SECURE KEY DISTRIBUTION IN WIRELESS SENSOR NETWORKS (WSNS)

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Christoph Knecht
2010

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Contents

Abstract

As wireless sensor networks (WSNs) get larger and become more popular, the confidentiality of data being sent is crucial. In order to prevent eavesdropping on the transmitted information, suitable key-exchange schemes, that comply with the challenging requirements in wireless sensor networks, need to be implemented.

In this paper we implement two different key-exchange mechanisms: The first is based on the well-known key-exchange scheme by Diffie and Hellman, the second is a key-exchange scheme developed for WSNs by Eschenauer and Gligor. Both algorithms were developed for Wiselib, a library of generic algorithms for WSNs. We evaluate these algorithms thoroughly in terms of performance in small and large scaled WSNs. Performance measurements include the amount of traffic caused by the key-establishment and the time it takes until every node has established secure connections to all nodes within transmission range. Both implementations consider the limited resources available on wireless sensor nodes and try to minimize storage and message overhead.

We further show that the key-exchange scheme by Eschenauer and Gligor performs much better than the Diffie-Hellman variant concerning data space used and amount of data being sent, and is thus much more suited for the purpose of securing WSNs.

List of Figures

List of Tables

Acknowledgment

First of all I would like thank all of my three supervisors, Carlos Anastasiades, Philipp Hurni and Zhongliang Zhao for their continuous and kind support. Thank you for giving me food for thought, taking your time to discuss problems and for helping me writing this thesis. Thanks go as well to:

Prof. Dr. Torsten Braun for the opportunity to write a thesis suitable to my interests.

Sebastian Barthlomé, Ulrich Bürgi and Benjamin Nyffenegger for drinking coffee with me, discussing, advising and fooling around.

My girlfriend Sara, my brother Matthias and my parents for supporting me during the whole time of this work.

Chapter 1

Introduction

1.1 Motivation

In the last few years, wireless sensor networks (WSNs) have become larger and much more popular. Today's applications of wireless sensor nodes range from environmental monitoring, traffic control, industrial regulation to electronic homes and also military usage. All of these applications include data being sent that is confidential and / or of personal nature. Eavesdropping on the transmitted information must to be prevented. As wireless technologies are not limited by walls, keeping sensitive data secure is even more important, as traffic can be monitored from anyone and anywhere within transmission range. To fulfil this need for security, node-to-node encryption of the transmitted data is vital. Due to the limited hardware possibilities it is a challenging task to provide suitable but still secure key-exchange solutions. As sensor node networks can get very large and may be planted in hostile areas as well, one has to consider that parts of the network can be detected and / or captured. Even if physical access to one or several nodes gets possible, the network must not be compromised. The Wiselib Project [1, 2] (see Section 2.1 for details) offers generic algorithms for WSNs. However, no security mechanisms exist yet in Wiselib. Therefore, it is important that new developed algorithms can rely on a secure and efficient key-exchange scheme to ensure the safety of their transmitted data.

A rather naïve approach to secure WSNs consists of using a single key for the whole network. Unfortunately in terms of security, this solution is very weak. If one node gets captured an attacker would be able to compromise the whole network immediately. A solution to this problem is given by a key-exchange scheme, where two neighbouring nodes establish a key between each other.

Despite the need of a suitable key-exchange mechanism the implementation should consider the limited resources on small sensor nodes. Several kilobytes of RAM and a CPU with 20 MHz are not really capable to process state of the art asymmetric cryptography. Last but not least energy efficiency is a major concern in WSNs - therefore the amount of messages being sent and processed should be kept low.

1.2 Tasks & Problem Formulation

The goal of this thesis is to implement two key-exchange mechanisms (Diffie-Hellman [3], Eschenauer & Gligor [4]) in the context of the generic algorithm library Wiselib. The thesis thoroughly evaluates these two key distribution schemes in the network simulator Shawn [5] with respect to:

- additional data space used
- traffic statistics
- time to establish all possible links in a network

1.3 Contributions

Our contribution consists of implementing key-exchange schemes as generic Wiselib-algorithms. Two schemes were implemented, the one proposed by Diffie-Hellman and the one by Eschenauer & Gligor. The original Diffie-Hellman scheme was slightly modified due to the resource constraints on the sensor nodes. This led to the adapted scheme further referred to as *Diffie-Hellman Lite*. The Eschenauer & Gligor scheme - designed for wireless sensor networks - did not have to be adjusted. We developed a small helper toolchain for the Eschenauer & Gligor algorithm: All of them are explained in detail in Chapter 3.

1.4 Outline

In Chapter 2, we present the details of the original key-exchange schemes. Chapter 3 then describes our implementation of the two key-exchange schemes, lists and justifies the differences to the original schemes. In Chapter 4 we introduce the different simulation parameters used for evaluation. Besides that, we analyze and discuss the simulation results. Lastly, Chapter 5 summarizes and discusses the results given in the previous chapters.

Chapter 2

Related Work

2.1 Wiselib

When developing algorithms for wireless sensor nodes, one is challenged with various different platforms and hardware specifications. Porting of existing algorithms to other platforms is time consuming and annoying. In order to find a solution to this problem, the generic algorithm library Wiselib was developed. The goal of Wiselib is to provide generic, template-based algorithms that can be used on any platform. Wiselib algorithms access and interact with an Operating-System Abstraction Layer (OSA) and can be compiled for a specific target platform with platform-specific Makefiles. Besides several different hardware platforms, a network simulator called Shawn (see next Section) is supported as well.

2.2 Shawn

The network simulator Shawn is a tool capable of simulating large sensor node networks containing several thousand nodes. Complete topological control is available and visualization and logging functionality maximizes the possibilities to evaluate the simulation results. Different transmissions models can be chosen; besides writing an own model, CSMA and a reliable model are best suited for most of the testing purposes. These transmission models can be combined with various disturbance simulations, such as a non-radial transmitting range (limited by environmental effects) or the random dropping of a specified amount of packets. All these parameters can be supplied in a configuration file. The simulation itself is round-based, where every round simulates one second of (simulated) time. If the transmission model is set to reliable and the parameter “immediate” is set to false, sending and receiving of messages only occur once in a round for every node. A node can be in three different states, either active, sleeping or inactive. The state of the node can be controlled from within a processor. A processor is an instance of an algorithm running on a node - several processors can run at the same time. A simulation finishes when either the maximum simulation round limit is hit or when all simulated nodes set their state to inactive.

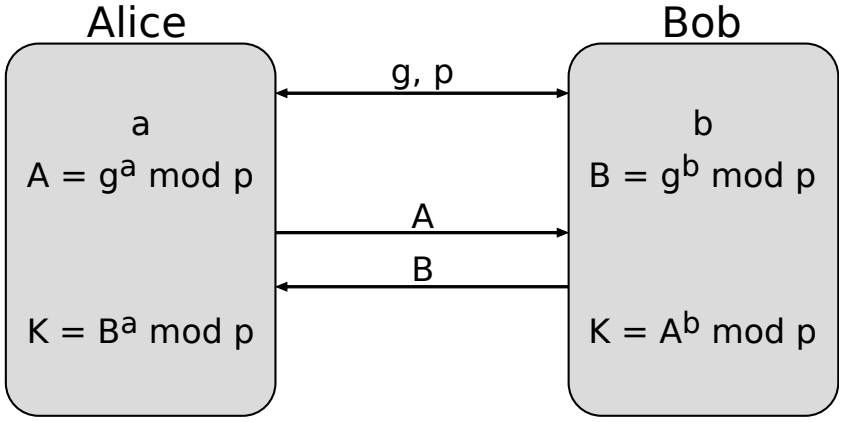

Figure 2.1: *Diffie-Hellman* - Key-Exchange Scheme

2.3 Diffie-Hellman

The proposed key-establishment solution consisted of the following few steps (as an example for two individuals - Alice and Bob - requiring a secure key to communicate - see Figure 2.1 for illustration):

- Alice and Bob agree on a prime number p and on a prime root g such as $2 \leq g \leq p-2$
- Alice and Bob now each choose a random number a (respectively b) satisfying $1 \leq a \leq p-2$. The number a is used as the private secret.
- Alice and Bob both derive their public secret A (respectively B) by computing $A = g^a \bmod p$
- The public secrets can now be transmitted over an insecure channel
- To calculate the final key, Alice computes $K = B^a \bmod p$
- Both parties derive the same key because

  Alice: $K = B^a \bmod p = (g^b \bmod p)^a \bmod p = g^{ba} \bmod p = g^{ab} \bmod p$
  Bob: $K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p$

Even if an adversary would capture all the transmitted data, it would be of no use to him. He would have to compute the discrete logarithm of the public secrets to be able to derive the private secret in order to compromise the whole system.

