Table of Contents

Acknowledgments

After working about five months this bachelor thesis is finished. Many people have contributed to my work by giving guidance, opportunities and support.

First of all I want to express my gratitude to the people of Twente Institute for Wireless and Mobile Communications (WMC) for the opportunity to finish my bachelor thesis at their research center. The assignment was challenging, because I have learned a lot about computer security and wireless communication, and I also enjoyed it, because it was practical as well.

Also I want to thank my supervisors: Simon Oosthoek, Assed Jehangir and Sonia Heemstra de Groot for their time, comments, guidance and for our interesting weekly discussions at WMC.

Next, I want to thank Jordi Jaen Pallares, the author of the PFP for helping me out with the source code and the interesting discussions we had about security in the PFP implementation.

Finally, I express my gratitude to my friends and family for their support and especially my mother for correcting the drafts of this thesis.

Abstract

The broadcast nature of wireless computer networks causes these networks to be insecure. Therefore data in such networks needs to be protected to prevent it can be read and modified by an attacker. Data protection in such networks is accomplished by establishing a secret (henceforth called a key) between the trusted devices. The key represents the secure association between the trusted devices and is used for encrypting the original data into a ciphertext to ensure only the trusted devices *with* this key can retrieve the original data.

This report analyzes and describes a protocol, the PAN Formation protocol (PFP), and its implementation for establishing a key over an insecure communication channel, and analyzes its security aspects as well. Furthermore, the report describes two contributions for the problems we found in the protocol.

- One problem with the protocol's application is that it could not easily be started and this is solved by implementing a user-interface for the protocol.
- A second problem of the protocol is the scalability of the user-interactions. A solution for this problem is an extension of the protocol, called Transitive PFP (TPFP), where devices can automatically establish keys based on the secure association they already have with other devices.

The result of this research is that with further development of the PFP and TPFP, secure associations in ad-hoc networks can be created easily by the user. Still there is a number of difficulties to overcome regarding to the PFP implementation and the TPFP implementation has to be further developed.

1 Introduction

This Chapter presents background information, thesis structure and the project achievements.

The structure of the Chapter is as follows:

- Section 1.1 gives background information about the MAGNET project and how the PFP is related to MAGNET. Additionally background information about the security of communication data is given.
- Section 1.2, gives the structure of the report and the project's accomplishments.

1.1 Background

MAGNET stands for My personal Adaptive Global NET and is a world wide Research and Development project focused on Mobile and Wireless Systems. MAGNET Project has various research partners from different countries. The partners are a mix of Universities, Research Institutes, Industrial Partners and SME's (Small and Medium Enterprise). WMC is one of these partners.

In the MAGNET Project research is done on user-centric networks and examples of such networks are Personal Networks (PN) or Personal Area Networks (PAN) [NH-ADPN]. The goal of MAGNET is to introduce new technologies in such networks to improve the quality of life for the user. The need for user's mobility contributes that many communication links in these networks are wireless. More information on MAGNET and these user-centric networks can be found at [MAGNET] and at [MD2.1.2].

1.1.1 Security of communication data

Security in wireless communication is more of an issue than in wired technology. Data is transmitted using a radio and all devices in range of the transmitting device are able to receive this data and therefore wireless networks are more vulnerable to attacks. A secure communication channel has to be created for sending and receiving sensitive data in a wireless network. A secure communication channel is a communication channel where original data is encrypted with a shared key, such that devices with this same key are the only ones who can retrieve the original data from this channel. Devices wanting to use a secure communication channel, must first establish a shared key and this key represents the secure association between them. MAGNET has developed a network protocol, called PFP, for establishing such a key, which will be described in Section 1.1.2.

There are different concepts on how secure associations in networks can be created. One concept is that all the devices in a network have one common key, the group key. A second concept is that every pair of devices has a secure association with each other and therefore each of these pairs has to establish a key (henceforth called a pairwise key). In the pairwise key concept much overhead is spent on key distribution, but an advantage of pairwise secure associations is that a mutual secure association can be revoked without losing secure connections to other devices. This document is focused on a protocol for pairwise key establishment and will not discuss the group key establishment.

In terms of how long a key is valid, there are different types of keys. A long-term key can be used for a long period of time and is not normally used for encryption of normal communication data. Instead, the derived or exchanged sub keys from the long-term key are used to send such data. These keys are periodically refreshed, which limits confidentiality lost due to compromise of one session key. For example, if an attacker is somehow able to acquire a sub key, the attacker can only use it for some period of time, and still does not learn much about the long-term key. This document focuses on the establishment of such a long-term, pairwise key.

1.1.2 Imprinting

Imprinting, often called pairing, is the procedure for initiating a secure association between two devices. This secure association is created by establishing a new key, where both devices are successfully authenticated. Authentication is the process of proving to an entity that he is the entity who he claims to be and vice versa. Manual imprinting is a special kind of imprinting, where the user is involved for authentication of the two devices. The PAN Formation Protocol (PFP), which is the main focus of this document, is a manual imprinting protocol for two devices in a PAN or a PN.

