GROUP KEY ESTABLISHMENT

Przemysław Błaśkiewicz
Wroclaw University of Technology
Wrocław, Poland
przemyslaw.blaskiewicz@pwr.wroc.pl

Michał Koza
Wroclaw University of Technology
Wrocław, Poland
michal.koza@pwr.wroc.pl

Tomasz Strumiński
Wroclaw University of Technology
Wrocław, Poland
tomasz.struminski@pwr.wroc.pl

1. Introduction

In our work we focus on the problem of secret key distribution among a group of sensors. One can imagine a situation where a large number of sensors are deployed in order to collect data. For some reason, the owner of the network may want this data to remain secret; hence an encryption of transmissions must be used. This is even more important since transmission is wireless. We deal with small and simple devices with limited computational, energetic, and memory resources, thus using asymmetric cryptography is impossible or at least undesired. On the other hand, the use of multiple, strong (and therefore long) symmetric keys for communication with each neighbor is impossible due to memory scarcity and poor immunity to key harvesting attacks. Indeed, during key pre-installation there is no knowledge about future architecture of the network so each node would have to keep a key for communication with every other node. Using one symmetric key for all communications is not a good idea either. A deployed sensor network is usually not guarded nor hidden, so attacker can easily find some nodes by, e.g., light reflex or unlucky position. Simple devices are not tamper-proof thus extracting the key from one device would open the entire communication to the attacker. Our solution is based on periodical establishment of a temporary group key (*GK*). The establishment is protected using the key levels protocol, thus it is difficult to capture the temporary key during its transmission but also stealing from captured node will suffice only for some short period of time.

1.1 Key levels

Our key establishment protocol utilizes the *key levels* protocol [CGK09]introduced by Cichoń, Grzaślewicz and Kutyłowski to securely transmit the group key to group members. The *key levels* protocol is a modification/extension of the protocol proposed in [EG02]by Eschenauer and Gligor. They propose to preinstall a subset of keys (called *key share*) of size m from a *key pool* on each device in the following manner:

- The key pool $\mathcal{K}$ contains randomly selected keys from a key space.
- Each device receives a random subset of keys from the pool $\mathcal{K}$
- When two devices wish to establish a secure communication link, they exchange the identifiers of the keys known to them. If they share at least one key from $\mathcal{K}$, they derive a session key from the shared keys (in our protocol they use one of the shared keys: the one which requires least amount of computation to process).

This reduces security problems arising due to device capture and extraction of stored secret keys. By capturing a device, the adversary gets only a small subset of the key pool. Consequently, compromising the whole network is much harder.

The goal of the modification in [CGK09]is to further improve security but without increasing the size of keys subset stored on a device and without increasing communication complexity. The idea of the key levels scheme is that each key K from the key pool $\mathcal{K}$ can appear on many levels: $K_1, K_2, \ldots, K_L$ (where L is a protocol parameter) such that: $K_1 = K$ and $K_i = G(K_{i-1})$ where G is a secure one-way function. This way, the key on higher level can be easily derived from the key on a lower level, but not the other way round. Key installation looks as follows:

- Choose a subset $S_A \subset \mathcal{K}$ of cardinality m at random (m is the size of the *key share*);
- For each $K \in S_A$ choose $l \in \{1, \ldots, l\}$ according to some probability distribution $\mathcal{P}$ independently of other devices and install K_l on the device (in our protocol the level number is chosen uniformly).

In order to establish a secure connection, two devices A and B have to do the following:

- Determine index of the shared key K
- Exchange level numbers i_A and i_B of the key K (assume $i_A < i_B$)
- The node with the smaller level number – here A – derives K_{i_B} from K_{i_A} by applying function G $i_B - i_A$ times.

Note that by capturing a device, the adversary can brake only some fraction of all connections established using keys stored on the device. All connections using these keys on lower levels are secure.

2. The algorithm

2.1 Implementation idea

From the above description it can be inferred, that the situation for running the algorithm is the following. All nodes can communicate via radio in plaintext, which is an underlying connectivity graph of the network. On top of that an induced connected graph of pair-wise secure connections is constructed to meet two goals. First, it must contain the node primarily storing *GK*. Second, it must contain all (or as many as possible) other nodes.

To achieve this goal, we use a bottom-up approach. First, on the connecting graph small cliques of securely connected nodes are formed that grow and connect with one another with time. At the same time, the dissemination of *GK* is achieved by a viral infection mechanism: whenever a node establishes a secure connection with another node in possession of *GK*, this information is propagated. Then, the newly infected node passes *GK* to all members of its secure clique.