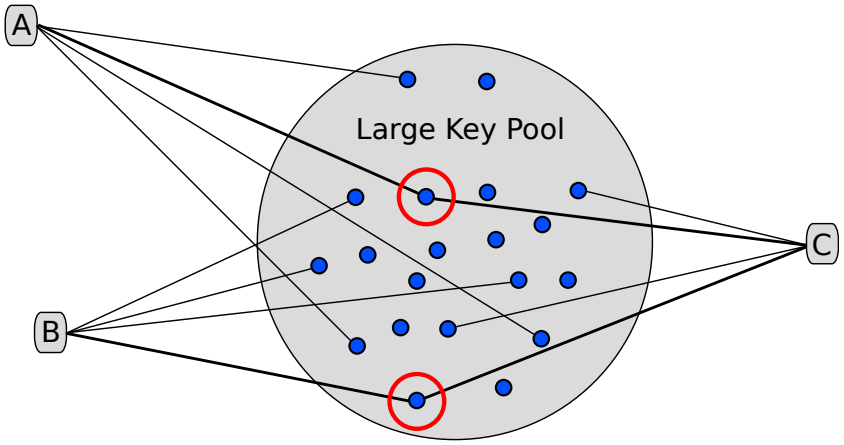

Figure 2.2: *Eschenauer & Gligor* - Random Key Drawing. Three nodes pick random keys out of a big pool. Common keys are red circled. Node A and B do not share a common key yet. They will have to negotiate on a key in the third initialization phase using a secure path over node C.

2.4 Eschenauer & Gligor

Eschenauer and Gligor's key-exchange mechanism is based on a huge pool of keys where keys get drawn out randomly and assigned to every node. This solution is a tradeoff between using a single key for the whole network and having a key agreement based on public cryptography for every link. The proposed scheme contains three major initialization phases:

- Randomly draw k keys out of a huge pool of keys. Each key has a corresponding key identifier. The keys and their identifiers are hardcoded on the nodes prior to deployment. The series of hardcoded keys is further referred to as key ring.
- During boot phase the nodes broadcast a list of all their identifiers. A neighbouring node (node within transmission range) checks whether it shares a common identifier. If true the common key will be used for future communication.
- Due to the probabilistic nature of the key drawing it is possible that two neighbouring nodes do not share a common key. Therefore, in the third phase two nodes establish an encrypted path over a common neighbour (or a neighbour of a neighbour if necessary) and then agree on an unused key of either one of the nodes to use in future communication.

Revocation of keys is possible (e.g. as a result of compromised nodes). As the keys are drawn randomly, existing key agreements between nodes may get unusable and new keys have to be agreed on. From now on, unusable key agreements between nodes will be referred to as "damaged links".

When two nodes each pick k keys out of a big pool containing P keys, the probability p of sharing at least one common key is given by the simplified equation 2.1 (from [4], Section 3.1).

$$p = 1 - \frac{(1 - \frac{k}{P})^{2(P-k+\frac{1}{2})}}{(1 - \frac{2k}{P})^{(P-2k+\frac{1}{2})}} \tag{2.1}$$

When having a low number of neighbours, we need to have a probability of $p = 1.0$ to share a common key with a neighbour in order to connect the network at all times. This necessary probability however decreases as the number of neighbours increases. Two other equations are needed in order to be able to compute this desired probability (from [4], Section 3.1).

$$c = -\ln(-\ln(P_c)) \tag{2.2}$$

$$p' = \left(\frac{\ln(n)}{n} + \frac{c}{n}\right)\frac{(n-1)}{(n'-1)} \tag{2.3}$$

Equation 2.2 defines the constant c related to P_c, i.e. the probability that the whole network can be connected. Equation 2.3 finally computes the necessary probability of two neighbours sharing a common key in dependency of the constant c, the total number of nodes in the network n and the average number of neighbours n'. As an example we could choose $P_c = 0.999$, a network of total 900 nodes ($n = 900$) and an average of 20 neighbours ($n' = 20$) per node. This would result in a necessary probability of

$$p' = 0.72$$

for sharing a common key with a neighbour. If we would want to use a key ring size of 100 keys on every node, we now could calculate with help of equation 2.1 the maximum size of the key pool that fits the requirements (i.e. $p = p'$). This calculation shows that a pool size of at maximum

$$P = 7900$$

keys is possible and the network should be connected.

Chapter 3

Design & Implementation

In this Chapter we present our implementations of the two key-exchange schemes referred to in Chapter 2. Obstacles that occurred during implementation are analyzed in detail and example message flows are presented.

3.1 Diffie-Hellman

In this Section we are going to introduce our own derivation of the original Diffie-Hellman key-exchange scheme called *Diffie-Hellman Lite* (*DH Lite*). We are going to point out the obstacles that led to the modified scheme, and an example message flow will be presented in the end.

3.1.1 Requirements

According to the original design presented in Section 2.3 there are several requirements to be fulfilled in order to successfully establish a key using the Diffie-Hellman key-exchange scheme. Those requirements are:

- A large prime number p
- A prime root g such that $2 \leq g \leq p-2$

A prime root is computationally intensive to calculate for a given prime number p as the factorization of $p-1$ is necessary to be know. Sophie-Germain prime numbers have the unique property that $2p+1$ is a prime number as well. This allows very fast computations of prime roots for prime numbers such as $p' = 2p+1$ (where p is a Sophie-Germain prime) as the factorization of $p'-1$ is trivial.

3.1.2 Diffie-Hellman Lite

In order to provide a reasonable level of security, the prime number used to derive the key from has to be very large. As mentioned in the paper from Raymond and Stiglic [6] it should be at least 2048 Bit long to ensure a moderate level of security. Because of the very strict resource constraints on today's customary sensor nodes, it is impossible to generate prime numbers of this

length in a reasonable amount of time. Therefore, in a first attempt, we considered to use a list of fixed prime number / generator pairs, similar as it is done in SSH. Relating the implementation to sensor nodes however, this solution proved to be unusable as well because it induced storage related problems and message overhead. An example calculation:

- Let us assume there is a list of four different prime number / generator pairs. 4096 Bit prime numbers are used.
- After agreeing on a prime number to use, every node has to derive a public and private secret related to this prime number.
- Therefore not only four prime numbers have to be stored, but four different public and private secrets as well. This means every node needs to be able to store $12 \cdot 4096$ Bits.
- Additional space is needed to keep track what prime number was used for which link.
- Increased traffic: In order to establish connections to neighbours, the public secrets have to be exchanged. In the worst case, four different public secrets need to be sent to the neighbours, since not all nodes may have agreed on the same prime numbers to use.

In order to prevent this storage and message overhead, we decided to implement a modified version of the original Diffie-Hellman key-exchange scheme called *Diffie-Hellman Lite*. The only difference of *Diffie-Hellman Lite* compared to the original scheme is that we rely on one fixed prime number / generator pair only. The paper of Raymond and Stiglic mentions that the use of Sophie-Germain prime numbers is a secure solution if there is need for fixed prime numbers as the resulting (derived) secrets get very large as well. In our implementation we decided on using a 4096 Bit long safe prime number from [7]. Safe primes have the form $p' = 2p + 1$ where p is a Sophie-Germain prime. In contrast to the example calculation given above, we now only have to store one public and private secret per node and keeping track of the prime numbers can be dropped. Last but not least if we want to establish connections with all neighbours, we simply have to broadcast our public secret once and all connections can be established. This results in the most efficient variant of a Diffie-Hellman like key-exchange scheme in respect to the amount of data being transmitted.

As mentioned earlier, there is also the need of a fixed prime root. To test whether a candidate number is an actual prime root (also called secure generator) the following condition has to hold:

- For each prime factor f of the number $p'-1$ it has to be evaluated whether $n^{((p'-1)/f} \mod p' \neq 1$ is true. The variable n represents the prime root candidate, and p' is the used prime number.

Because we decided to use a safe prime, this computation is really easy: The only possible prime factors of $p' - 1$ are 2 and p (because $p' = 2p + 1$). We decided to use the small prime root $g = 2$. The rest of the *Diffie-Hellman Lite* scheme follows the original implementation.

Although a big computational part is skipped in the *Diffie-Hellman Lite* scheme, it is still very improbable that it actually runs on real sensor nodes. The arithmetic to derive the private and public key still needs lots of computational effort and the third party libraries (see Section

3.1.3 we need are not available yet for today's widely used sensor nodes compilertoolchains. Using one fixed prime number / generator pair may additionally impose some security weakness as it is known in advance what prime number will be used.