The PFP is a protocol for securely establishing a shared pairwise key between two devices over an insecure channel. Although the PFP has been developed in MAGNET, it could also be used in other networks than PNs or PANs. The protocol's name gives a wrong interpretation of what it does and suggests that the protocol is used to form a network, therefore we only use its abbreviation, PFP.

PFP must be a secure network protocol for establishing a key and for secure networks protocols three criteria must be met:

- Authentication: Verifying that the originator of data or message really is the identity, who it claims to be.
- Confidentiality: The assurance that sensitive data is not disclosed to unauthorized identities.
- Integrity: Protection against modification of data.

1.2 Project achievements and thesis structure

This report analyzes the PFP and its security aspects and describes my two contributions to the PFP in MAGNET as well.

Before we can analyze the PFP and its security expects, we have to understand some computer security concepts. Chapter 2 is focused on some of these concepts, which may aid in understanding the PFP. This Chapter also discusses Diffie-Hellman key exchange, on which the PFP is based.

In Chapter 3 we analyze the PFP by looking at the protocol's message exchange, analyze if this message exchange is secure enough and describe how this new pairwise key is generated. Also we pay some attention to the source code of the PFP implementation, which has been developed in the MAGNET project. This Chapter also describes, how this application can be installed on the Nokia 770 and which problems we had in the process.

This thesis brings two contributions to the PFP and the first contribution is a graphical user-interface for the PFP application, which simplifies the user-interaction to start the key-exchange for the user. The PFP application has to be started from the command line interface (CLI) and lot of command line options have to be provided. This especially is a problem when we run the PFP application on the Nokia 770. The development and the implementation of a Graphical User Interface (GUI) to the PFP is described in Chapter 4.

The second contribution is a specification of an extension for the PFP. A problem with the current PFP is that user-interaction is required for each key exchange between a pair of devices. In MAGNET this extension, called Transitive PFP (TPFP) imprinting, has been introduced for solving this scalability problem of user-interactions by using existing secure associations to establish a key between pairs. Chapter 5 proposes a specification for this TPFP imprinting procedure. Finally Chapter 6 gives the conclusion of this project.

2 Computer security concepts and algorithms

As already been mentioned in the introduction security is important, because PFP is a security mechanism for establishing keys. Secure connections can be created using encryption and decryption techniques on the communication data in such a way that it is unreadable for attackers. A number of general security concepts will be explained in this Chapter and will aid in the understanding of the PFP, which will be discussed in Chapter 3. More information about cryptography and network security can be found in [FS-PC, MI-TSS].

2.1 Encryption and decryption

Encryption is done by using an algorithm and a key for changing the original data, known as the *plaintext*, into the *ciphertext*. A *ciphertext* is unreadable without special knowledge. Most of the encryption techniques use a special value, a key. The key determines the *ciphertext;* a different key gives a different *ciphertext* and it is computationally difficult to calculate the *plaintext* if only the *ciphertext* is known.

Decryption is the reverse process of encryption and changes the *ciphertext* into the original *plaintext*. Most of the decryption algorithms can only decrypt the ciphertext into the *plaintext* if a certain value, the key, is known. Without the right key, the *ciphertext* can not be decrypted into the original *plaintext*. Two main encryption types exist: symmetric and asymmetric encryption. Both types will be discussed in the next sections.

2.1.1 Symmetric encryption

The basic concept of symmetric encryption is that the symmetric algorithm uses the same key for encryption and decryption. An application for symmetric encryption is the encryption of communication data with a key, ***K***, between two entities, for example Alice and Bob. Alice and Bob have already acquired somehow the same key, ***K***. When Alice encrypts the *plaintext* into the resulting *ciphertext* with the key ***K***, Bob can decrypt the *ciphertext* with the same key ***K***. This principle can also be used for securely sending data over an insecure communication channel. Other persons not having the key ***K***, are not able to decrypt the *ciphertext* retrieved from the communication channel into the original *plaintext*.

Below we give some properties of symmetric encryption in formulas:

$E(\boldsymbol{K_A}, M) = M'$, encrypts the *plaintext* M, into the resulting *ciphertext*, M'.
$D(\boldsymbol{K_B}, P) = P'$, decrypts the *ciphertext*, P into P'.

$D(\boldsymbol{K_A}, E(\boldsymbol{K_B}, plaintext)) = plaintext$ if $K_A = K_B$

$E(\boldsymbol{K_A}, M)$ is the encryption function with the following parameters: the key, $\mathbf{K_A}$ and the *plaintext*, M. The result of function E will be the *ciphertex*t, M'.

D(K_B, P) is the decryption function with the following parameters: the key, $\mathbf{K_A}$ and the *ciphertext*, P. The result of function D will be P'. The decryption function will only return the original *plaintext* if the keys are the same ($\mathbf{K_A}$=$\mathbf{K_B}$).

There are a number of different symmetric encryption algorithms: two popular and well-known encryption algorithms are Advanced Encryption Standard (AES) and Blowfish. AES supports 128/192/256 bits key sizes. Blowfish has variable key length with the maximum length of 448 bits.