2.2 State diagrams

The algorithm can be viewed to have two behavior patterns: that of a "seeker", wherein the node actively seeks for secure connections, and that of a "server", where it responds to actions by other nodes. For a certain period of time, they are executed in parallel, but after a node receives the group key it becomes only a server. This obviously reduces the amount of messages sent by nodes, as broadcasting the own key share information makes no sense when the goal (i.e., the Group Key reception) has been reached. This situation is depicted in Fig. 1.

Seeker. A newly switched on node enters this state. The *"have GK?"* condition checks if the node is in possession of the *GK*. The first check is executed immediately, so that the group leader does not enter the neighborhood discovery phase, but rather transits directly to the second, serving phase. Next, *"have neighbors?"* tests the neighborhood for any active nodes within radio range. If it fails, the node executes `sleep` for a certain period of time waiting for more favorable conditions.

The next loop is entered only when it makes sense to broadcast any key information. At this time also, the second behavior pattern is invoked, but it will be described in detail in the next paragraph. Using a controlled flood mechanism, the *key share* information is broadcast with increasing `ttl` values. The `wait` operation is

introduced to allow any ACK packets to arrive from potentially distant stations. The node stops being a "seeker" the moment it receives a GKmsg message containing a valid Group Key, which triggers "Y" on *"have GK?"*.

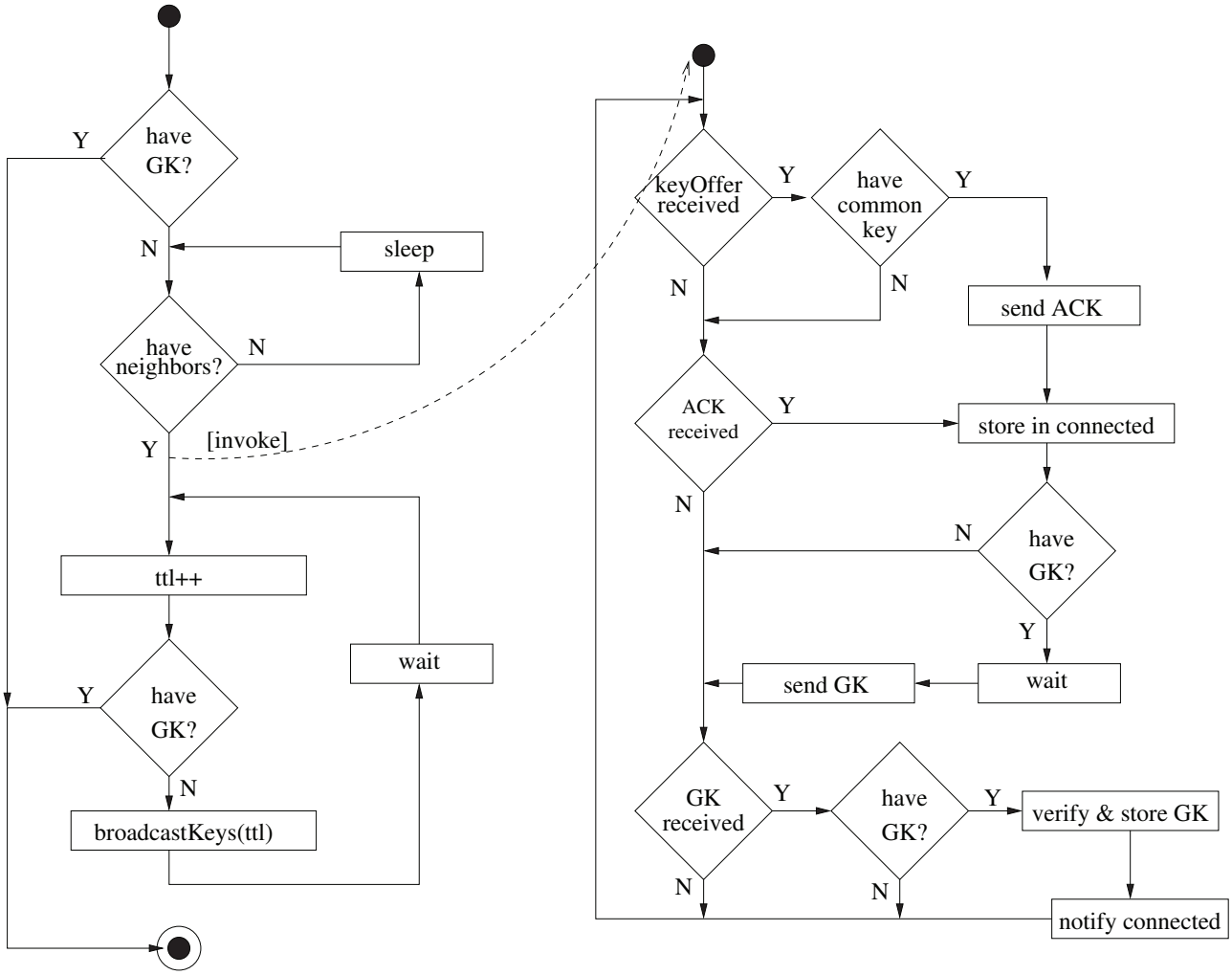

Figure 1: Activity diagram for "seeker" (left) and "server" (right) phases of the algorithm.

