee security.

Fortunately, a closer look reveals that *finding* s node-disjoint paths in a graph is not actually necessary; it suffices to make each step of the key establishment protocol resilient to the compromise of $s - 1$ nodes. Based on this observation, we modify the fundamental approach by developing it into a reactive algorithm that establishes additional edges in the key graph with the intent of shortening the s node-disjoint paths to a length of 2. Our algorithm has advantageous properties in terms of memory usage and network message size. It has a time complexity of $O(kn)$ and can be parameterized to use only constant memory on each device.

4. RKEP: THE RECURSIVE KEY ESTABLISHMENT PROTOCOL

In this section, we present a novel approach for establishing pairwise shared keys, that does not require a node to discover s node-disjoint paths to the target node. This approach is based on two observations: Firstly, in our application scenario, the key graph can be modified by temporarily introducing new key edges to it. Secondly, if two nodes in the key graph share s common neighbors, discovering the s node-disjoint paths is trivial, because the source node (i. e. the node initiating the key establishment) only needs to ask its direct neighbors.

The Recursive Key Establishment Protocol, RKEP, consists of two components: The graph construction algorithm and the key establishment protocol. The graph construction algorithm is responsible for building the initial key graph and for establishing an initial set of key edges whenever a new node is added to the graph. The key establishment protocol works on graphs built using this algorithm and establishes a key edge between any pair of nodes on demand.

We start out by giving an overview of the key establishment protocol in the next subsection. After that, data structures and protocols are discussed in detail in Sections 4.2 and 4.3. We analyze the key establishment protocol with regard to its requirements on the key graph in Sec. 4.4. Based on that analysis, the graph construction algorithm is then derived in Sec. 4.5. In Sec. 4.6 we provide the proof that our presented solution always guarantees a successful key establishment and in Sec. 4.7 we show how this new structure of the network can be achieved in practice. Finally, we present some thoughts about an extended attacker model in Sec. 4.8.

4.1 Overview

The fundamental concept behind RKEP is to augment the key graph by temporarily inserting additional key edges. More precisely, our goal is to add edges such that there are s paths between source and target node that have exactly one intermediary node, i. e. there exist s 2-hop paths between source and target. After that, the key establishment is straightforward.

The s 2-hop paths are established by forwarding the key establishment query recursively through the network, as follows: The source device sends a *Key-Establishment-Query* to all neighboring

devices. The devices receiving this query now check if they have a shared key with the target device. If this is the case, the device will respond with an *Established*-message which is sent to the querying device. If the device does not have a shared key with the target device, it will recursively forward the query to its neighbors. After receiving at least s *Established*-messages, a device can establish a new key to the target device using the devices which sent the *Established*-message as the intermediary devices.

Whenever a new key is established by this method, the device that established the new key (possibly as a recursive query) sends an *Established*-message to all devices it has outstanding *Key-Establishment-Query*-messages from. This way, requesting devices are notified of the new key and the *Established*-messages propagate back to the source.

In addition to the *Established*-message, the device that established the key sends a *Cancel*-message to all neighbor devices from which it got an *Established*-message. This *Cancel*-message informs the nodes that they can remove the established key with the target device, since it was used to establish a new key and is not needed anymore. This "clean-up" is important, since otherwise all nodes of the network would almost simultaneously establish keys to the target device – which is not in line with the highly constrained memory on the devices.

The query is finished when the source gets at least s *Established*-messages (and thus can establish a key to the target). Like all other requesting devices, the source device then sends the corresponding *Cancel*-messages.

The *Cancel*-messages ensure that all temporary keys are removed right after they have been used and the data structures corresponding to this query are removed from the devices.

Fig. 2 shows an example for the key establishment algorithm for $s = 2$: Device E (the source device) needs to establish a key to device A (target). E sends an *Key-Establishment-Query* to its neighbors B and D (Fig. 2(a)). Device B has a key to the target device A, and responds with an *Established*-message. D, however, does not have a key to A yet, and forwards the query to B and C (Fig. 2(b)). Both, B and C have a key to the target device A, and will respond with an *Established*-message sent to D (Fig. 2(c)).

D has now received $s = 2$ *Established*-messages and can itself establish a new key to device A by using B and C as the s distinct intermediary devices.

After establishing this new key, device D sends an *Established*-message to E and a *Cancel*-message to B and C. B and C will ignore this message since they did not actually establish a new key to the target (Fig. 2(d)).

When E receives the *Established*-message from D, it has also received $s = 2$ *Established*-messages and establishes a new key to device A with D and B as intermediary devices. After source device E established a key to the target device A, it sends *Cancel*-messages to B and D. As before, B will ignore this message. D, however, now removes the key to A since it was needed only temporarily in order to help device E establish a new key.

4.2 Data Structures

In the desctiption of the protocol and its associated procedures, the following data structures are used:

***this*:** The device ID of the device running the procedure.

***MyDeviceList*:** List of all devices this device shares a key with.

***QueryID*:** The triple {*SourceID, TargetID, Counter*}. *Counter* is used for unique identification of the current query and set by the source.

***QuerySet*:** A set containing all queries which the device has seen so far and which are still active. All queries stored in *QuerySet* can be referenced through their *QueryID*. For each query the following information is stored:

- ***Key*:** The *QueryID* of this query.
- ***EstablishedSet*:** The set of devices which answered that they share a key with the target (*TargetID*).
- ***RequestingSet*:** The set of all devices from which the query with *QueryID* was received. An *Established*-message modifies this set: When an *Established*-message is received from a device which is part of the *RequestingSet*, it is removed from the *RequestingSet* and added to the *EstablishedSet*.

4.3 Procedures

RKEP comprises of three procedures: The called procedure when a new *Key-Establishment-Query* is received (*onKeyEstablishmentQuery*), the called procedure when an *Established*-message is received in response to such a message (*onEstablishment*) and the called procedure when a *Cancel*-message is received (*onCancelQuery*). The following sections describe these procedures in detail.

4.3.1 *Receiving a* Key-Establishment-Query*-message*

The procedure *onKeyEstablishmentQuery* is called whenever a device receives a *Key-Establishment-Query*-message from a neighboring device. Note that a neighboring device is a device with which this device shares a key (there is an edge in the graph between the corresponding nodes).

The pseudocode for this procedure is shown in Proc. 1. When the procedure is called it first checks if it already shares a key with the target device by looking it up in *MyDeviceList* (line 2). If a key is found and this query was not initiated locally (line 3), an *Established*-message is sent back to the requesting device (line 7).

If no key is found, the device checks if it encountered this query before (line 10). If this is the case, *Sender* is added to the *RequestingSet* for this specific *QueryID*. This makes it possible to notify all requesting neighbors if and when a key from this node to the target was established successfully.

If the device has not encountered this query yet, it adds a new entry to the local query database (*QuerySet*). When adding the new query, the corresponding *EstablishedSet* is initialized with the empty set and the *RequestingSet* with the *Sender* from which this query was just received. Since this device does not share a key to the target yet, it forwards this query to all of its neighbors (lines 16-18).

When a device needs to establish a new key to another device, it initiates a new query by calling the procedure *onKeyEstablishmentQuery* on locally giving its own ID as the *Sender*.