2.1.2 Asymmetric encryption

An asymmetric encryption is an algorithm, which uses one key for encryption and an other key for decryption and vice versa. Each party generates one pair of keys: one key, the private key, is only known by the owner. The second key, the public key, is public and everybody is allowed to know this key. The pair of keys are generated in such a manner that, if the *plaintext* is encrypted with the private key, it can only be decrypted with the public key and vice versa.

Some examples of an asymmetric encryption algorithm are ElGamal and RSA (Rivest, Shamir and Addleman). ElGamal and RSA are based on the discrete logarithms problem and these keys must have a relative large key length to be secure enough. A 1024 bits key is usually recommended as minimum key length. More information can be found in [MI-TSS]. Another kind of asymmetric cryptographic algorithm is one based on elliptic curves. The advantage of elliptic curves algorithm is for the keys to be much smaller. Additionally this algorithm is less computationally expensive than cryptographic algorithms based on discrete logarithm problems. It is claimed that a 160-bit public key based on elliptic curves scheme is as secure as a 1024-bit public key based on the RSA scheme.

2.1.3 Comparing Symmetric and Asymmetric Encryption

Both types of encryption are widely used, because both have different properties. In some applications one encryption type has advantages over the other. The main advantage of symmetric encryption is that it is faster. The advantage of asymmetric encryption is that it can be used for solving different types of security problems, such as key exchange, authentication, certificates, etc. We give a short overview of both encryption types below.

The advantages of using symmetric key encryption are:

- Encryption algorithms with symmetric keys are faster, because most of them typically require smaller keys and are less computationally expensive than asymmetric key algorithms.
- A key can be shared among two or more parties. Every party who has acquired this key, can decrypt and encrypt messages with the same key. When a party has encrypted a message with a certain key, an other party, who has acquired this same key, will be able to decrypt the *ciphertext* into the original *plaintext*.

The advantages of using asymmetric key encryption are:

- Asymmetric key encryption can perform different functions. For example, confidentiality: if you only want a certain party to read a message, or authentication: if you want everybody to be able to decipher the message and to ensure that they know it is coming from you.

- Asymmetric key encryption can be used for digital signatures or certificates.
  - A digital signature is additional data added to a message, so the receiver can validate that the data is coming from the originator and not coming from another person. The digital signature has been encrypted with the originator's private key and the receiver can validate the message by decrypting it with the originator's public key. A message containing identification data and a digital signature is a certificate. More information about digital signatures and certificates can be found in Chapter 3.7 till Chapter 3.12 of [MI-TSS].
- Asymmetric key encryption is often used for key exchange [FS-PC, MI-TSS]. One party can send key exchange parameters by encrypting it with the other's party public key. The message can only be decrypted by the other party, who has this private key. The problem with symmetric encryption is that it is necessary to have a key to securely exchange key parameters. An example of a key exchange protocol is Diffie-Hellman.

The PFP is based on the Diffie-Hellman key exchange. In order to better understand the PFP, it is useful to understand the Diffie-Hellman key exchange. In the next section a short explanation of the Diffie-Hellman key exchange is given.

2.2 Diffie-Hellman key Exchange

Diffie-Hellman is a cryptographic key exchange protocol allowing two devices to exchange a secret key over an insecure channel. Diffie-Hellman is based on the discrete logarithm problem, where a device uses the public key of the other node and its own private key to create a new shared key. After the key exchange, these devices can establish a new secure communication channel using this new key. Each public and private key is a unique pair of keys, therefore the newly generated key will also be unique.

A detailed description of this key exchange will be given below in detail, describing the sequence shown in Figure 1:

If device ***A*** and ***B***, wishes to establish a new symmetric key, the following must be accomplished:

1. p is a large prime and is predefined.
2. g is also predefined and is $2 =< g =< p-2$.
3. A and B both use the same value for p and g.
4. p and g are chosen such that:
   - for every i, $1 < i < p-1$ there must power j such that $i = g^j \bmod p$
5. ***A*** chooses a random number a, which ***A*** must keep secret and this number is ***A***'s private key.
   - ***A***'s public key is: $P_A = g^a \bmod p$
6. ***B*** chooses a random number b, which ***B*** must keep secret and this number is ***B***'s private key
   - ***B***'s public key is: $P_B = g^b \bmod p$
7. ***A*** sends to ***B*** its public key.
   - ***A*** -> B: P_A