3.1.3 External Libraries

The arithmetic with very large numbers such as 4096 Bit long prime numbers cannot be done with the standard `math.h` implementation. The whole calculation of the public and private key is therefore achieved using the GNU Multiple Precision Arithmetic Library [8].

3.1.4 Key-Exchange Procedure

Since we decided to use a fixed prime number / generator pair, the first step of the original scheme (agreeing on what prime number / generator to use) can be neglected. Figure 3.1 depicts the message flow for one node. The key-exchange procedure looks as follows:

- Every node periodically broadcasts a `HELLO` message. The `HELLO` message contains a list of known neighbours in the payload.
- If a `HELLO` message is received, every receiving node checks whether its ID is already in the list of the known neighbours. If true, there is no further action needed. If false, it means that the sending node needs to establish a key to the receiving node. The receiving node broadcasts its public secret. Due to the maximum payload limitation of 2048 Bit in Shawn, the public secret is split up in four `SECRET_EXCHANGE` messages; each message containing 1024 Bit of the public secret. As the payload is not the only information stored in the message, we agreed on using the next smaller size possible that still allows easy splitting of the used secret. The `HELLO` message solution ensures that no unnecessary transmissions of the public secret occurs but dynamic network changes, such as addition of nodes, still remains possible.
- After a node has received all four `SECRET_EXCHANGE` messages, it is able to reassemble the public secret and compute the key. The resulting key is then fed to the hash function SHA256 [9], which we ported to the Wiselib. The resulting hash is split in half and the first half is then stored as the final key for this specific link since we agreed that 128 Bit keys are secure enough for sensor networks (tradeoff between security and resource constraints).
- In the following, when an encrypted message has to be sent, the data is encrypted using the AES128 [10] module provided in Wiselib.

To be able to send encrypted messages through the whole network, we rely on a solid routing mechanism. As Figure 3.2 shows, our algorithm takes place between the application and routing layer. These layers model the internal structure of the Wiselib. If messages have to be encrypted they are passed on to the routing layer, as they might need to travel to destinations beyond the neighbourhood. The initial key-exchange mechanism however needs broadcasting of messages

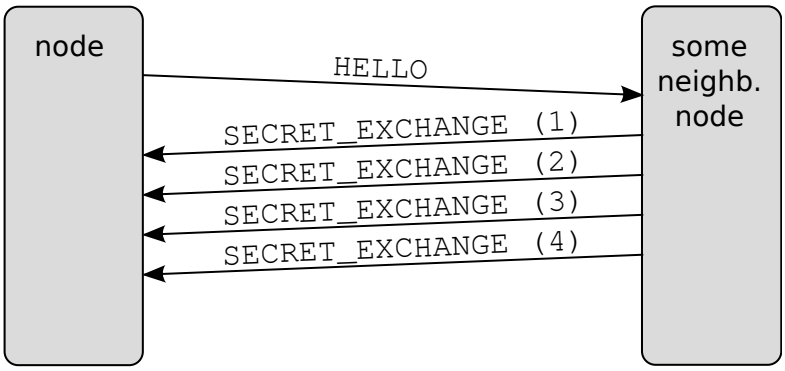

Figure 3.1: *Diffie-Hellman Lite* - Message Scheme. This scheme shows the case where a node is not yet in the known neighbour list sent with the `HELLO` message and is therefore broadcasting it's public secret.

to the neighbours. Therefore it directly accesses the radio interface in order to prevent message overhead.

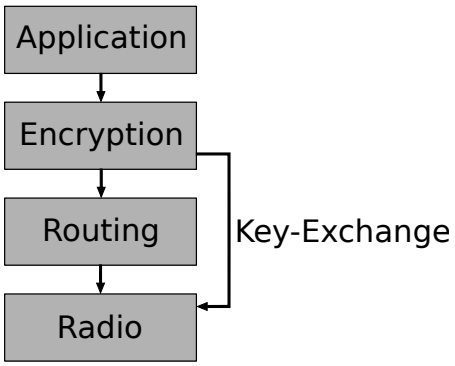

Figure 3.2: Encryption - Position within Wiselib. Encrypted messages are sent using the routing functionality, the initial key-exchange mechanism however directly accesses the radio interface in order to minimize message overhead.

3.2 Eschenauer & Gligor

As mentioned in Section 1.3, we developed a small helper toolchain for the *Eschenauer & Gligor* (*EG*) algorithm. In this Section, we introduce the different tools and their purpose. Afterwards, we give a full examination of the message flow used for the key-establishment, covering all the special cases that can be handled by our implementation.

3.2.1 Requirements

As mentioned in Section 2.4, the first step of the key-exchange scheme is to draw random keys out of a large key pool and then store them on the nodes prior to deployment. In our implementation we chose a length of two bytes for the identifier and 16 bytes for the key identical to the contiki ids. Therefore, it is possible to use approximately 65000 different keys of 128 Bit length. In order to implement the Eschenauer & Gligor key-exchange scheme, the following parameters need to be specified:

- Maximum size of the key ring on every node, i.e. how many keys will be stored on a node prior to deployment.
- Network size (number of nodes in total)
- Mean number of neighbouring nodes
- Key pool size (number of keys in the pool)

3.2.2 Helper Toolchain

eg_params

Eschenauer & Gligor show that the probability for a node to have a common key to a neighbour decreases as the neighbourhood and the whole network grows. The tool eg_params - as the name suggests - calculates the maximum pool size possible in respect to the parameters key ring size, amount of neighbours, total amount of nodes and the probability P_c of the whole network being connected (see 2.4). This tool basically provides a handy interface to the calculations as they were done at the end of Section 2.4.

eg_params2

This tool provides the same functionality as eg_params but here the pool size can be specified as well. It is mainly used to see how different parameters affect the key probabilities.

potgen

The tool potgen is used to generate the key pool. In the following, the key pool will be referred to as "pot" as well. The tool potgen takes the desired pool size - and optionally - the output filename as arguments. In order to calculate the random keys that build the pool, a random number is chosen (using the std-lib call `rand()`). The random number is afterwards fed to SHA256 and the resulting hash is split up in two 128 Bit long keys.

ringpatcher

Every node has to get its keys prior to deployment. When using the Shawn simulator, patching of the keys takes place at runtime, but if real nodes are used, the patching of the keys has to be done prior to flashing the binary. The ringpatcher tool is used to write the keys and key-identifiers into the compiled binary. The arguments the ringpatcher takes consist of the key ring size, the amount of neighbours, a random seed, the pot file and (of course) the binary file to be patched.

The ringpatcher first searches for the offsets of the key- and identifier-list in the binary. In order to better understand why this is done, we briefly explain the two most crucial lines of code from `eschenauer_gligor_algorithm.h` (lines 36 & 37):

```
static const uint8_t
  keys[ RING_SIZE * 16 ]
      = "aa22bbbbcccc";
uint8_t
  identifiers[ RING_SIZE * 2 + NEIGHBOUR_COUNT * 2 ]
      = "aa11bbbbcccc";
```

These two lines initialize both an `uint8_t` array that will be used later on to store the keys and identifiers in it. The reason the keys array is declared as `static const` is that it gets allocated in the `.text` segment (part of an object file that contains executable instructions) and will

therefore be stored on the ROM of the nodes where more storage is available than on the RAM. The NEIGHBOUR_COUNT definition specifies the amount of neighbouring nodes. Because one node has to agree on a maximum of $n-1$ new keys (n being the same as NEIGHBOUR_COUNT), the identifiers array needs to provide enough space to store the new key-identifiers. Since the keys array is declared static, no keys can be modified or added. Therefore, an additional array is needed to store the new keys that are agreed on in the third phase of the key-exchange scheme. Line 38 of eschenauer_gligor_algorithm.h initializes this array:

```
uint8_t
  backup_keys[ NEIGHBOUR_COUNT * 16 ]
      = "aa00bbbbcccc";
```

After the size of these arrays is declared, initial identifier strings such as aa22bbbbcccc are assigned to them. These strings serve two purposes: The first is to ensure that the arrays actually get allocated (if there would be no assigned value, the compiler would delete the arrays during optimization). The second purpose is to help the ringpatcher find the starting point of the arrays.