4.3.2 *Receiving an* Established*-message*

A device that shares a key with the target device of a query responds to a *Key-Establishment-Query* with an *Established*-message. A device that receives such a message can therefore deduce that its *Sender* shares a key with the target of that query.

The pseudocode for the corresponding procedure *onEstablishment* is shown in Proc. 2. The device first checks if the *Established*-message it received belongs to an active query (line 2) – if not, the message is ignored.

The sender of the message is added to the *EstablishedSet* for this query and removed from the *RequestingSet* if necessary (lines 3-4).

Since the *EstablishedSet* holds all device IDs which share a key with the target, when there are at least s entries in it, a device can

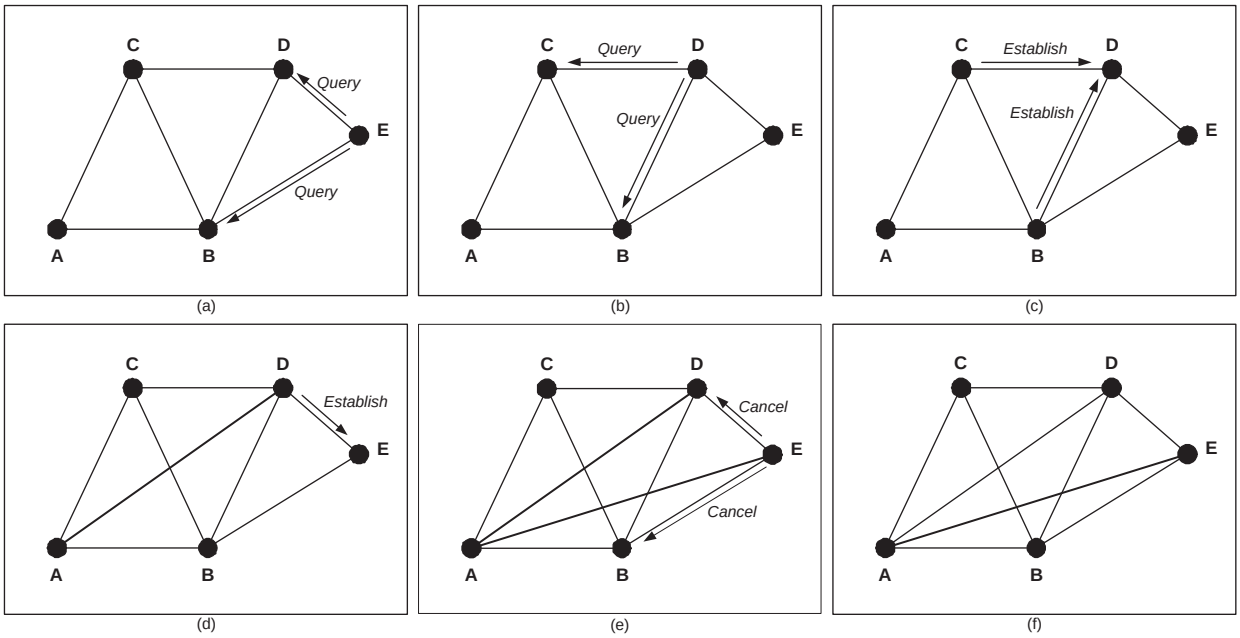


Figure 2: Simple RKEP Example: Device E needs to establish new key to device A. Bold lines are newly established keys.

establish a new shared key with the target, using the devices in the *EstablishedSet* as intermediary devices (line 5). However, establishing a new key to the target device is only necessary if the *RequestingSet* is non-empty or the device is the source device of the query (line 6).

If the new key establishment is successful, the device has to announce this to all devices who requested the key establishment, namely the devices stored in the *RequestingSet*.

Finally, a *Cancel* message is sent to any device in the *EstablishedSet*. This is needed for the "clean-up" described in the next subsection.

4.3.3 *Receiving a* Cancel*-message*

The *Cancel*-message is used to remove all temporary keys and local data for a certain query throughout the network. Receiving a *Cancel*-message for a query from a certain device implies that this device has no longer an interest in the establishment of a key to the target of this query.

The pseudocode for the corresponding procedure is given in Proc. 3: When receiving a *Cancel* message, the *Sender* of the message is removed from the *RequestingSet* (line 5). In case the *RequestingSet* has become empty through this action, all data about the corresponding query is removed from the local database (line 8). Additionally, a key which has been established in the context of this query is removed (line 7).

4.4 RKEP Deadlock

The algorithm presented in the last subsection works on many s-connected graphs, but not all. However, we have devised a construction algorithm for key graphs that can guarantee successful key establishment using RKEP.

To see the situation that might occur, consider the following case (Fig. 3), which results in a deadlock when running RKEP: Device A needs to establish a new key to device J. It therefore sends a *Key-Establishment-Query* to E, C and B (Fig. 3(a)). None of the devices so far have a key with the target J, so they all forward the *Key-Establishment-Query* to their respective neighbors (Fig. 3(b)).

The first device which has a key with the target device and is reached by the query is I. I will respond to E with an *Established*-message. In accordance with Proc. 2, E adds I to its *EstablishedSet* (Fig. 3(c)). Analogously, device F adds H to the *EstablishedSet*. The *EstablishedSets* of device E and F now contain exactly one element, and both devices wait for more *Established*-messages.

Device G gets two *Established*-messages (from I and H), and can therefore establish a new key to J (Fig. 3(d)). G now sends two key shares via I and H to J, thus establishing a new key with J (Fig. 3(e)).

G announces the newly established key by sending an *Established*-message to C and D (Fig. 3(f)), which will add G to their *EstablishedSet*. At this point, the algorithm has reached a deadlock: Devices E, C, D and F have exactly one entry in their *EstablishedSets*, and are waiting for additional *Established*-messages. However, no device can establish a new key to the target and no further *Established*-messages will be sent.

A deadlock occurs in the following situation: There is no device in the graph that does not share a key with the target device and that has at least s neighbors that already share a key with the target device. In other words, RKEP will *never* deadlock if the following statement is true for the key graph: If the algorithm has not yet terminated, there is always a set of s devices that share a key with the target device and for this set, a device exists that shares a key with all devices in the set, but not the target device. If this property can be assured for any pair of source and destination device, RKEP will always be able to establish a shared key between them.

In the next section, we describe a graph construction algorithm that ensures the above property. Furthermore, we prove in Sec. 4.6 that the adapted graph structure guarantees successful key establishment using RKEP.