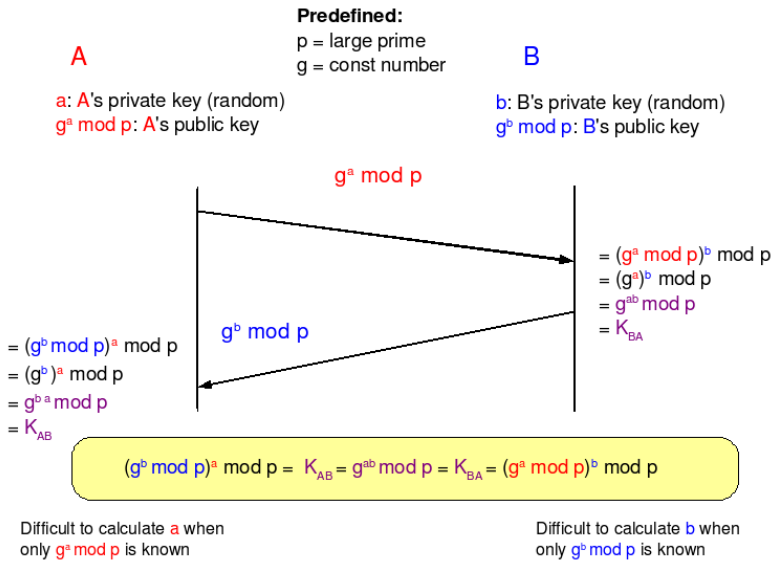

Figure 1: Message sequence of the Diffie-Hellman key exchange.

8. ***B*** now knows the ***A***'s public key. Therefore ***B*** can calculate the new symmetric key between ***A*** and ***B***. Calculation of the new key can be done as follows:
   - $P_B=g^b \bmod p$
   - $(P_A)^b \bmod p = (g^a \bmod p)^b \bmod p = (g^a)^b \bmod p = K_{BA}$
9. ***B*** sends to ***A*** its public key.
   - B -> A: P_B
10. ***A*** now knows the ***B***'s public key. Therefore ***A*** can calculate the new symmetric key between ***B*** and ***A***. Calculation of the new key can be done as follows:
    - $P_B=g^b \bmod p$
    - $(P_A)^b \bmod p = (g^b \bmod p)^a \bmod p = (g^b)^a \bmod p = K_{AB}$

    To prove, that the keys generated on A and B are the same:
    $(g^b)^a \bmod p = g^{ba} \bmod p = (g^a)^b \bmod p = K_{AB} = K_{BA}$

An advantage of Diffie-Hellman is that it can establish a shared key over an insecure channel. Both parties have established the same pairwise key ($K_{AB} = K_{BA}$) with the other's public key and its own private key. The protocol is confidential in the sense that the private key can not be known by others.

A known weakness in the Diffie-Hellman Key exchange is that authentication is not provided. ***A*** and ***B*** do not have any guarantee that they are communicating together. An intruder may pretend that he is ***B*** and can lure ***A*** into key exchange. This kind of attack is known as middle person attack, or man in the middle attack. The PFP which will be discussed later, is designed to solve the vulnerability of middleperson attack by using a out of band channel for authentication.

2.3 One-way functions

One-way functions are functions where it is easy to compute the result of such function, but computationally infeasible to calculate the parameters if only the result is known. One-way functions are often used in security protocols as well as in other applications. A hash function is an one-way function and its properties are discussed in this report, because it is used in the PFP. An other kind of one-way function is a MAC function and can be used for the authenticity and integrity of a message.

2.3.1 Hash functions

A hash function is a function that accepts data of arbitrary length, called the pre-image and generates a fixed-size output, called the hash value [MI-TSS]. Reconstruction of the piece of data from the hash value should be computationally unfeasible. The hash value will not reveal any usable information about the piece of data that being hashed.

The one-way hash function H(M) computes a fixed-length hash value, *h* from message, *M* of arbitrary length, as follows:

$h=H(M)$

A one-way hash function has the following properties:

- for a any value of *M*, it is relatively easy to compute the hash value, *h*
- for any value of *h*, it computationally unfeasible to compute *M*.
- for a message *M* it is very difficult to find an other message *M'* that has the same hash value such
  h(M)=h(M')

Some examples of well-known popular hash-function are MD5, SHA-1 and SHA-256 with a length of 128, 160 and 256 bits, respectively. SHA-1 and SHA-256 will be used in this report.

2.3.2 Message Authentication Codes (MACs)

A message authentication code (MAC) is a short piece of information used for authenticating a message. A MAC algorithm accepts as input an arbitrary-length message and a secret key and outputs a *mac* value. A MAC function is comparable to a hash-function, but can protect the integrity of a piece of data and its authenticity as well.

The mac function MAC(K,M) computes the mac value from a message ***M*** of arbitrary length and with as key ***K***, as follows:

$MAC(\mathbf{K}, M)=mac$

A MAC function has the following properties:

- for a any value of *M* and ***K***, it is relatively easy to compute the mac value.

- for any given ***K***, it computationally unfeasible to compute a new pair (M',K) such that MAC (***K*** ,M) = *mac* = MAC (***K*** ,M')

For example, when two parties have pre-established a key, ***K*** and want to be certain that message is coming from each other, they calculate the encrypted *mac* value and append this to a message. The receiving party can also calculate the mac and if the received mac is the same as the calculate mac, they are certain that it is coming from each other.

Example of a MAC function is HMAC and DAA. The HMAC function used in the PFP application is a 160-bits MAC function and uses SHA-1 for calculation the mac value.