Server. When neighborhood discovery succeeds, the node starts listening for incoming packets with key share information (keyOffer) of its peers, ACK to its own information and also packets containing *GK*. The right-hand side of Fig. 1 depicts possible operations and their consequence.

The condition *"have common key"* succeeds if the received information contains a key that matches any of the keys in local *key share*. In such a case an ACK is sent back and the originator is stored in the *connected* list. Additionally, if the node is in possession of GK, the ACK is followed by GKmsg. The wait operation was introduced here to allow the receiving node time for processing the ACK message. The same procedure applies when an ACK is received, save the common key check: in this case the common key is determined by the sender. Lastly, on first receipt of GKmsg, a verification of validity is performed and the group key is stored. After that, all nodes in *connected* list are sent the GKmsg as well as all registered callbacks from higher level modules are triggered. The "server" operation does not finish; a listening node assures that newly joining nodes will be contacted and receive ACK and GKmsg messages, if adequate.

2.3 Stabilization remarks

In the initialization phase of the group key establishment protocol, the *key share* S_A of every station is filled with the set of pair-wise keys drawn uniformly at random from the *key pool* $\mathcal{K}$. For each key $K \in S_A$, the additional modification is applied due to a randomly drawn *key level*. After the sensor network has initialized in that way, one can build the graph of secure connections, i.e., the graph with sensor as a nodes and with edges between any two nodes sharing the same key. Obviously, the connectivity in the secure connection graph and reachability of nodes having the group key depends heavily on the size of *key shares* and the size of *key pool*.

Notice that the *key level* modification of Eschenauer and Gligor's [EG02]work does not have any impact on the connectivity between any two nodes: if only they have the common pair-wise key, nodes can agree its level for communication. This observation allows us to reduce connectivity problem in our secure connections graph to the problem of connectivity in the uniform random intersection graph. This reduction can be made under the assumption that all the nodes are physically connected, i.e., the graph of radio connections between nodes is connected.

The most comprehensive recent work on random intersection graph is by Blackburn and Gerke's [BG09]. Let n be the number of nodes, l the size of *key pool* $\mathcal{K}$ and m – size of *key shares*. Authors of [BG09]showed that there is a tight, asymptotic (w.r. to n) threshold, namely, if $\liminf_{n\to\infty} \frac{m^2 n}{l \log n} > 1$, then asymptotically almost surely intersection graph is connected. It is not when $\liminf_{n\to\infty} \frac{m^2 n}{l \log n} < 1$.

This connection between all parameters of our protocol was crucial when considering the proper pool size (when network size and *key share* were fixed) or *key share* size (when network size and *key pool* were fixed). Our experiments show that the Blackburn and Gerke's asymptotic threshold can be a good point to start when determining the parameters of real networks, i.e., when n is relatively small. For fixed network size and *key pool* size, we usually used *key share* size enlarged by a few keys compared to Blackburn and Gerke's threshold. Below, we present the sample of *GKE* parameters that we used to ensure the connectivity of the secure connections graph. We need to keep in mind, that the *key pool* size and *key share* size not only influences the security of the network, but also the performance of the group key establishment.

network size	key pool size	key share size
20	200	8
50	500	12
100	1000	15

3. Implementation details

3.1 Packet frames

The GKE module uses one packet frame layout for all messages exchanged during its operation. A general view is presented in Fig. 2.

Figure 2: Packet frame layout for GKE protocol.