4.5 Key Graph Construction Algorithm

In this section, we provide an algorithm to construct RKEP-compatible key graphs. The algorithm consists of two methods, one for adding a new node to a (possibly empty) key graph (Sec.

```
1: onKeyEstablishmentQuery(Sender, QueryID)
2: if QueryID.TargetID ∈ MyDeviceList then
3:    if Sender == this then
4:       key already established (no need to engage the protocol);
5:       exit;
6:    else
7:       sendto(Sender, Established(this, QueryID));
8:    end if
9: else
10:   if QueryID ∈ QuerySet then
11:      // we encountered this search already, so we just remember who asked
12:      QuerySet[QueryID].RequestingSet := QuerySet[QueryID].RequestingSet ∪ {Sender};
13:   else
14:      // this is a new search, we have to store it in our database: (Key, RequestingSet, EstablishedSet)
15:      QuerySet := QuerySet ∪ {(QueryID, {Sender}, ∅)};
16:      for all Device ∈ MyDeviceList do
17:         sendto(Device, KeyEstablishmentQuery(this, QueryID));
18:      end for
19:   end if
20: end if
```

Procedure 1: *onKeyEstablishmentQuery*: Reaction on receiving a **KeyEstablishmentQuery**-message

```
1: onEstablishment(Sender, QueryID)
2: if QueryID ∈ QuerySet then
3:    QuerySet[QueryID].EstablishedSet := QuerySet[QueryID].EstablishedSet ∪ {Sender};
4:    QuerySet[QueryID].RequestingSet := QuerySet[QueryID].RequestingSet − {Sender};
5:    if |QuerySet[QueryID].EstablishedSet| ≥ s then
6:       if (|QuerySet[QueryID].RequestingSet| > 0) or (QueryID.SourceID = this) then
7:          doKeyExchange(QueryID.TargetID); // using the nodes from QuerySet[QueryID].EstablishedSet
8:       end if
9:       if key establishment successful then
10:         for all Device ∈ (QuerySet[QueryID].RequestingSet - {this}) do
11:            sendto(Device,Established(this, QueryID))
12:         end for
13:         for all Device ∈ (QuerySet[QueryID].EstablishedSet - {this}) do
14:            sendto(Device,CancelQuery(this, QueryID))
15:         end for
16:      end if
17:   end if
18: end if
```

Procedure 2: *onEstablishment*: Reaction on receiving a **Establishment**-message

4.5.1) and one for removing a node from a key graph while preserving the necessary properties of the graph (Sec. 4.5.2).

4.5.1 Adding a Device to the Network

In [1], we showed how to obtain a s-connected graph by construction: Beginning with a fully connected graph for the first s nodes, each further node is connected to s randomly selected nodes of the existing graph. Due to [11], this method always yields graphs in which there always exist s node disjoint paths between any pair of nodes.

This graph construction method needs to be modified for RKEP, as follows: Again beginning with a fully connected graph for the first s nodes, each further node is connected to a clique of s nodes[1]. Connecting the new node with all s nodes of a clique results in a $(s+1)$-clique. Thus, a graph with n nodes is assembled of $(n-s)$ cliques each of $(s+1)$ nodes.

We will denote such an s-clique as *s-connector* reflecting the fact that a new node can connect to this set of nodes. When introducing a new node to a graph with $n > s$ an s-clique, i.e. an *s-connector* can always be found for the following reasons:

- The first s nodes will be fully connected, thus forming a clique – the first s-connector.
- By having an edge to every node of an s-connector, each newly introduced node itself introduces s new s-connectors.

Fig 4 shows the resulting graph structures for $s = 2$, $s = 3$ and an abstracted view for an arbitrary s.

As can be seen in Fig. 4(a), for $s = 2$ the resulting graph is a planar graph composed only of triangles. The triangles are $(s+1)$-cliques, while each side of any triangle represents an 2-connector (a 2-clique). For $s = 3$, it is easiest to imagine a three-dimensional structure (Fig. 4(b)). The $(s+1)$-cliques, i.e. the 4-cliques can be represented by tetrahedrons in 3D, while the 3-connectors are the 4 triangles of each tetrahedron. Generalizing this concept to an arbitrary s, we always get a graph composed of $(s+1)$-cliques, and each of these cliques has s s-connectors (s-cliques) (Fig. 4(c)). If

[1] A k-clique is a subgraph of k nodes which are fully connected

```
 1: onCancelQuery(Sender, QueryID)
 2: if QueryID ∈ QuerySet then
 3:    if Sender ∈ QuerySet[QueryID].RequestingSet then
 4:       // we got an cancel message from a device who asked us to establish, thus, this device does not need the query anymore
 5:       QuerySet[QueryID].RequestingSet := QuerySet[QueryID].RequestingSet - {Sender};
 6:       if QuerySet[QueryID].RequestingSet == ∅ then
 7:          removeKey(QueryID.TargetID); // agree with target device to remove this key
 8:          QuerySet := QuerySet − {QueryID};
 9:       end if
10:    end if
11: end if
```

Procedure 3: *onCancelQuery*: Reaction on receiving a **CancelQuery**-message

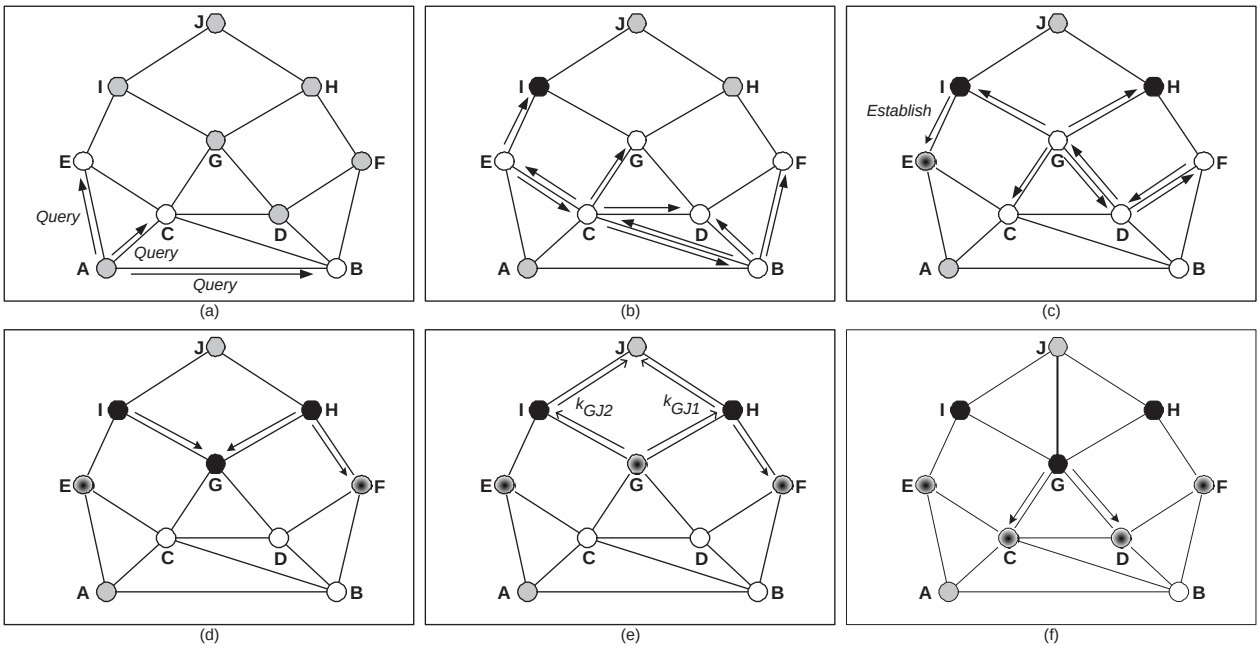


Figure 3: RKEP Deadlock: In (f), E, C, D and F each know of one node that shares a key with the target device J.

the network is correctly generated, any s-connector is unique since they consist of different nodes.