As mentioned before, the ringpatcher searches for the starting entries of the keys and identifier array. After they were successfully found, the ringpatcher initializes the random number generator with the given seed. The patcher then draws random keys out of the pool, sorts them in ascending order according to their identifiers and writes them into the binary. The key identifiers are the offsets at which the keys can be found in the pool. This means that the identifier 12 would belong to the key at position pool[12 * 16] (as each key has a length of 16 Bytes). It is important that the identifier list is in order; otherwise - if two nodes had two or more common identifiers but not in the same order - they would not agree on the same key to use while processing the lists:

- Two nodes share two common keys (identifiers 9 and 10)
- Node A has its identifiers in the order 9, 10
- Node B has its identifiers in the order 10, 9
- Both nodes sequentially take an identifier in the foreign list and check whether the own list contains this identifier
- Node A takes identifier 10, parses the foreign list and assumes that 10 is the common key
- Node B takes identifier 9, parses the foreign list and assumes that 9 is the common key

3.2.3 Key-Exchange Procedure

The key-exchange procedure follows the original mechanism presented in Section 2.4:

- Node A periodically broadcasts an EXCHANGE_IDENTIFIERS message. As payload the EXCHANGE_IDENTIFIERS message contains a list of key identifiers available on that node.

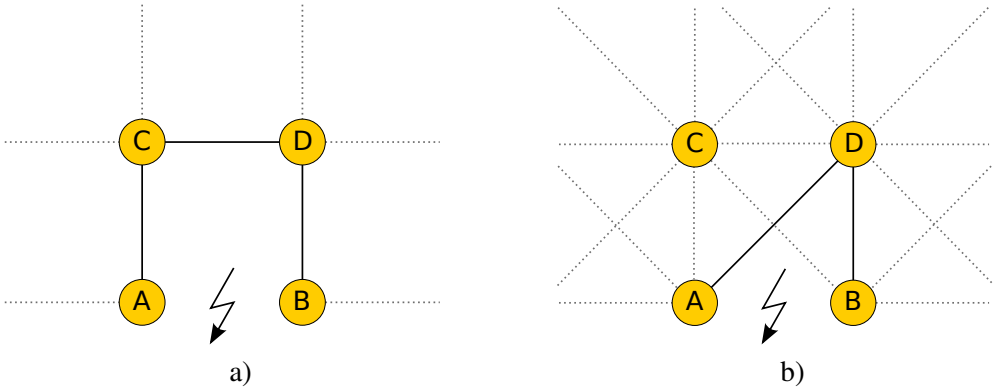

Figure 3.3: *Eschenauer & Gligor* - Path Establishing Phase. Node A and B do not share a common key yet. They will have to agree on one using a secure channel. Due to the square-lattice type of connectivity (transmission range 1.0), Figure a) only allows key proposals over two intermediate nodes. In Figure b) however, communication via one common neighbour is possible as the transmission range is extended to 1.5.

- If a node B receives the `EXCHANGE_IDENTIFIERS` message, it compares its own list for each of the foreign identifiers. If a common identifier is found, the list processing is cancelled and node B stores the identifier and the relating node in its neighbour list. For future communication, the key belonging to the identifier will be used for encryption.
- If no common identifier can be found, the third phase of the Eschenauer & Gligor key-exchange scheme begins. Figure 3.3 depicts the two possible scenarios being handled by our implementation:
  - Node B sends a `NEIGHBOUR_LIST_1` message back to node A. As the name suggests, the message contains a list of neighbours where a link has already successfully been established - in both depictions of Figure 3.3 this would be node D. The `NEIGHBOUR_LIST_1` message has a depth of one; meaning that it contains only direct neighbours.
  - Node A receives the `NEIGHBOUR_LIST_1` message, and checks whether they share a common neighbour. If that's the case - as node D in Figure 3.3 b) is - the first unused key from the keyring is sent back to node B as a encrypted `KEY_PROPOSAL` message over node D using the routing algorithm. After receiving node A's key proposal, node B as well searches for it's own first unused key. Both (the own and foreign) key identifiers are compared and the smaller one is agreed on. That means that eventually node B sends back another encrypted `KEY_PROPOSAL` message (again over node D). This step is necessary in order to prevent issues when both nodes broadcast their identifier lists shortly after each other. A sample message flow that fits this case is given in Figure 3.4.
  - If there is no common neighbour found (as it is the case in Figure 3.3) a), the search of common neighbours goes deeper: Node A broadcasts a `NEIGHBOUR_REQUEST`

message that contains the ID of node B as a payload. Every neighbour of node A that receives the message, checks whether it shares an encrypted link to node A and if true, sends a list of it's own neighbours (a `NEIGHBOUR_LIST_2` message) back to node B using the routing algorithm. In our example node C would send a list of it's neighbours to node B over node D.

  - Node B now checks whether it shares a common neighbour with any of the two hop neighbours of node A, and if true, the encrypted `KEY_PROPOSAL` messages can be exchanged once again over the newly found pathway. As node C and node B share a common neighbour - node D - an encrypted path can be set up to exchange the key proposals with node A.

- If a node finally accepts to use a foreign, proposed key it adds the new identifier-key pair to its own storage - the new identifier to the identifier array and the new key to the backup_keys array. In this case, the position of the new identifier does not matter as it is impossible that two new keys for the same node get added.

- The next time node A broadcasts it's identifier list, the list will contain the newly agreed key. Node B will recognize a common key and an encrypted link is established.

As the complexity of the path establishment grows rapidly when searching for paths over three or more intermediate nodes, we decided to allow two intermediate nodes at maximum. Due to the fact that broadcasting of identifier lists occurs over and over again, it is possible to fix links that would otherwise need paths over three or more intermediate nodes after several rounds; i.e. if you wait long enough, all the intermediate links get fixed and path establishing over two intermediate nodes becomes possible.

Entries in the backup_keys array are marked as free space if the corresponding key identifier is set to `0xffff`. This as wells enables blacklisting of keys if necessary; meaning blacklisted keys have their identifier set to `0xffff` and are thus declared as free space. When a new key needs to be added, the first location with identifier `0xffff` is chosen and both identifier and corresponding key get overwritten.

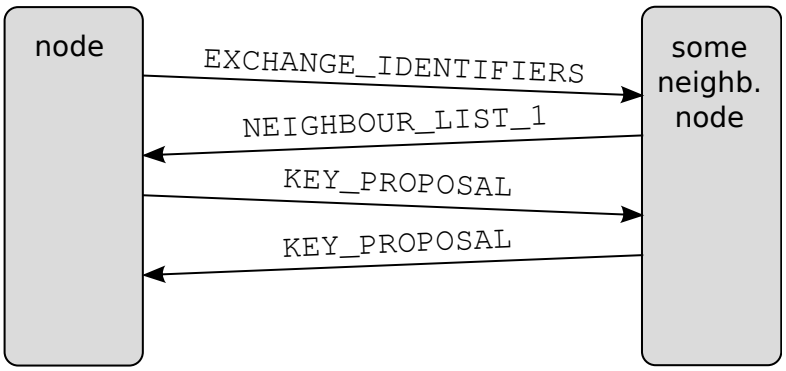

Figure 3.4: *Eschenauer & Gligor* - Message Flow Example. This Figure shows the case where two nodes do not share a common key in the initial key set. However, they share a common neighbour, which can be used to communicate over. Therefore, a `KEY_PROPOSAL` message is sent via the common neighbour. The receiving node has a more suitable unassigned key (lower identifier than the one proposed) and therefore sends a new `KEY_PROPOSAL` message back.

Chapter 4

Simulation & Evaluation

In this Chapter we introduce the different simulation configurations that were used to evaluate our key-exchange schemes. A comparison in terms of performance is done, unusual effects are explained and examined closely.

4.1 Simulation Setup

For each one of our key-exchange schemes, the simulation was performed in Shawn using 12 different configurations as a result of combining the following factors:

- Grid of nodes in size 5x5, 10x10, 20x20, 30x30, 35x35, and 40x40, node distance 1.0
- Transmission ranges 1.0 and 1.5 (1.0 for square-lattice type of connectivity, 1.5 for diagonal neighbours additionally. See Figure 3.3 as well)

The *Eschenauer & Gligor* algorithm was simulated with two different pool sizes: 1000 keys and 2000 keys. The key ring size on each node was kept constant at 100 keys. This decision was related to the amount of storage available on Tmote Sky sensor nodes. In order to obtain statistically sound results, the *Diffie-Hellman Lite* simulations were each repeated 100 times. Due to the probabilistic key distribution in *Eschenauer & Gligor*, which led to occurrences of the third key-exchange phase, the simulations results were spread at a much larger range than *Diffie-Hellman Lite*. Therefore, we decided to do more iterations when using *Eschenauer & Gligor*. We did even more iterations when using the larger pool size since the number of "unusable" simulations (where at least one node cannot be connected to the network) rises with the size of the network (see Section 4.2.2 for details). Table 4.1 summarizes the simulation parameters. We always used the same number of simulations for both transmission ranges.