The following table describes sizes and contents of the fields within the frame. Note that all types are **typedef**'d relative to `OsModel_P::Radio` WiseLib's namespace.

field	type	description
TYPE	msg_id_t	message type
DEST	radio_id_t	destination address
SRC	radio_id_t	source address
CLSTR	radio_id_t	cluster id, for which the message is valid
TTL	uint8_t	ttl value, if TYPE != TTY_MESSAGE then undefined
MSG_ID	msg_id_t	unique message id, used by TTL routines
LEN	size_t	length of payload
PAYLOAD	block_data_t	message payload

The field TYPE can take any of the following values:

NEIGHBORHOOD_SEEK: 33 a ping message for neighborhood discovery phase;

NEIGHBORHOOD_ACK: 34 a response to ping message in neighborhood discovery phase;

KEY_ACK: 31 message is an ACK for key share info; the PAYLOAD part then contains a serialized structure containing an identifier of the pair-wise key to be used between the two nodes and its common level;

GROUP_KEY: 32 message contains the Group Key encrypted with the pair-wise key established between the sender and the receiver previously;

TTL_MESSAGE: 35 this message is generated by controlled flood routines; it is to be broadcasted if a message's TTL field contains a value greater than 1. This value is decremented and the remaining message is broadcasted unchanged.

In the GKE implementation, the TTL_MESSAGE type is used to indicate that the message contains key share information (`keyOffer`). Whenever controlled flood routines receive such a message with TTL value equal to 1, a callback function is invoked. The PAYLOAD field then contains an array of pairs/**structs** (keyID, keyLevel), with the exception that the first pair is of the form (v,0), where v is a simple verification code: a result of an XOR operation on all keyID values contained in the message.

3.2 Timing – events in the network

Typical runs of GKE are presented in Fig. 3. It depicts is a situation where all nodes wake up at the same time and perform the algorithm without stop. The graphs show a total count of sent/received messages of three different types: neighborhood discovery, `keyOffer` and `GKmsg`.

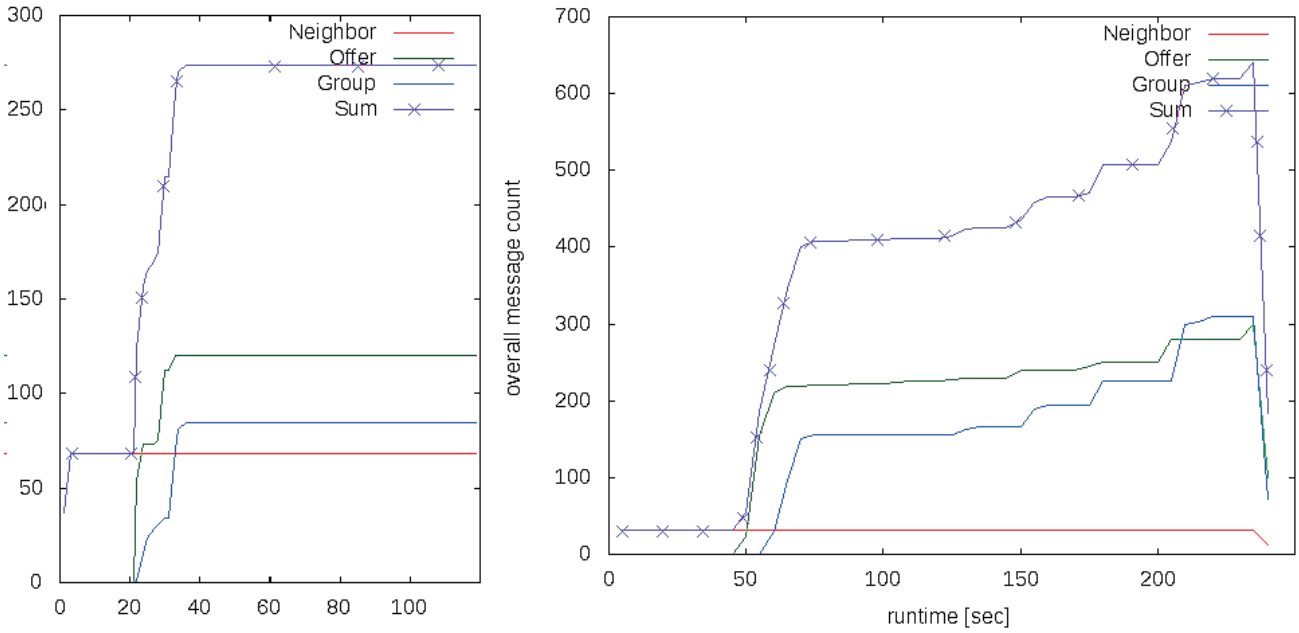

Figure 3: Message count for runs of GKE in simulation (left) and real-life setting (right).