In Sec. 4.6, we show that RKEP can always establish a shared key for a key graph constructed in this fashion. What remains to be discussed, is how can such a graph structure be retained when removing a node from the graph. The next subsection discusses this issue.

4.5.2 Removing a Device from the Network

Removing a device from the network can destroy the s-connected property of the corresponding key graph. A protocol that removes a device in a controlled disconnection procedure, i. e. one in which a device that disconnects from the key graph makes all necessary arrangements before leaving, works as follows: Observe that for RKEP, both the s-connected property of the key graph and the resulting structure of $(s+1)$-cliques has to be preserved. Therefore, we divide the removal procedure in two steps: Preserving the s-connected property and preserving the $(s+1)$-cliques.

Preserving an s-connected key graph. (This part of the removal procedure is also used in our previous work [1].) The solution for this half of the problem 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 the removal procedure is performed, keys can be replaced automatically, using the procedure described in Sec. 3, thus re-establishing the s-connected property of the underlying graph.

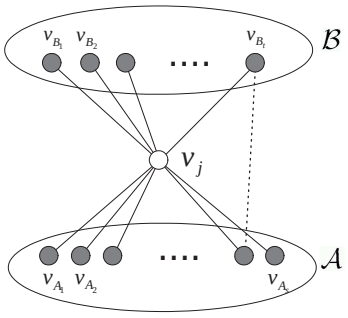


Figure 5: Re-establishing links

To see why this works, let the graph under consideration be $G = (V, E)$ with $V = \{v_1, v_2, \ldots v_n\}$ 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. Then we can

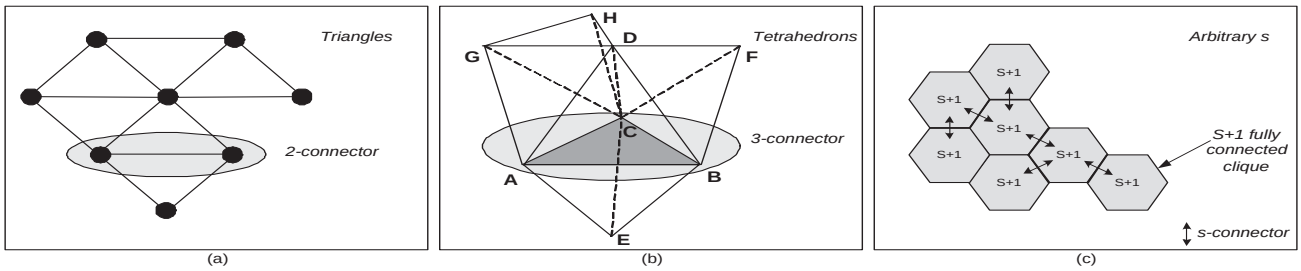


Figure 4: Graph structures when using only s-connectors for introducing new nodes

define two sets, $\mathcal{A}$ and $\mathcal{B}$ (see Fig. 5). 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. 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.

Preserving the $(s+1)$-cliques of the key graph. Whenever a device leaves the network, s s-connectors are going to be destroyed in the key graph. Each device which used such an s-connector needs to re-establish keys in order to connect again to a correct s-connector.

Consider the example in Fig. 6. Device D needs to leave the network and sends a *Leaving*-message to all devices from its $\mathcal{B}$-set (Fig. 6(a)). The *Leaving*-message includes the devices of the s-connector which D used to enter the network. The devices receiving this message compare the received s-connector to their own[2].

Those devices which find an $(s-1)$-match between their own s-connector and the received s-connector can easily identify the device to which they have to establish a new key: It is the only device which is contained in the received s-connector but not in their own. Note that due to the construction algorithm, the s-connectors either match in $(s-1)$ devices or not at all. In Fig. 6(b) devices E and F are in that situation. Comparing the s-connectors, E finds that it needs to establish a new key to B, and F finds that it needs to establish a key with C.

Meanwhile device G has no match at all when comparing the s-connectors. In such a case the node (G) needs to ask a device of its own s-connector for that device's own s-connector – this is the same procedure as when a new device is introduced. In the example, there is only node E to ask, which responds with the list (B, C) (after E is done with re-establishing keys itself). From this list, G can choose a partner randomly (in the example, C).

After establishing new keys, all affected nodes send a *Ready*-message to D (Fig. 6(c)). When D has received a *Ready*-message from all devices in its $\mathcal{B}$-set, it can safely leave the network – all properties have been restored (Fig. 6(d)).

4.6 Proof

In this section we prove by induction over n (total number of devices) that in a network constructed with the above given rules RKEP can always establish a new key between any pair of devices.

[2]Those s devices a device used to enter the network

Induction Statement. In a graph constructed with the RKEP rules, every node can construct a new edge to any other node using RKEP.

Induction Start, $n = s+2$. We start with $n = s+2$ since $s+1$ nodes are always fully connected, and therefore no new edge is needed. Moreover, when introducing a new node to a graph consisting of $(s+1)$ nodes, it does not matter which s nodes are used, since every set of s nodes is fully connected and therefore forms a correct s-connector (see also Fig. 7).

In a graph with $n = s+2$ nodes there are only two nodes which do not share an edge. These two nodes share s common neighbors. Hence, when the corresponding device needs to establish a new key, it sends an *Key-Establishment-Query* according to Proc. 1. Since all s neighbors have a key to the target device, all devices will respond with an *Established*-message, thus enabling the device to establish a new key.

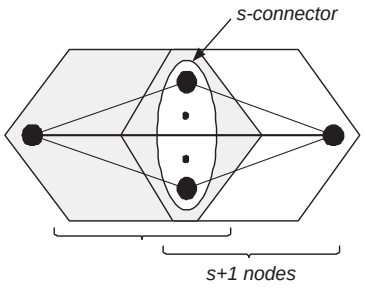


Figure 7: Induction Start: $n = s+2$

Induction Step. Suppose the induction statement is proved for all graphs constructed using 4.7 with at most n nodes. Consider an arbitrary graph of n nodes in which the induction statement is true. We now introduce a new node Y by connecting it to an s-connector (Fig. 8).

Observe that since queries are forwarded to each neighbor, even for the same source-destination pair, queries forwarded from different nodes do not influence each other. Hence, the sequence in which queries are forwarded and answered is not relevant to the protocol. This also means that there only needs to be one successful sequence of key establishment queries for a successful key establishment using RKEP. Especially, for any key graph G, if RKEP can establish a key between two nodes in a subgraph $H \subseteq G$, then RKEP can also establish a key between these nodes in G.