Every configuration used the DSDV routing-algorithm provided in Wiselib. The entire evaluation was carried out on the Ubelix-Cluster [11]. A simulation was terminated when every possible link between neighbouring nodes in the whole network was established. Simulation results measuring the amount of data sent only show data used for key-establishment.

	Number of Nodes					
	5x5	10x10	20x20	30x30	35x35	40x40
DH Lite	100	100	100	100	100	100
EG, 1000 keys	200	200	500	500	500	500
EG, 2000 keys	400	400	1000	1000	1000	1000

Table 4.1: Simulation parameters: Number of iterations used for different types of configuration. EG, 1000 keys corresponds to the *Eschenauer & Gligor* simulations with a pool size of 1000 keys, EG, 2000 keys to the simulations with a pool size of 2000 keys.

4.2 Simulation Results

In this Section we compare both key-exchange schemes in terms of performance and explain why the differences occur. Afterwards we discuss probabilistic effects in *Eschenauer & Gligor* that led to variations in the simulation results.

4.2.1 Performance

In the following, if nothing else is specified, all performance measurements for *Eschenauer & Gligor* were done with a pool size of 1000 keys.

Total Time for Key-Establishment

Figure 4.1 shows the amount of time it took until every node established a key to all its neighbours depending on the network size. Two things in this Figure need to be explained in detail:

- There is a constant displacement between *Diffie-Hellman Lite* and *Eschenauer & Gligor*
- When simulating *Eschenauer & Gligor* in large scale networks, we measured a high variation. Hence, the standard deviation bars are huge

The constant displacement occurs as a result of the implementation design: The *Diffie-Hellman Lite* algorithm periodically broadcasts a `HELLO` message. When received by neighbours, they wait for a short period of time and then broadcast their public secret. This means that as soon as the first node broadcasts its `HELLO` message, almost all links in the neighbourhood get established as nodes receive public secrets who did not send a `HELLO` message yet. Compared to *Eschenauer & Gligor*, where the identifier message is sent periodically, only links concerning the node that sent the identifier message get established. This results in a much faster key-establishment in *Diffie-Hellman Lite*.

In order to understand why the huge standard deviation bars occur when using *Eschenauer & Gligor* in large networks, we first have to take a look at the small networks, where no deviation occurs. This happens when the *Eschenauer & Gligor* simulation had the “perfect run”. The “perfect run” means that all links were established using only the second phase of the key-exchange scheme; i.e. all nodes shared a common key in advance - there was no need to agree on a key in the third, the path establishing phase. Therefore, every node had to send its identifier

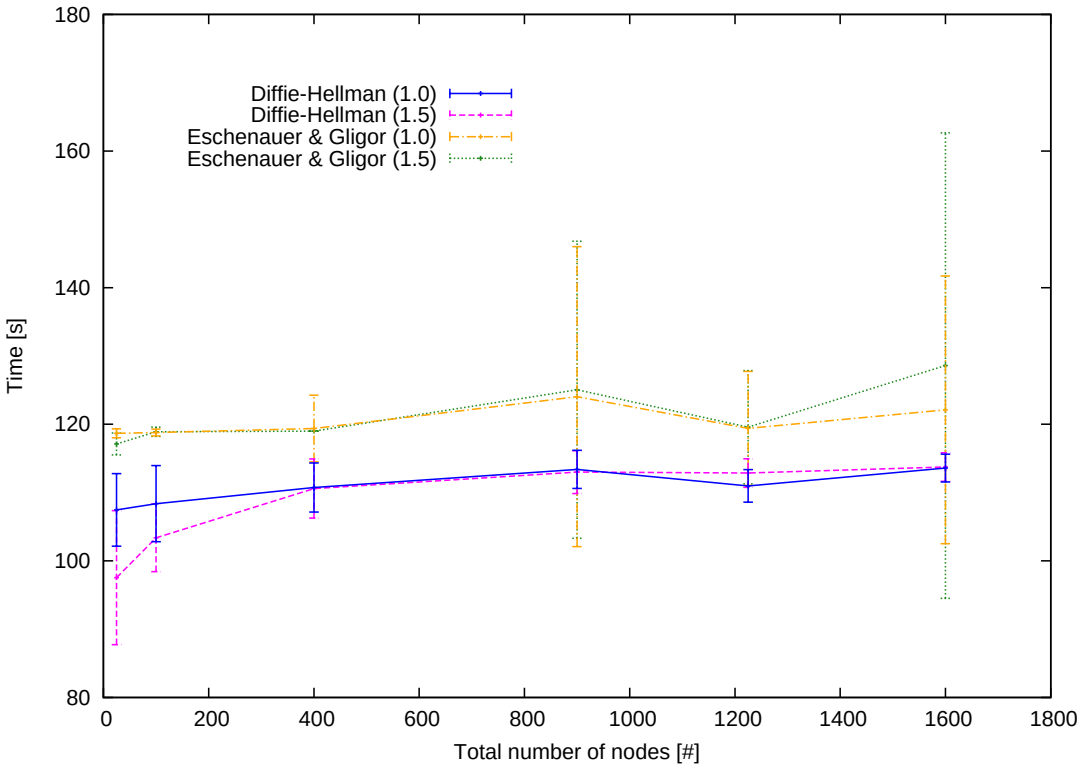

Figure 4.1: *Diffie-Hellman Lite* vs. *Eschenauer & Gligor*: Time until the whole network is connected. Both algorithms were measured with different transmitting ranges - each having 1.0 and 1.5.

message once and the whole network was connected. On the other hand, when having large scale networks, the probability that some of the links are damaged and need to be fixed in the third phase of the key-exchange scheme increases (see Section 4.2.2 for details). As described at the end of Section 3.2.3, damaged links that would require a path over several intermediate nodes in order to be fixed, will be re-established iterative. Therefore, lots of additional broadcasting of identifier messages occurs, thus resulting in huge deviation for the time it takes until the whole network is connected.

Data Overhead for Key-Establishment

In Figure 4.2, we compare the number of messages being sent until the network is connected (all keys have been established) in relation to the network size. The main difference between the results of *DH* and *EG* is, that in *Diffie-Hellman Lite*, every node has to send at least 5 messages: One being the `HELLO` message, and four messages containing the public secret. On the other hand, when *Eschenauer & Gligor* hits the "perfect run", every node has to send only one message - the message containing its identifier list. We can once again see the deviation in *Eschenauer & Gligor* that occurs in large scale networks because of third phase key-establishment (Figure 4.2).

In Figure 4.3 we compare the number of bytes being sent until the network is connected in relation to the network size. The difference of the results lies in the huge public secret being transmitted in *Diffie-Hellman Lite*. This means transmitting 512 Bytes (4096 Bit long prime number) per node, in comparison to 200 Bytes in *Eschenauer & Gligor* (100 keys per node, 2 Bytes per identifier). Of course the performance of *Eschenauer & Gligor* strongly relates to the amount of keys being stored on the nodes prior to deployment. However, 100 keys for 2-8 neighbours ensures that enough unused keys are left unassigned and can be used for newly added nodes, or when keys get blacklisted. The deviation in *Eschenauer & Gligors* large scale network simulations once again relates to the third phase key-establishment.

Additional Data Space used for Key-Establishment

Concerning the amount of data space used, *Eschenauer & Gligor* performs much better than *Diffie-Hellman Lite*, as in *Eschenauer & Gligor*, for every link the node id, and a 2 Byte identifier needs to be stored. Additional data space is used to store the keys and the identifiers. *Diffie-Hellman Lite* on the other hand has to store the public secret and the node id for every link, thus wasting much more space, as the public secrets are 4096 Bit in size. Once the key has been derived the secrets could be dropped, but as dynamic memory management is not possible in Wiselib, the space remains allocated. Additionally $3 \cdot 4096$ Bit are used to store the prime number, the public and the private secret.