3 PFP key exchange protocol

The PFP is a protocol to establish a pairwise key or when they have already pre-established a key, this protocol should able to validate if this key is still the same on both sides. A pairwise key is a key that is only known to one pair of devices. For each secure association between two devices, a different key is used and therefore a new key has to be established by each pair. The PFP is based on the Diffie-Hellmann key exchange to securely establish key parameters. Afterwards, both devices use these parameters to generate the new pairwise key, which we shall call the Permanent Key. This Chapter describes and analyzes the PFP and its implementation.

3.1 PFP Design

The PFP has been developed in the MAGNET Project. In the process of designing this protocol, it was decided it should contain the following features:

1. Pairwise keys. Exchange of keys between a pair of devices, so that each pair of devices has a mutual secure association in the personal network. When a device has been compromised, the user is able to revoke all secure association between this device. All devices except the compromised device still have their secure associations to other devices and these secure connections are not lost.
2. Long keys. The keys should be of sufficient length to derive session keys or keys that can be used in lower level layer communication.
3. Easy Authentication. Minimal user-interaction to establish a secure association. For example, no long keys should be entered by the user.

The general working of PFP is briefly described below and will be described in more detail in the rest of this chapter:

1. Diffie-Hellman key exchange is used to securely establish a temporary key over an insecure channel. (see Section 2.2)
2. An out of band channel is used for authentication of the initial messages.
3. The temporary key is used to establish a new secure temporary connection.
4. The devices use this secure connection to establish key parameters.
5. Afterwards both devices will be able to generate a new permanent key.

3.2 Proximity Authentication Channel (PAC)

The PFP is based on Diffie-Hellman key exchange for establishing and generating a new key. The problem with Diffie-Hellman is that messages are not authenticated. The solution in PFP is to use a Proximity Authentication Channel (PAC). The PAC is an out of band communication channel between two devices. The two devices can authenticate message from each other by means of the received data on this authentication channel. The PAC is a temporary communication channel and is only used for authentication purposes. An example of a PAC is the user. The user can act as a channel and can re-

trieve information displayed on one device and enters this into an other device. Other examples of PACs are infrared, serial cables, etc.

In the PFP the user will act as the PAC and the PFP uses the PAC as follows:

- One device sends a random number, a code to the PAC. The device displays the code on the screen.
- The user reads this code and will enter the code into the other device. The PFP will use this code to sign the messages by appending an encrypted HMAC to these messages. The other device can check the signed messages and with its calculated signed message.

3.3 PFP Key generation

The PFP has is its own method of generating a new pairwise key. We assume that device A and device B wish to establish a new permanent key. A short description of this generation process:

A pairwise key is generated as follows:
$PK_{AB} = SHA\_256\ (E(TK_{AB}, N_A | N_B)\ )$, when $N_A <= N_B$ or
$PK_{AB} = SHA\_256\ (E(TK_{AB}, N_B | N_A)\ )$, when $N_B < N_A$
$TK_{AB} = g^{ab} \bmod p = K_{AB} = K_{AB}$ (see Section 2.2, Diffie-Hellman)

The identifier for the permanent key is generated as follows:
$PKID_{AB} = SHA\_256\ (N_A, N_B)_{32}$ when $N_A <= N_B$ *or*
$PKID_{AB} = SHA\_256\ (N_B, N_A)_{32}$ when $N_B < N_A$

- PK_{AB} is the permanent pairwise key between A and B. TK_{AB} is the temporary key established using the Diffie-Hellman based key exchange. More information about Diffie-Hellman key exchange can be found in the previous section 3.2. The permanent key, PK_{AB}, is generated of the temporary key and the two nonces[1] of the two devices.
- N_A and N_B are nonces and randomly chosen by A and B. The nonces, N_A and N_B, are the variables used to generate the permanent key and are securely exchanged between the two devices. The secrecy of the nonces must be maintained as much as possible. To ensure that the permanent key is the same on both sides, N_A and N_B have to be swapped in the key generation process, if N_A is larger than N_B. The nonces are also used in the validation process.
- $PKID_{AB}$ is the identifier for the permanent pairwise key between A and B. The advantage of the PKID is that this identifier can be published. The PKID is used to distinguish different permanent keys from each other without publishing the permanent key. When two devices have different permanent keys, they are able to detect this situation, because they also have different PKIDs.

1 The word nonce means number used once. A nonce is a variable that is used once in authentication or in key exchange protocols. The nonce differs each time and each authentication and key exchange procedure is therefore unique. For an attacker it is difficult to generate a corresponding unique reply for the procedure. In the PFP the word nonce is wrongly chosen, because in the validation procedure the same nonce is used and therefore the nonce is used more than once. In this document we have not changed the word nonce, because it might be confusing, when we are referring to other MAGNET documentation.

Below we give the sizes of the different parameters used to generate the permanent key:

- p, prime used for key exchange. p is the oakley prime group 2 and is 1536 bits long.
- g, the constant. In the PFP the value 2 is used.
- a,b, the private keys of A and B. The private keys are 256-bit primes.
- g^a mod p, g^b mod p, the public keys of A and B and are 1536 bits long.
- N_A and N_B are the nonce of A and B and are 256 bits long.
- PK_{AB}, the permanent key between A and B and is 256 bits long.