Suppose that for the new graph, an arbitrary node A attempts to establish a key to any node B using RKEP. Three cases have to be examined:

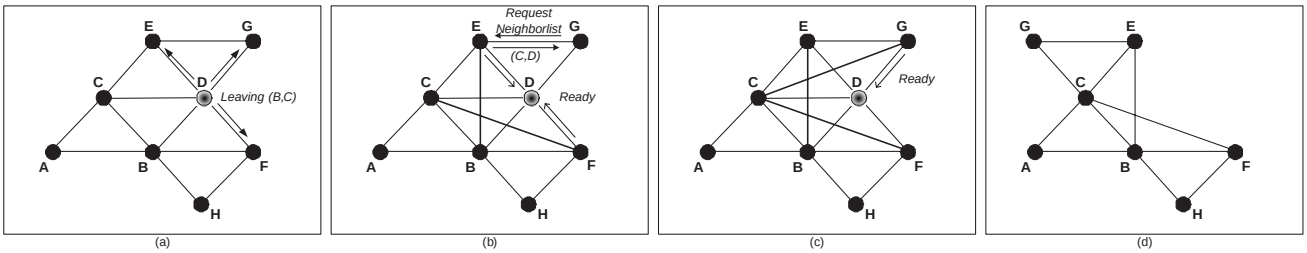


Figure 6: Removing a device from the network ($s = 2$)

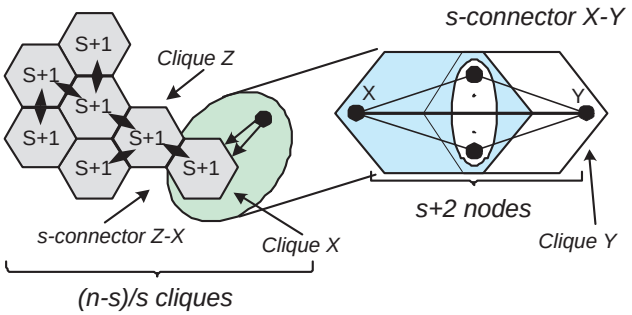


Figure 8: Induction Step: Adding one node

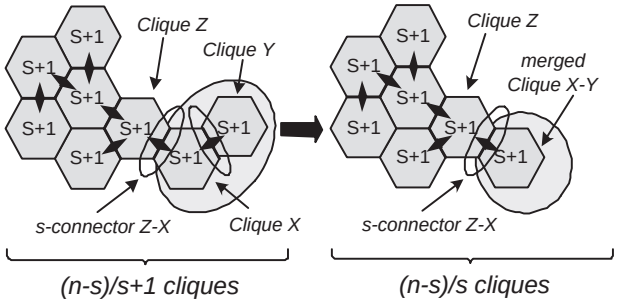


Figure 9: Induction Step, case 3: Merging cliques

Case 1: $(A \neq Y) \wedge (B \neq Y)$. In this case, by the induction statement, the subgraph *without* Y is sufficient to establish a key between A and B.

Case 2: $A = Y$. Using RKEP, all nodes in the s-clique that Y is connected to receive a *Key-Establishment-Query* from Y. Since the query is forwarded recursively, by the induction statement, all these s nodes will establish a key to B, allowing Y to establish a key with B via these nodes.

Case 3: $B = Y$. Recall from Sec. 4.5.1 that the key graph consists of $(s+1)$ cliques connected by s-cliques. Without loss of generality, consider the subgraph obtained by only examining the shortest sequence of $(s+1)$-cliques from the one containing A to the one containing Y: By construction, the s nodes to which Y did initially connect are part of an $(s+1)$-connector. Thus, there is a node X that is part of this $(s+1)$-connector, but does not share a key with Y (Fig. 8). X will eventually receive a *Key-Establishment-Query*, and establish a key with Y via the s other nodes. By doing so, an $(s+2)$-connector is formed.

Now, remove a node from this $(s+2)$-connector that is neither X nor Y nor part of the s-connector joining X's s-connector to neighboring node Z's s-connector (Fig. 9). Note that the graph obtained in this fashion is still a graph that could have been generated by 4.7. However, this graph only has $m \leq n$ nodes, and all nodes have received and forwarded a *Key-Establishment-Query* for nodes A and Y. Hence, by the induction statement, a key between A and Y can be established. □

4.7 Practical Considerations

In the last sections, we discussed a method to obtain an RKEP-compatible key graph. What remains to be discussed is how this structure can be generated in a practical setting, i.e. when introducing new devices to the network. There are at least three possibilities to achieve this:

User-driven Introduction. Whenever a device gets introduced in the network, the user randomly chooses a device already in the network and establishes a new key in an out-of-band secure way (e.g. via a secure physical connection). Any device is part of an $(s+1)$ connector from the time of its introduction to the network. Therefore, it can send the IDs of s devices that are part of an s-connector to the new device (in fact, there are s such s-connectors). The device now displays them to the user, e.g. on a small display, who can now choose $s-1$ more devices and finish the introduction process in the same way as for the first device. Having established a new key to the other devices, the new device is fully introduced in the network and can use RKEP to establish any further keys. It is up to the user to balance the amount of keys stored on the nodes to prevent memory overflow.

Master Programming Device. The usage of a master device is another possibility to introduce new devices to a network. In this case, the master device is used only to introduce devices, and is not needed later on for the correct function of the network. Such a master device would share a *unique master key* with each device in the network (set by the manufacturer or by the user before using it for the first time).

The master device is implicitly trusted and can be authenticated to each device through the master key. For introducing a new device to the network, the master device chooses s devices from the network that form an s-connector) and generate a unique key between them and the new device. Towards that end, the master device stores a global view of the entire network.

After the master device has established s new keys between the new device and an s-connector of the network, it can be taken offline. The use of master device is on the one hand less cumbersome for the user, since he only has to introduce the new device to the master device. On the other hand, a centralized component – although usually offline – was introduced, which could become the primary target of an attack.

Automatic Key Establishment. As a third alternative, RKEP itself can be used in order to set up the correct keys. The user establishes new keys with s randomly chosen devices of the network. Even if this results in the new device *not* being connected to an correct s-connector, the new device can request the first s devices of one of the devices to which it already has a key. After obtaining this information, it can – just like in the manual case – establish new keys to this s connector using RKEP. Having established new keys with a correct s-connector, the initial keys exchanged by the user can be removed. The difference to the manual case is that the user only needs to exchange s keys when introducing the new device and the device itself will establish keys to a correct s-connector.

The automatic approach is sound for the following reason: The initial devices are already part of the network and can therefore establish a key with any other device of the network. When these devices have all established a key to the same device, the new device can do so as well, since it now has s intermediary nodes to the device. Thus a device, even if not yet fully introduced, can establish new keys to devices in the existing network – thus establishing keys to an correct s-connector.

4.8 Extended Attacker Model

As stated in Sec. 2 for the purpose of this work we assumed only eavesdropping (*passive*) attackers. However, to achieve a more realistic attacker model the analysis of the following (*active*) attacker classes is also needed:

***fail-stop* attacker:** This attacker silently halts all functions of a device. This class includes device failures (e.g. due to low power).

***byzantine* attacker:** This attacker may act arbitrary in order to prevent the correct execution of the key-distribution protocols. This class of attacker subsumes the *fail-stop* attacker class.

Note that we do not consider denial-of-service on the physical layer (e.g. jamming the frequency) since this is always possible and can only be counteracted by measures on the physical layer (e. g. frequency hopping)

A complete and thorough analysis of the extended attacker model is beyond the scope of this paper and still work in progress. However, as we showed in [12], the basic approach for coping with the *fail-stop* attacker is the introduction of redundancy. This means that we build our initial key graph with a connectivity level of $z > s$, while the algorithm needs only s node-disjoint paths to establish a new key. Thus even if there are f device failures (*fail-stop* attackers) with $f < z - s$, there are still at least s cooperating devices left. Coping with *byzantine* attackers is more complicated and can be achieved through the introduction of replication and voting mechanisms.