Conclusion

As depicted in Figures 4.2 and 4.3, *Eschenauer & Gligor* performs much better than *Diffie-Hellman Lite* when it comes to the amount of data being sent through the network. It is interesting to note that in all three Figures depicting the performance (4.1, 4.2 and 4.3) the different

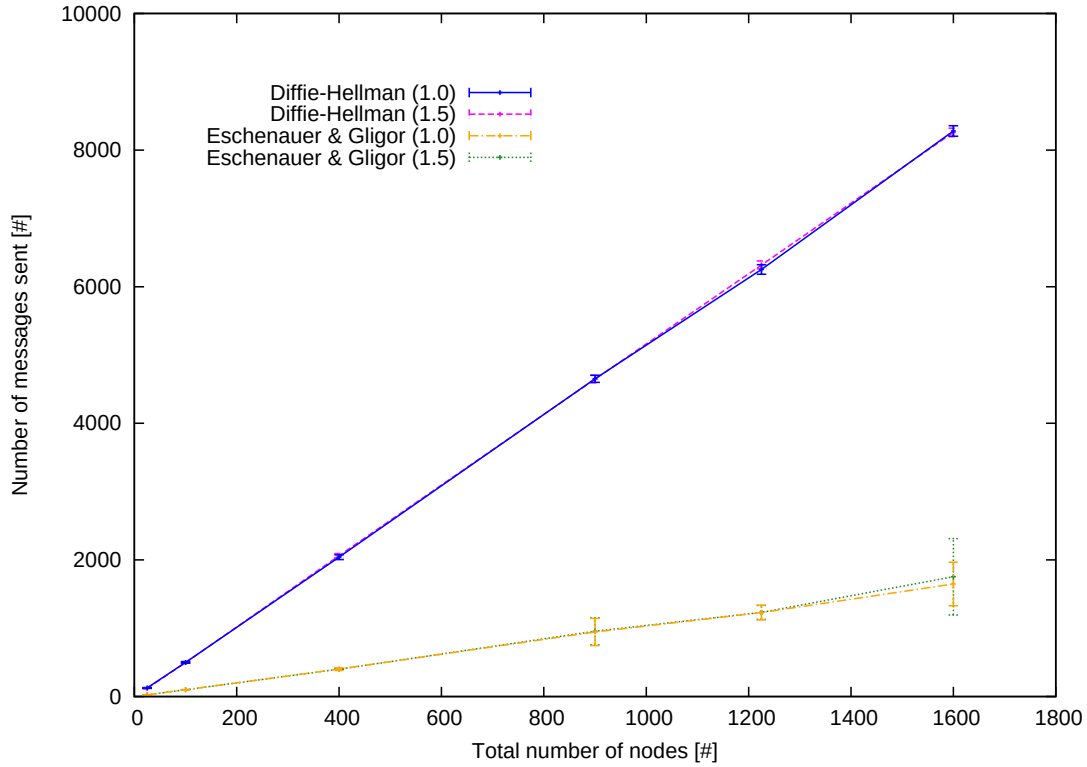

Figure 4.2: *Diffie-Hellman Lite* vs. *Eschenauer & Gligor*: Number of messages sent until all possible links were covered. Both algorithms were modelled with different transmitting ranges - each having 1.0 and 1.5.

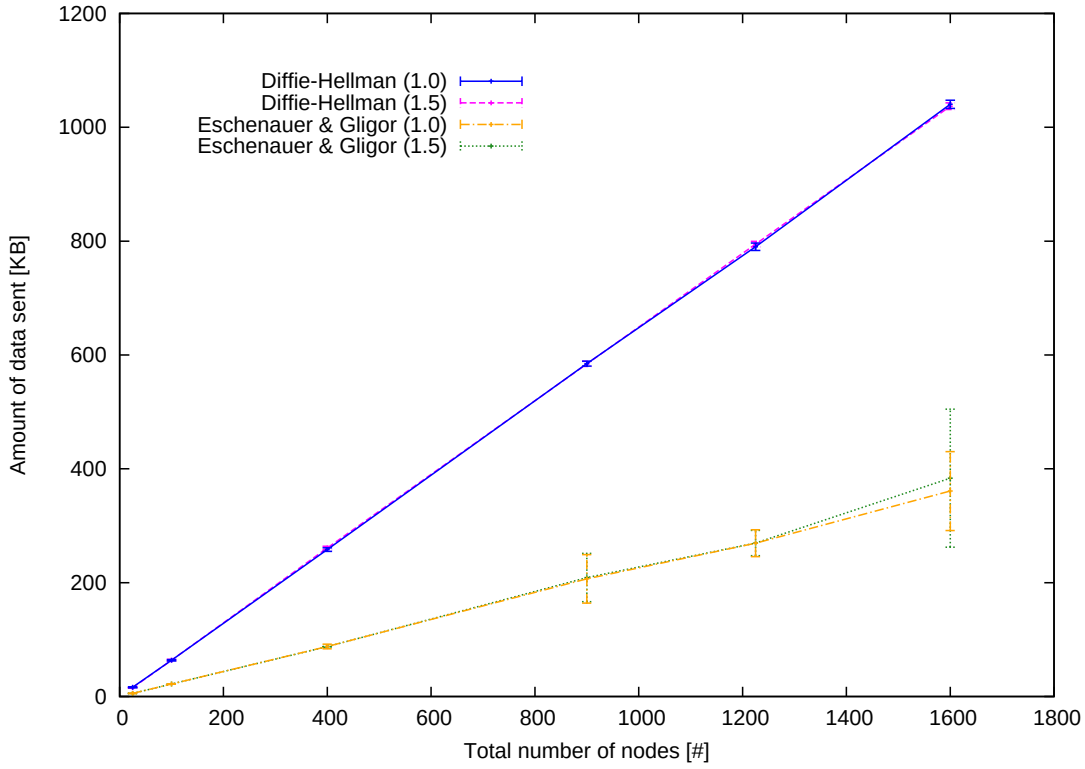

Figure 4.3: *Diffie-Hellman Lite* vs. *Eschenauer & Gligor*: Number of Kilobytes sent until all possible links were covered. Both algorithms were measured with different transmitting ranges - each having 1.0 and 1.5.

ranges do not affect the mean value much. The only differences that occur in the larger networks (when using *Eschenauer & Gligor*) are the standard deviation bars. When using the bigger transmission range, the deviation gets bigger as well. The reason this occurs, is because more links may potentially be left unestablished after the second phase of the key-exchange.

4.2.2 Eschenauer & Gligor: Probabilistic Effects

Due to the probabilistic key distribution of the *Eschenauer & Gligor* algorithm, it is very probable to have damaged links in large scale networks. As we've seen in Figure 4.1, the standard deviation bars get really big for the largest networks we simulated. Figure 4.4 takes a closer look on what happened in the *Eschenauer & Gligor* simulation that was done for the 1600 nodes network with a transmission range of 1.5 and a pool size of 1000 keys. It shows that 86% (of the 500 iterations that were simulated) reside at 119 s. As this were the fastest runs, these were the so-called "perfect runs" where every node just had to broadcast its identifier list once. 8.2% of the iterations are scattered between 152 s and 311 s, i.e. third phase key-establishment was necessary here. The rest, 5.8%, are iterations where at least one node could not be connected to

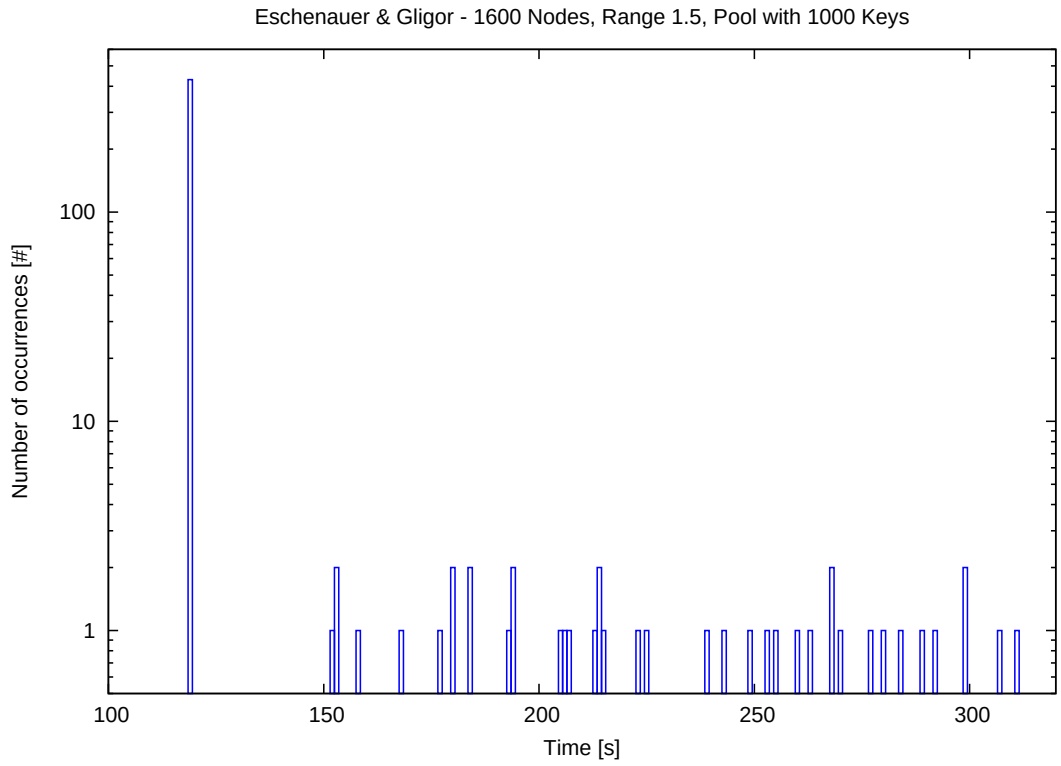

Figure 4.4: *Eschenauer & Gligor*: Time Histogram for 1600 Nodes, Range 1.5. Please note the logarithmic scale on the y-axis. 500 iterations were done, 86% reside at 119 s ("perfect run"), 8.2% are scattered between 152 s and 311 s, 5.8% are iterations where at least one node could not be connected to the network.

the network. We will investigate this further in the next Paragraph. Figure 4.5 takes a closer look on the amount of data being sent in these 500 iterations. Once again, 86% of these 500 iterations were the "perfect runs" where 350 KB of data was sent. The 8.2% of scattered results reside between 403 KB and 1054 KB. As there was almost three times the amount of data sent in the worst case as there was in the "perfect run", lots of links needed to be fixed in these iterations.