3.4 PFP Message sequence

This section describes the message sequence of the PFP. Two devices, ***A*** and ***B***, wish to establish a pairwise key: where ***A*** is the client and ***B*** is the server. The server listens for incoming connections and the client connects to the server. After the connection is established they initialize a secure association with each other. Initially, both PFP parties are started in authentication mode (AUTH mode). After they have determined that they have not exchange keys yet, they enter into key exchange mode.

An explanation about the notation used to describe the PFP, can be found in Appendix A.

In MAGNET a new identifier, the Magnet_ID, has been introduced to differentiate devices. The reasons for introducing a new identifier are:

- No network interface addresses are needed to distinguish devices from each other. Addresses can change and a device can have multiple IP addresses or physical addresses, therefore a unique identifier to differentiate devices is more suitable.
- PFP uses the Magnet_ID for distinguishing different devices and therefore it could be used on different protocol layers levels. For example, a new PFP application can be implemented to operate on top of the link layer.

The Magnet_ID is calculated as follows:

$P_A = g^a \bmod p$, the public key of ***A***
$ID_A = SHA1(P_A)$, Magnet_ID of ***A***

3.4.1 PFP Key Exchange

The message sequence of the key-exchange scenario is displayed in the next diagram (Figure 2). The second section in Figure 2 is the part of message sequence where the actual key exchange is performed. The message sequence scenario, which is described in the following sections, assumes that device ***A*** and device ***B*** have not yet exchanged a permanent key and want to establish a new one.

1. The client, ***A,*** connects to server, ***B***, and ***B*** sends its Magnet_ID to ***A*** with the message Req_ID()
   B -> **A**: *Req_ID(SHA-1($g^b \bmod p$))*

2. ***A*** receives the message Req_ID and searches the Magnet_ID in the local database whether A can find the identity of ***B*** or not. Subsequently the following two situations can occur:
   - ***A*** finds the ***B***'s identity and returns the corresponding PKID and Magnet_ID to ***B***. This key validation scenario is discussed in Section 3.4.2 later on.
   - ***A*** does not find ***B***'s identity and returns an EAP_FAILURE message to ***B***. This key scenario will be explained here. As ***A*** cannot find ***B***'s identity and corresponding key, ***A*** must enter into key exchange mode to establish a new key with ***B***. Subsequently ***A*** sends a new generated code, **K**, to the PAC. In the case when the PAC is the user, it would display ***K*** to the user and it is followed by entering ***K*** into device ***B***.
     A -> **B**: *EAP_FAILURE()*
     A -> **PAC**: *K*

3. ***B*** receives the EAP_FAILURE() message, which triggers ***B*** to enter also into key exchange mode. ***B*** then receives ***K*** from the **PAC**. In our case ***K*** will be displayed on ***A*** and the user must enter the code into device ***B***.
4. ***B*** starts the key exchange by sending an empty Req_ID() message to ***A***.
   B -> **A**: *Req_ID()*
5. ***A*** receives this request message and returns the Resp_PFPBegin message containing its public key, $P_A = g^a \bmod p$.
   A -> **B**: *Resp_PFPBEGIN(g^a mod p)*
6. ***B*** receives the message Resp_PFPBegin and retrieves ***A***'s public key. ***B*** calculates the new temporary key, $\mathbf{TK_{AB}} = (g^a)^b \bmod p$, according to Diffie-Hellman.
   B then returns the Req_PFPExchange message. which contains the following:
   - The public key of ***B***, $P_B = g^b \bmod p$.
   - HMAC of the ***B***'s public key encrypted with the key (code), K.
   - The $\mathbf{TK_{AB}}$ encrypted *ciphertext* contains ***B***'s nonce and the HMAC of ***B***'s public key.

   B -> **A**: *Req_PFP_Exchange (g^b mod p | HMAC(K,g^b mod p)*
   | E(g^{ab} mod p, N_B | HMAC(K,g^b mod p))

   The reason for sending unencrypted a second HMAC in the message is that **A** can easily authenticate **B**'s public key without decrypting the message. More information about how the HMACs are used for authentication will be discussed later in Section 3.4.3.