5. PROPERTIES AND EVALUATION

In this section, we analyze of the properties of our approach with respect to memory usage and network traffic. This is followed by simulation results contrasting the performance characteristics of our protocol with those achieved using a key exchange protocol using the max-flow algorithm for discovery of node-disjoint paths.

5.1 Analysis

As shown in Sec. 4.6, RKEP together with the key graph construction rules guarantees that key establishment is possible for any two devices in the network. To analyze our key establishment protocol we now discuss the theoretical worst cases with respect to memory usage, network traffic.

5.1.1 Memory Requirements

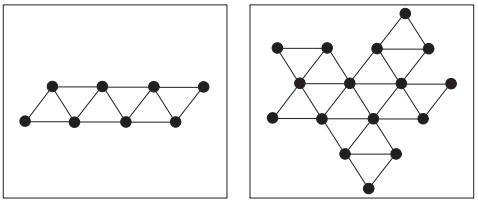
Network Setup. Each device needs s shared keys to connect to the initial key graph. Since each key is stored on two devices, we have an average of $2s$ keys per device. Thus, the initial key graph requires on average $O(1)$ memory space on each device.

Key Establishment. Each entry in the *QuerySet* of a device (one per query) consists of the *RequestingSet* and *EstablishedSet*, which, taken together, contain at most the direct neighbors of this device, i.e. on average $2s$. Moreover, for each query, at most one temporary key will be established between a device and the target device. Thus, a query requires $O(1)$ temporary space on each intermediate device and at most $O(n)$ on the target device.

The memory requirement on the *target device* depends on the key graph structure: Every device that receives an RKEP query tries to establish a new key to the target device. When it has done so, it sends a Cancel-message notifying other devices that their temporary keys are no longer needed. Thus, at least s temporary keys need to be held on the target device in order to establish a new one, i.e. in total $s + 1$.

However, there is not only one device trying to establish a new key to the target, the amount of temporary keys on the target device depends on how many devices do simultaneously establish a key to the target device. The number of such devices is influenced by the key graph structure. Fig. 10(a) shows a graph structure which we call a "chain". In this graph, the number of temporarily needed keys on the target device is exactly $(s+1)$. In contrast, in the graph structure given in Fig. 10(b), $4(s + 1)$ temporary keys are needed.

In order to keep the memory requirements on the target device constant, we have developed an additional algorithm which influences the key graph structure when introducing a new device. The algorithm is based on limiting the memory for the initial key graph with an parameter. When enforcing this parameter, the key graph grows in a controlled fashion and the "chain" structure can be enforced. Discussing that algorithm in detail is beyond the scope of this paper – however, we note that the memory requirement on the target device can be tuned to $k(s + 1)$, a constant value.



(a) "Chain" (b) Random

Figure 10: Graph Structures $(s = 2)$

5.1.2 Network Traffic

Network Setup. When introducing a new device to the network, the amount of exchanged packets depends on which method for introducing a device is used. When using the *user-driven* or the *master-device* approach (Sec. 4.5.1), only neighbor lists are sent over the wireless channel. Since the size of the neighbor lists is constant, a constant number of messages is transmitted for each new device. Thus, for the setup of the whole network we need $O(n)$ packets. In contrast, when using *automatic key establishment*, we

need at most $(s-1)$ new key establishments, where key establishment is in $O(n)$ (see next paragraph for explanation). Thus, the number of messages needed for the network setup is $n(s-1)O(n)$, i.e. $O(n^2)$.

Key Establishment. The total amount of messages for one query is linear in n, i.e. $O(n)$: Each *Key-Establishment-Query* is sent at most twice on each key edge, each *Established*-message is sent at most once per key edge and each *Cancel*-message is also sent at most once per key edge. Thus, we get at most $4|E|$ messages with $|E| = (n-s)s + \frac{s(s-1)}{2}$, which is linear in n.

Message Loss. A transport layer mechanism that recovers from message loss can be achieved with at most linear overhead: Let X be the number of messages sent in the ideal case. In order to recover from message loss, a stop-and-wait protocol can be used, where each protocol message is acknowledged and re-sent when the acknowledgement fails to arrive. In case a protocol message is lost, one retransmission is needed, whereas in case an acknowledgement is lost, both the original protocol message and the acknowledgement need to be retransmitted. For a uniformly distributed message loss probability p, the number of messages actually sent can be approximated as $X(p) = 2X + pX + 2pX = (2+3p)X$.

5.1.3 Summary

We summarize our theoretical analysis of RKEP in Tab. 1:

	Memory	Traffic
Network Setup	$O(1)$	$O(n)$
Key Establishment	$O(1)$	$O(n)$

Table 1: RKEP Performance

5.2 Simulation Results

To evaluate our approach, we have built a simulator which is able to use different strategies for key establishment or network setup. In the simulation, nodes are always in transmission range of each other and transmitted packets are never lost. That way, the measurements are not polluted with effects not directly related to the key establishment protocols.

We simulated RKEP, as well as our original approach. In the simulation, the path discovery algorithm for the original approach is based on a global view of the network on each device: Each device that is added to the network transmits a link-state packet containing all its neighbors. This packet is flooded through the network using the encrypted links. With a global view of the network, each device can locally search for the node-disjoint paths in the key graph using the Max-Flow (MF) algorithm [13].

We used scenarios of 25 to 250 devices in increments of 25. We also varied the security level s to show how this influences the measured values. We measured the number of packets and the amount of memory used on a device. The memory measurements are based on the assumption of 128-bit symmetric keys and 16-bit device IDs.

5.2.1 Memory Requirements

Network Setup. Fig. 11 shows the maximum memory use for a single device while the network grows. The diagram also shows two graph structures for RKEP: one for a enforced chain-structure of the graph and one for a random structure. In the RKEP-chain case, the memory requirement remains constant – the algorithm enforces exactly that. Without enforcing any graph structure, the memory requirement grows about linearly with the number of devices. For storing a graph in the Max-Flow-case, we need at least linear memory. With 2 bytes per device ID, the minimum amount needed for storing the whole graph would be $2ns$ bytes. For comparison, these amounts are also shown in the diagram as grey lines. The worst-case memory needed to store keys for all devices in the network is also shown as a grey line (*fully connected graph*).

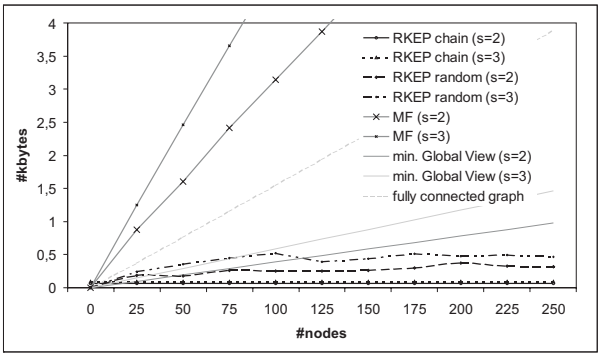


Figure 11: Maximum memory usage on a device

Key Establishment. Fig. 12 shows the maximum increase in memory use of all devices during key establishment. As can be seen, memory usage is constant for the RKEP-chain case and linear for the RKEP-random case. Note that the linear growth is mostly due to the memory requirements on the target node. An interesting effect with RKEP is that for higher values of s, the number of temporary keys that needs to be established actually goes down. This is a result of the s-connectors growing larger. The memory consumption of the Max-Flow-based approach is very high for the source node (which has to compute the node-disjoint paths). For this node, memory consumption grows very fast and in a non-linear way. This is due to the fact that Max-Flow algorithms work on directed graphs and find *edge*-disjoint paths: In order to be used for our purposes, a graph transformation is needed that generates a graph of a size proportional to the square of the original graph size.