The left figure presents a simulated, strongly connected network, while the right one is a real-life run of the algorithm. Despite this difference both diagrams share similar features. Firstly, neighborhood discovery messages are emitted at time 0 and their number does not rise later. After approximately 45 seconds, which is the time GKE waits after its `init()` is invoked (this is a precaution to allow routing algorithm to stabilize), `keyOffer` messages start circulating. Soon after that some nodes have established pair-wise links with nodes in

possession of Group Key, therefore GKmsg's appear (around 65 second). Later, the TTL mechanism adjusts its timeout before next batch of keyOffer messages is sent out – one can see that steps on the line representing this message type occur at increasing intervals. Simultanously, GKmsg's are issued as a consequence of newly created pair-wise links.

It should be noted, that from graphs of this type one can infer information about network's topology. The steepness of the curves determines connectivity between nodes. Also, displacement between consecutive batches of GKmsg can indicate that the network is physically split into clusters of nodes connected by a narrow one-node link.

3.3 Key generation

In the original protocol, the keys and their levels in *key share*s loaded to each node are calculated in advance. This ensures that a node has no information on other nodes' *key share*s, and, consequently, the whole *key pool*. However, for simplicity, in our implementation the nodes generate keys for themselves at boot time.

This is done as follows:

- Each node has the same random seed;
- using the seed all nodes can generate the same key pool;
- using their private seeds nodes they choose key indexes and key levels;
- nodes calculate their key shares and remember appropriate keys from the pool and raising it to given levels;
- other keys are not stored.

Of course this is not a secure solution. Adversaries could capture a node and regenerate the entire key pool. This is used only in order to simplify the procedure of key pre-installation.

3.4 Module interconnections

The Wiselib library is designed to be as generic as possible. This not only means that the code written in Wiselib runs on different hardware platforms, but also that modules implementing the same concept can be easily exchanged. The GKE module follows this loose coupling paradigm and therefore, to some extent, the underlying modules can be substituted for those implementing the same concept.

The Group key establishment module (Keylevels) uses external modules to accomplish two main goals: to acquire information about the environment it works in and to perform the group key establishment algorithm.

For the recognition of the environment, GKE must be supplied with modules implementing the neighborhood discovery and clustering concepts. Our GKE implementation also makes use of modules implementing routing and controlled flooding concepts and the module of symmetric cryptography.

3.4.1 Neighborhood discovery. The battery saving was the main purpose of using neighborhood discovery in the process of group key establishment. Namely, from the point of view of a node, the start of the "seeker" algorithm is senseless without having any other sensor within the communication range. To address this problem, we developed a simple neighbor discovery mechanism. The final version of GKE module, uses the FRONTS neighbor discovery module Echo.

3.4.2 Clustering. The GKE module requires that the supplied implementation of the Clustering concept must at least provide following methods:

- is_leader(),
- cluster_id().

The cluster_id() method is used to narrow the scope of work for GKE to only a part of the network. As the search for the common group key makes sense only when at least one node possesses (or is able to generate) *GK* we needed to ensure that such a node appears in the group of nodes performing GKE. Even though, in general,

being a leader does not necessarily mean having a group key, we utilized `is_leader()` function for determining the initial node in possession of *GK*.

In most of cases, the GKE would be initialized with an actual implementation of `Clustering`, e.g., its FRONTS implementation concept. In order to simplify some experiments, we also provided a `FakeClustering` implementation of the `Clustering` concept. The `FakeClustering` helps treating the entire network as a big cluster with the leader set a priori.

3.4.3 Routing. In `Wiselib`, any routing implementation implements the `Radio` concept. There is only one additional requirement on the routing algorithm supplied to the GKE module: it should provide a bidirectional routing. Our implementation was extensively tested with FRONTS `End-to-end communication` implementation and `Dsdv` routing implementation.

3.4.4 Controlled flood. To perform group key establishment algorithm, we needed to implement controlled flooding (TTL broadcast) in `Wiselib`. Our controlled flooding module `TTLFlooding` implements the `Radio` concept. The only task it performs is to broadcast message to all nodes within some n-hop range for this, the static set of already received messages is used. Additional important methods available in `TTLFlooding` class are:

`set_ttl(uint8_t new_ttl)` – for setting up the `TTL` level of message about to send,

`set_cluster(node_id_t cluster)` – for narrowing the `TTLFlooding` to work in a specific cluster.

3.4.5 Cryptography. In order to utilize symmetric encryption in the process of key establishment, a slightly modified implementation of `AES` from the `Wiselib` was used. The main reason for using symmetric encryption is secure delivery of *GK* to all nodes using their pair-wise keys. Our modification addressed the problem of setting up the key for encryption in run-time.

3.4.6 Group key establishment callback. The group key establishment module `Keylevels` itself implements the `Radio` concept. The `recv` callback of `Keylevels` is triggered when a sensor receives `GKmsg`, which can be properly decrypted. The value of *GK* is passed as payload to the callback routine.