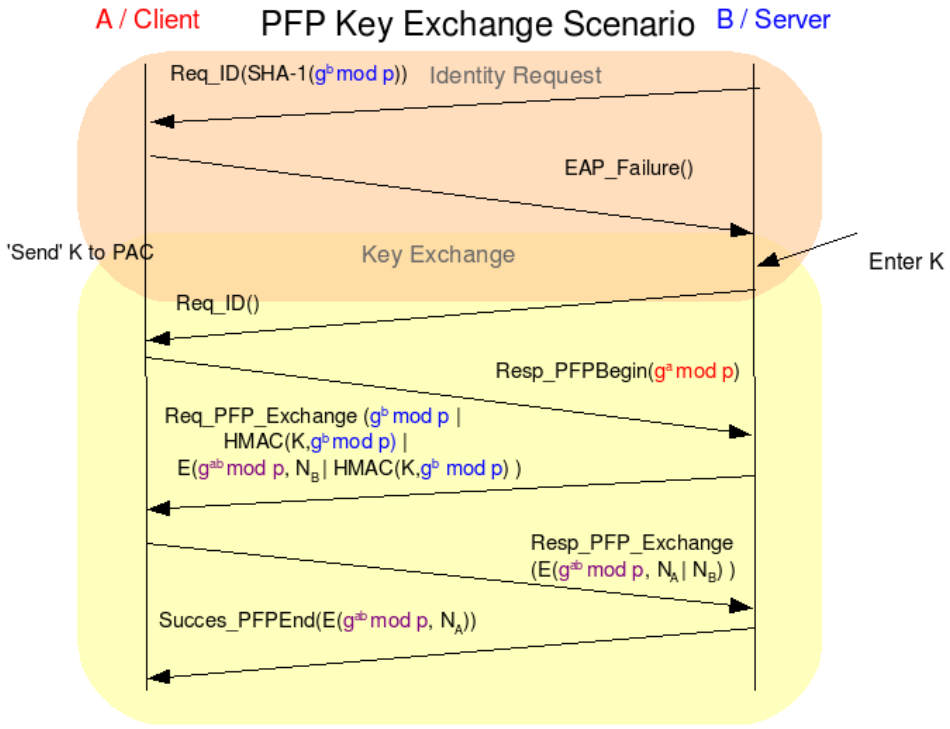

Figure 2: The message sequence diagram of the PFP key exchange scenario between device A and B.

7. ***A*** receives the Req_PFPExchange message, retrieves the ***B***'s public key ($\mathbf{P_B}$) from this message and calculates the temporary Diffie-Hellman key, $\mathbf{TK_{AB}}$. A then decrypts the message with TK_{AB}. Subsequently ***A*** checks the two HMACs in the message against the local HMAC. On success ***A*** knows that authentication of ***B*** has been done successfully and retrieves ***B***'s nonce, N_B. ***A*** now can generate its nonce N_A and sends N_A and N_B encrypted in the message Resp_PFP_Exchange.
   A -> **B**: *Resp_PFP_Exchange ($E(g^{ab} \bmod p, N_A \mid N_B)$)*
8. ***B*** receives the message Resp_PFPExchange and decrypts the message with the key $\mathbf{TK_{AB}}$ and retrieves the nonces $\mathbf{N_A}$ and $\mathbf{N_B}$. ***B*** checks if ***B***'s nonce in the message is the same as the nonce locally stored. The new permanent key, $\mathbf{PK_{AB}}$, can now be generated by ***B***, because it has all the parameters needed to generate this new key.
   B returns the last message Success_PFPEnd, which contains the encrypted $\mathbf{N_A}$.
   B -> **A**: *Succes_PFPEnd(E($g^{ab} \bmod p, N_A$))*
9. ***A*** receives the message Succes_PFPEnd and retrieves the nonce of ***A***, N_A. ***A*** checks this nonce with the local nonce and generates the new permanent key, $\mathbf{PK_{AB}}$.

Finally ***A*** and ***B*** have both generated the same permanent key, $\mathbf{PK_{AB}}$. Subsequently the permanent key identifier of ***A*** and ***B***, $\mathbf{PKID_{AB}}$ is generated, followed by storing these parameters and corresponding Magnet_ID into the local databases.

3.4.2 PFP Key Validation

The scenario given below describes the PPF validation scenario, where the pre-established permanent key between ***A*** and ***B*** has to be validated in order to be certain that the key is still valid. The key validation is accomplished by sending the devices' nonces to each other. If both devices are able to decrypt the messages with the pre-established permanent key and are able to confirm each others' nonces, they are certain that they have the correct key parameters and therefore will have the correct permanent key. This scenario is illustrated in Figure 3 and the the second section in this Figure is the key validation process:

1. ***A*** connects to ***B*** and ***B*** will send its Magnet_ID to ***A*** with the message Req_ID()
   B -> **A**: *Req_ID(SHA-1(g^bmod p))*
2. ***A*** receives the message Req_ID and seaches the Magnet_ID in local database, whether A can find ***B***'s identity. ***A*** does find the identity of ***B*** and returns the $\mathbf{PKID_{AB}}$ of the mutual key between ***A*** and ***B,*** and the corresponding Magnet_ID.
   A -> **B**: *Resp_ID($PKID_{AB}$ | SHA-1(g^a mod p))*
3. ***B*** receives the Resp_ID message. ***B*** also searches the database, whether it can find the ***A***'s Magnet_ID. ***B*** does find a corresponding match and will send in an encrypted manner a random value and its nonce, N_B, to ***A***.
   B -> **A**: *Req_Retrieve_PKID ($PKID_{AB}$ | E(PK_{AB}, N_B | Random))*

   The reason for appending a random value in the message is that the ciphertext will differ for each validation process. It will be more difficult for an attacker to generate a corresponding response message if the ciphertext differs each time.