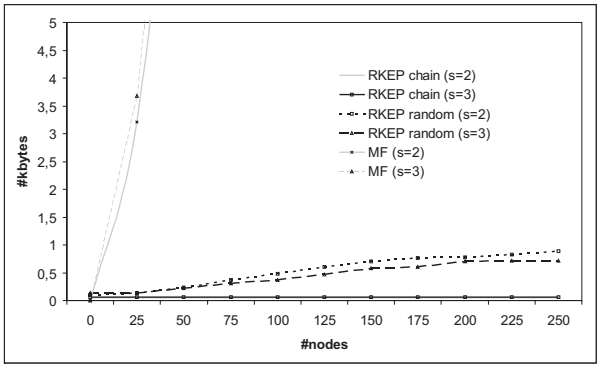


Figure 12: Memory usage on a device during key establishment

5.2.2 Network Traffic

The next two diagrams show our simulation results with regard to the number of packets sent over the network. Note that the graph structure does not affect the network traffic caused by RKEP: For

network setup, only a constant number of packets is needed to exchange the initial keys and for key exchange, RKEP messages will be sent over all edges, regardless of the structure of the graph.

Network Setup. Fig. 13 shows the number of packets needed to set up the network for user-driven RKEP and Max-Flow. The diagram shows the number of packets needed for incrementally building the graph by adding nodes one to n. With s packets per per node, traffic grows linearly for RKEP, while establishing the global view requires linear effort per node and results in quadratic growth for the Max-Flow-based approach.

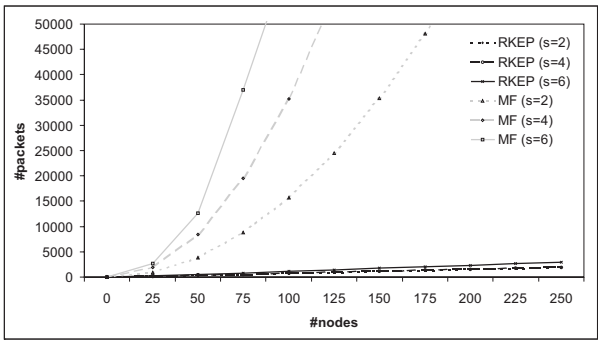


Figure 13: Network traffic during network setup

Key Establishment. Finally, Fig. 14 shows the amount of network traffic needed per key establishment. Here, the proactive Max-Flow approach can trade off memory for network traffic: The only packets needed are those that transport the actual key shares. The solid black line shows the worst-case effort needed to do this. RKEP does not fare as well, but still manages a linear growth rate with respect to the number of devices.

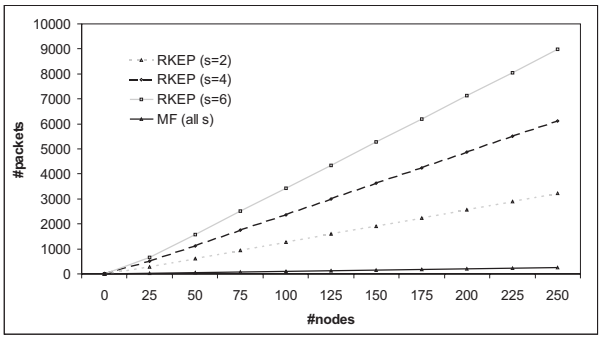


Figure 14: Network traffic during key establishment

Message Loss. To recover from message loss we added a simple stop-and-wait protocol. We measured the amount of network traffic for 10% and 20% message loss probability – usual loss rates for wireless networks. We also performed measurements for a 0% loss rate to show the overhead of the stop-and-wait protocol (see Fig. 15). The continuous line corresponds to "RKEP ($s = 2$)" from Fig. 14. The introduced overhead is in line with the analytical results.

5.3 Summary

As our theoretical analysis and also the simulation showed, the memory requirements of our algorithm are constant in the case of a "chain" structure and sublinear otherwise. Thus from the point of memory footprint our algorithm scales almost infinitely. In contrast, the network traffic during key establishment grows linearly and therefore limits the scalability. However, this is a general problem of a reactive algorithm that does not use any knowledge about the network: The only way to "find" another device is to query all other devices, i. e. communicate with $(n - 1)$ devices. Thus, for a reactive algorithm with no additional information linear growth is also the lower bound. Our algorithm represents a good trade-off between memory requirements and network traffic. We achieve constant memory requirements while causing linear growth in network traffic, which is the lower bound for reactive algorithms.

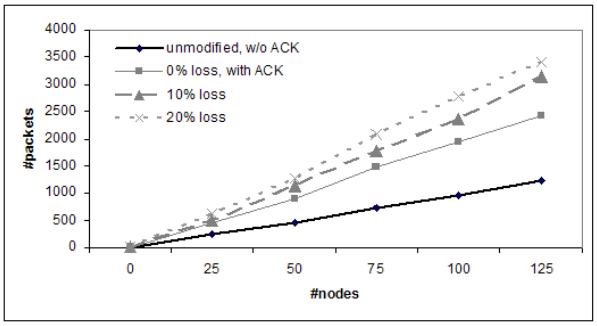


Figure 15: Network traffic during key establishment including message loss for RKEP ($s = 2$)

6. RELATED WORK

Several solutions for securing wireless sensor or ad-hoc networks have been proposed in the literature. A number of them employ asymmetric cryptography to reach their goal [14, 4]. Using asymmetric cryptography on highly resource constrained devices is often not feasible due to delay, energy and memory constraints [15, 2]. Even though very recent publications show that it is *possible* to use asymmetric cryptography on such constrained devices, these implementations still use considerably more valuable resources. For comparison, an implementation of symmetric cryptography on a 8-bit microcontroller uses 7150 bytes of program memory [16]. In contrast, the smallest available implementation of asymmetric cryptography using elliptic curve cryptography uses about 30 kB of program memory [17]. As a consequence, for the common configuration in which an asymmetric algorithm is used for key exchange and a symmetric one for communication, the memory footprint increases by a factor of more than five. Miniaturization and economical considerations are also a major factor for sensor networks, counteracting the increase of resources achieved by technological progress: There is an interesting observation by Karlof and Wagner that sensor network devices will more likely ride Moore's law *downward* [18]. They make the point that instead of doubling computational power every 18 months it may be more likely that the devices become even smaller and cheaper solutions are sought.

Several approaches using symmetric cryptography make use of a central base station [19]. However, the availability of a central authority (base station) in an ad-hoc network cannot always be assumed. Moreover, such a central authority clearly is the first target for any attacker and thus becomes a single-point-of-failure.