As mentioned earlier, we simulated the *Eschenauer & Gligor* algorithm with different key pool sizes; one containing 1000 keys and one containing 2000 keys. We also agreed on a fixed key ring size (100 keys) to store on the nodes. As Figure 4.6 shows, the probability of having at least one node that cannot be connected to the network increases highly as the network size grows when using the pool with 2000 keys. With help of equation 2.1, we can calculate the probability of sharing a common key to a neighbour, when drawing a certain amount of keys out of big pool containing P keys. For a fixed key ring size of 100 keys and a pool size of 2000 keys the probability of sharing at least one common key to a neighbour is at $p = 0.995$. If we analyze the network with 900 nodes and a transmission range of 1.0 we have 1740 different links that need to be established in the whole network! The probability that some of these links

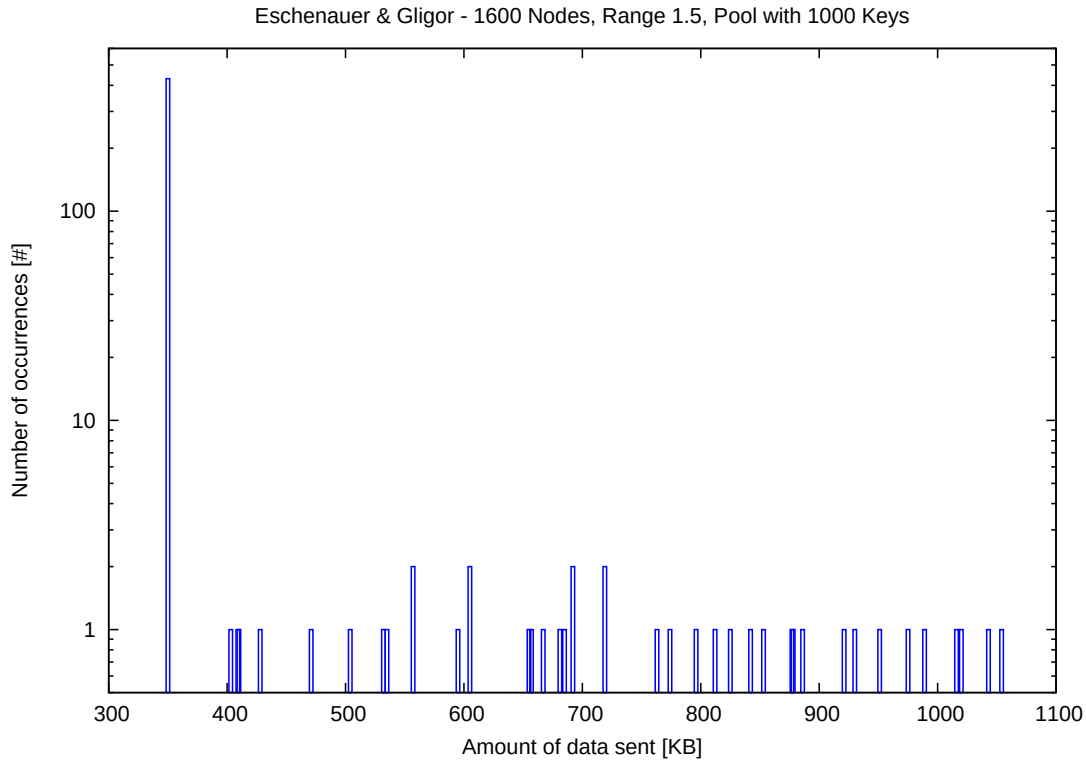

Figure 4.5: *Eschenauer & Gligor*: Histogram of the Amount of Data sent for 1600 Nodes, Range 1.5. Please note the logarithmic scale on the y-axis. 500 iterations were done, 86% reside at 350 KB ("perfect run"), 8.2% are scattered between 403 KB and 1054 KB, 5.8% are iterations where at least one node could not be connected to the network.

are damaged is tending towards 100%. As seen in Figure 4.6, the measurements support this calculation: The probability that at least one node cannot be connected for the network with 900 nodes and transmission range of 1.0 lies at 98%. On the other hand - if we consider the measurements with a pool size of 1000 keys, we have a probability of 99.999% of sharing a common key with a neighbour. Figure 4.6 shows that the probability that whole nodes cannot be connected to the network is reasonably low even in large scale networks.

Figure 4.6 also shows that simulations with higher transmission range perform even worse in terms of connectivity. Equations 2.2 and 2.3 reveal: For both ranges, one having 4, one having 8 neighbours at maximum, the necessary connectivity of sharing a common key to a neighbour resides at 1.0, i.e. in order to connect the network at all times, two nodes need a probability of 1.0 for sharing a common key. This means that the increased transmission range does not (yet) help preventing connectivity problems (as for 4 and 8 neighbours the necessary probability is the same), but instead there are just more links to set up. Of course with more links needing to be established, the number of damaged links increases. As stated in the calculation example at the end of Section 2.4 however, we could improve the situation if we had for instance 20 neighbours on average. Then the necessary probability of sharing a common key to a neighbour in order to connect the whole network lies at 0.72. Therefore, with a probability of 0.995 of sharing a common key to a neighbour when using the pool with 2000 keys, we would practically always be able to fully connect the whole network.

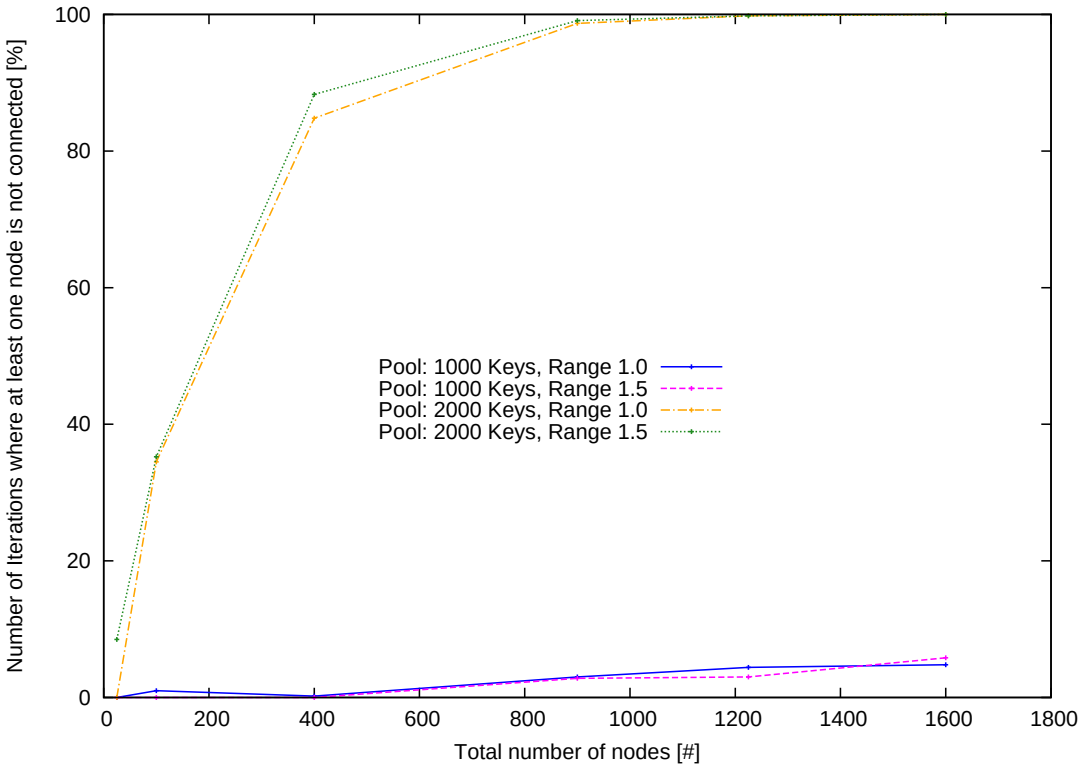

Figure 4.6: *Eschenauer & Gligor*: Number of iterations where at least one node can not be connected to the network.

Chapter 5

Conclusion

5.1 Summary & Conclusion

In this thesis we implemented two different key-exchange mechanisms as generic Wiselib algorithms and compared both in terms of performance (time and data) in small and large scaled WSNs. The implementation of *Eschenauer & Gligor* followed the original scheme exactly. The original scheme by Diffie and Hellman however, did not suit the requirements in wireless sensor networks, so we implemented a simplified version of the original scheme called *Diffie-Hellman Lite*. When it comes to the amount of data sent, we clearly showed that our implementation of the *Eschenauer & Gligor* key-exchange scheme performed much better than *Diffie-Hellman Lite* (the original scheme proposed by Diffie and Hellman would perform even worse). *Diffie-Hellman Lite* sends about five times the number of messages compared to *Eschenauer & Gligor*, and 2.9 times the amount of data. In terms of the time needed to establish all the connections in a network, *Diffie-Hellman Lite* is slightly faster than *Eschenauer & Gligor*. Additionally, a lot more data space is needed on the nodes when using *Diffie-Hellman Lite*.

When evaluating *Eschenauer & Gligor*, using larger pools than 1000 keys (when having 100 keys on every node) resulted in too many nodes being disconnected when evaluating large scaled networks. As the percentage of disconnected nodes remains constant for given key- and neighbour-constraints, the absolute value of disconnected nodes increases with the total number of nodes.

Regarding the actual simulations done, the amount of data being sent, *Eschenauer & Gligor* performed as expected. We were however a bit surprised that *Eschenauer & Gligor* needed more time to establish all connections than *Diffie-Hellman Lite*, but as explained in Section 4.2.1 it was related to an implementation detail. Time however, is not of major concern in WSNs. The amount of data transmitted, which impacts on the energy consumption, is more important. As Shawn did not provide any way of measuring computational delays during simulation, we wanted to make sure that these delays would not cause any differences in the simulation results. The key deriving in *Diffie-Hellman Lite* was the computationally most demanding step in both key-exchange schemes. Therefore, we investigated on additional delays caused by this key deriving. In order to estimate the time needed for the *Diffie-Hellman Lite* key deriving on sensor nodes, we investigated on a scaling factor between computation on a sensor node and a Pentium IV. Simple mathematical computations, repeated a thousand times were used to calculate the

scaling factor, as the actual key deriving could not be measured on real nodes (see Section 5.2). Eventually, we measured the time to derive a key on the Pentium IV and with help of the scaling factor, we estimated how much time it would take to derive the key on a sensor node. Fortunately, the additional delay imposed by the key deriving was negligible and therefore had no impact on our simulation results.

Overall we can safely confirm that the *Eschenauer & Gligor* is much more suitable for WSNs than *Diffie-Hellman Lite*, as less storage space is needed, less data has to be sent, keys can be revoked and less computation needs to be done on the nodes. The performance of *Eschenauer & Gligor* however, strongly relates and the size of the key pool used, the key-ring size and the mean amount of neighbours. As we've seen in the simulation results - when using large scale networks - increasing the key pool size leads to disconnected nodes or subnets. To improve this situation, either the pool size has to be kept low, or the key-ring size has to increase. Raising the key-ring size however is limited by the amount of storage available on the sensor nodes. Besides that, raising the key-ring size means that more identifiers need to be broadcasted. This results in more traffic sent and therefore higher energy consumption. Another solution would be to increase the transmission range, in order to increase the neighbourhood. As stated at the end of Section 2.4, with a big enough neighbourhood, the key-ring size can be hold constant while increasing the key pool size and still maintain connectivity through the whole network. However, increased transmission range as well results in higher energy consumption. Detailed evaluations would need to be done to determine the most energy efficient solution combining transmission range and key-ring size.

5.2 Pitfalls & Future Work

At first we wanted to evaluate both algorithms on real sensor nodes as well. However, at the time of writing this thesis, it was not possible to compile our algorithms for the Tmote Sky nodes, due to a Wiselib related bug. Early versions of the *Eschenauer & Gligor* algorithm were running on the nodes - *Diffie-Hellman Lite* never did, as there was no suitable arithmetics library provided in Wiselib. Unfortunately, the final version of *Eschenauer & Gligor* was not running either on sensor nodes.

Several transmission models are available in Wiselib and beside the reliable model we used, CSMA was planned to be used in the simulations as well. Unfortunately very strange behaviour occurred: *Eschenauer & Gligor* performed much worse than *Diffie-Hellman Lite* in terms of the amount of data being sent. There was no explanation for this, as *Diffie-Hellman Lite* performed exactly the way it did when using the reliable transmission model. We agreed to discard these simulation results as it was very probable that only a CSMA related bug would be visualized.

It would be very interesting to see how our implementation of *Eschenauer & Gligor* would perform on real sensor nodes. It would be interesting to see how establishing keys and encrypting messages would compare to unencrypted traffic in terms of e.g. energy consumption.

As for improvements of the Eschenauer & Gligor key-exchange scheme, there are two (interesting) papers that could further enhance the algorithm's performance:

- [12] deals with the problem when lots of nodes get captured in WSNs. Modified random

key drawing in the first phase of the scheme is used to further improve resistency against node capturing.

- [13] is a further improvement of the paper mentioned above. It uses deployment knowledge in order to raise the probability of sharing common keys.

References

[1] "WISEBED - Wiselib Project," Oct. 2010. [Online]. Available: http://www.wiselib.org/

[2] T. Baumgartner, I. Chatzigiannakis, S. P. Fekete, C. Koninis, A. Kröller, and A. Pyrgelis, "Wiselib: A Generic Algorithm Library for Heterogeneous Sensor Networks," in *EWSN*, ser. Lecture Notes in Computer Science, J. S. Silva, B. Krishnamachari, and F. Boavida, Eds., vol. 5970. Springer, 2010, pp. 162–177. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11917-0

[3] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[4] Eschenauer and Gligor, "A Key-Management Scheme for Distributed Sensor Networks," in *SIGSAC: 9th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2002.

[5] A. Kroeller, D. Pfisterer, C. Buschmann, S. P. Fekete, and S. Fischer, "Shawn: A new approach to simulating wireless sensor networks," Feb. 01 2005, comment: 10 pages, 2 figures, 2 tables, Latex, to appear in Design, Analysis, and Simulation of Distributed Systems 2005. [Online]. Available: http://arxiv.org/abs/cs/0502003

[6] J. francois Raymond and A. Stiglic, "Security Issues in the Diffie-Hellman Key Agreement Protocol," *IEEE Transactions on Information Theory*, vol. 22, pp. 1–17, Dec. 19 2000. [Online]. Available: http://crypto.cs.mcgill.ca/~stiglic/Papers/dhfull.pdf

[7] "Sophie Germaine Primes for Diffie-Hellman Key Exchange," May 2010. [Online]. Available: http://www.floatingdoghead.net/bigprimes.html

[8] "The GNU Multiple Precision Arithmetic Library," May 2010. [Online]. Available: http://gmplib.org/

[9] National Institute of Standards and Technology (NIST), "Secure hash standard," Federal Information Processing Standards Publication (FIPS PUB) 180-2, Aug. 2002. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[10] National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)," Federal Information Processing Standards Publication (FIPS PUB) 197, Nov. 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[11] "UBELIX - University of Bern Linux Cluster," Oct. 2010. [Online]. Available: http://ubelix.unibe.ch/

[12] H. Chan, A. Perrig, and D. Song, "Random key predistribution schemes for sensor networks," in *Proceedings of the 2003 Symposium on Security and Privacy*. Los Alamitos, CA: IEEE Computer Society, May 11–14 2003, pp. 197–215.

[13] W. Du, J. Deng, Y. S. Han, S. Chen, and P. K. Varshney, "A key management scheme for wireless sensor networks using deployment knowledge," in *INFOCOM*, 2004. [Online]. Available: http://www.ieee-infocom.org/2004/Papers/13_1.PDF