To overcome the drawbacks of a central authority, recent work uses symmetric cryptography in a purely decentralized fashion, i.e. without the need for any central authority. However, these approaches usually assume that such networks can be pre-configured

and that it is a priori known how big the network is, or how big it might get. These approaches do also not address the issue of easy addition or removal of devices. The following approaches have been published:

A simple approach was shown by Basagni et al. in [20]. However, their approach assumes tamper-resistant devices, a notion that we consider problematic in the present context.

Eschenauer and Gligor [21] proposed a solution using random key predistribution. Before deployment, every device is supplied with a random 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 key graph connectivity can be made afterwards, thus no guarantee that two arbitrary device can communicate securely.

Extensions have been introduced by Chan et al. [3]. They present three different approaches: In the approach most directly related to ours, a set of secure and authenticated links can be established after deployment of the sensors. Again, due to the still-random predistribution, two arbitrary devices might not be able to establish a secure link without relying on other devices. For this reason, secure communication between arbitrary devices may not always be possible.

The approach by Zhu et al. [22] also uses an initially distributed set of random keys. In addition to [3], Zhu et al. propose a pairwise key establishment protocol using *multiple* paths. In this way, by splitting of a pairwise key over multiple untrusted paths (as initially proposed in [5]) resistance against attackers can be improved. However, 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 contrast to the approach presented here, no presumption about the real number of the device-disjoint paths is possible.

A very recent approach by Chan et al. [23] proposes a deterministic scheme for key predistribution which guarantees for any two nodes from the network that there always exists one intermediary node (i.e. a node that shares a key with both of them). This intermediary node is then used to establish a new key. However, in their approach, the secret is not split up, making the intermediary node an implicitly trusted device. Thus, the secrecy of a new key cannot be guaranteed after a single node has been compromised. In contrast our approach guarantees the secrecy of any new key for up to s compromised nodes.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel, memory-efficient approach for guaranteed key establishment in wireless networks. By adjusting the value of s, our approach can be parameterized according to the security needs of the network.

As stated in Sec. 4.8 the presented approach does not yet consider devices that fail or exhibit malicious behavior, i. e. the extended attacker model. The thorough analysis of the extended attacker model is topic of future work.

We are in the process of integrating our protocols into TinySec [16] on the Berkeley Mica2 Motes [24]. For the future, this provides us with the tools to report on practical experience with our approach in a realistic environment. Furthermore, we are going to conduct further performance studies on a emulated network infrastructure, as described, for example, in [25].

Acknowledgements

Our thanks go to Christian Becker and Pedro José Marrón, who read earlier drafts of this paper and provided many helpful suggestions. We also thank Steffen Maier and Gregor Schiele for their help with the simulations. Furthermore, we would like to thank our shepherd, Adrian Perrig, and the anonymous reviewers for their valuable comments.

8. REFERENCES

[1] A. Wacker, T. Heiber, and H. Cermann, "A key-distribution scheme for wireless home automation networks," in *Proceedings of IEEE CCNC 2004*, IEEE Communications Society. Las Vegas, Nevada, USA:, Jan., 5-8 2004.

[2] M. Brown and D. Cheung, "PGP in constrained wireless devices," in *Proceedings of the 9th USENIX Security Symposium*, 2000.

[3] H. Chan, A. Perrig, and D. Song, "Random key predistribution schemes for sensor networks," in *IEEE Symposium on Security and Privacy*, May 2003.

[4] L. Zhou and Z. J. Haas, "Securing ad hoc networks," *IEEE Network*, vol. 13, no. 6, pp. 24–30, 1999.

[5] L. Gong, "Increasing availability and security of an authentication service," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 5, pp. 657–662, 1993.

[6] A. Itah and M. Rodeh, "The multi-tree approach to reliability in distributed networks." in *Proceedings of the 25th Symposium on FOCS*, 1984.

[7] D. Sidhu, S. Abdallah, and R. Nair, "A distance vector algorithm for alternate path routing." 1990, submitted for publication.

[8] S. J. Lee and M. Gerla, "Split multipath routing with maximally disjoint paths in ad hoc networks." in *IEEE International Conference on Communications*, 2001, pp. 3201–3205.

[9] A. Srinivas and E. Modiano, "Minimum energy disjoint path routing in wireless ad hoc networks." in *ACM Mobicom*, 2003.

[10] J. W. Suurballe, "Disjoint paths in a network." *Networks*, vol. 4, pp. 125–144, 1974.

[11] K. Menger, "Zur allgemeinen Kurventheorie," *Fund. Math.*, no. 10, pp. 96–115, 1927.

[12] A. Wacker, T. Heiber, H. Cermann, and P. J. Marrón, "A fault-tolerant key-distribution scheme for securing wireless ad-hoc networks," in *Proceedings of the second Conference on Pervasive Computing, Pervasive 2004*. Vienna, Austria: Springer-Verlag, April, 19-23 2004.

[13] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," *SIAM Journal on Computing*, vol. 4, no. 4, pp. 507–518, Dec. 1975.

[14] J.-P. Hubaux, L. Buttyan, and S. Capkun, "The quest for security in mobile ad hoc networks," in *Proceeding of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, 2001, pp. 146–155.

[15] D. Carman, P. Kruus, and B. Matt, "Constraints and approaches for distributed sensor network security," NAI Labs, Tech. Rep. #00-010, Sept. 2000.

[16] C. Karlof, N. Sastry, and D. Wagner, "Tinysec: a link layer security architecture for wireless sensor networks," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM Press, November 3–5 2004, pp. 162–175.

[17] V. Gupta, M. Millard, S. Fung, Y. Zhu, N. Gura, and S. Shantz, "Sizzle: a standards-based end-to-end security architecture for the embedded internet," in *Proceedings of Third IEEE International Conference on Pervasive Computing and Communications, PerCom 2005*. Kauai, Hawaii: IEEE, March 8–12 2005, pp. 247–256.

[18] C. Karlof and D. Wagner, "Secure routing in wireless sensor networks: Attacks and countermeasures," *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*, vol. 1, no. 2–3, pp. 293–315, September 2003.

[19] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar, "SPINS: Security protocols for sensor networks," in *Mobile Computing and Networking*, 2001, pp. 189–199.

[20] S. Basagni, K. Herrin, D. Bruschi, and E. Rosti, "Secure pebblenets," in *Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing*, 2001, pp. 156–163.

[21] L. Eschenauer and V. D. Gligor, "A key-management scheme for distributed sensor networks," in *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS-02)*, 18–22 2002, pp. 41–47.

[22] S. Zhu, S. Xu, S. Setia, and S. Jajodia, "Establishing pair-wise keys for secure communication in ad hoc networks: A probabilistic approach," George Mason University, Tech. Rep. ISE-TR-03-01, Mar. 2003.

[23] H. Chan and A. Perrig, "PIKE: Peer intermediaries for key establishment in sensor networks," in *Proceedings of IEEE Infocom*, Mar. 2005.

[24] "Crossbow Technology Inc.: Motes, Smart Dust Sensors, Wireless Sensor Networks," Webpage. [Online]. Available: http://www.xbow.com

[25] D. Herrscher, S. Maier, J. Tian, and K. Rothermel, "A Novel Approach to Evaluating Implementations of Location-Based Software," in *Proceedings of the 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2004)*, San Jose, CA, USA, July 25–29 2004, pp. 484–490.