4. ***A*** receives the message Req_Retrieve_PKID, decrypts the message and retrieves ***B's*** nonce and the random value. ***A*** then returns an encrypted Resp_Retrieve_PKID message containing ***A***'s nonce, ***B***'s nonce and the same random value generated by ***B***.
   A -> **B**: *Rsp_Retrieve_PKID ($PKID_{AB}$ | E(PK_{AB}, N_B | N_A | Random))*
5. ***B*** receives the message Rsp_Retrieve_PKID and decrypt it. ***B*** will retrieve ***A***'s nonce, ***B***'s nonce and the random value. ***B*** checks if the retrieved ***B***'s nonce and random value is the same as the local value. Subsequently ***B*** generates the new permanent key and checks if it is the same as the current permanent key. When successful, it sends encrypted the message Succes_RetrieveEnd, which contains the ***A***'s nonce, N_A and the random value.
   A -> **B**: *Success_RetrieveEnd($PKID_{AB}$ | E(PK_{AB}, N_A | Random))*
6. ***A*** receives the message Success_RetrieveEnd, decrypts the message and retrieves ***A***'s nonce. Subsequently ***A*** checks the nonce of the message with the nonce stored local.

If ***A*** and ***B*** have validated the nonces on both sides, they are certain that they have the right key.
The PKID is sent in all the encrypted messages. Nodes can check their PKID with the one in the message and detect which key is used for encryption. Besides the receiving node can validate if they have the right key.

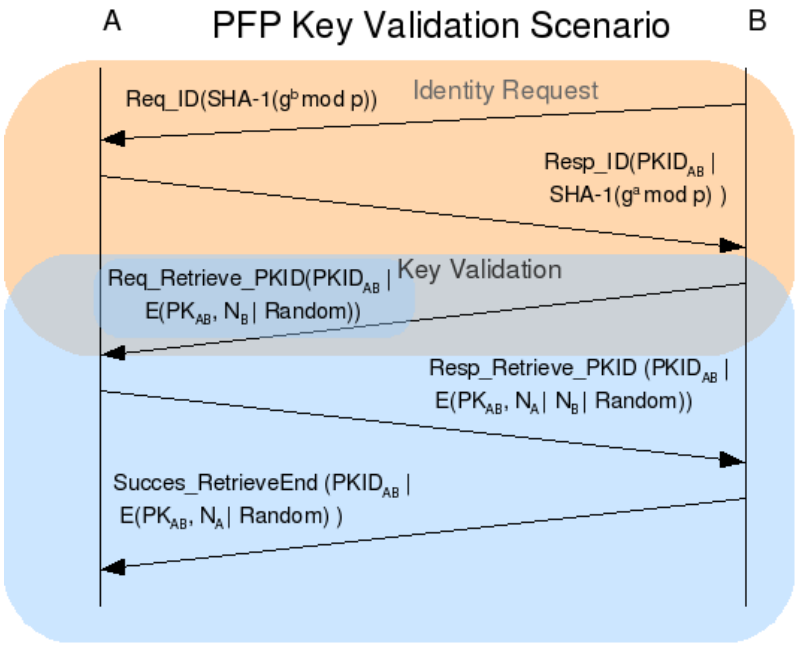

*Figure 3: The message sequence diagram of the PFP key validation scenario between device **A** and **B**.*

3.4.3 Analyzing the PFP scenarios

In this section we give an analytic description of how the middle-person vulnerability is prevented in the PFP. The first part of the key exchange scenario of the PFP is comparable to the Diffie-Hellman key exchange, because both devices publish their public key to each other and use the discrete logarithm problem for calculating the new established Diffie-Hellman key.

In order to reduce the risk of a middle-person attack, these public keys have to be authenticated. First we discuss how ***B***'s public key is authenticated below:

- ***B*** sends its public key, $g^b \bmod p$, an additional HMAC(**K**, $g^b \bmod p$) in the *Req_PFP_Exchange* message. As K was sent previously to the PAC and is only known to ***A*** and ***B***, the HMAC prevents that the attacker modifies the ***B***'s public key. ***A*** calculates the HMAC with the local **K** and validates the local HMAC with the one from the received message. If they are not the same, the received public key is not from ***B***.

A's public key is indirectly authenticated and the authenticity of ***A***'s public key is not accomplished by appending a HMAC field. ***A***'s public key is sent in the Resp_PFPBegin message, but it is authenticated after the Req_PFP_Exchange message. The Req_PFP_Exchange message also contains the ciphertext, $E(g^{ab} \bmod p, N_B \mid HMAC(K, g^b \bmod p))$, which has been encrypted with the key, $g^{ab} \bmod p$ (TK_{AB}). ***A*** can generate a new temporary key according to Diffie-Hellman, because ***A*** has already authenticated ***B***'s public key. ***A*** knows that ***B*** has established the correct ***A***'s public key, if ***A*** is able to decrypt the Req_PFP_Exchange message, retrieve the HMAC(**K**, $g^b \bmod p$) value and this HMAC(**K**, $g^b \bmod p$)

from the message is the same as the calculated HMAC(**K**, $g^b \bmod p$). If the HMACs are the same our indirect conclusions is that ***B*** has received the correct public key of ***A***. The next diagram illustrates how we come to this conclusion.

HMACs on ***A*** are the same	->	***A*** has successfully decrypted the *ciphertext*	->	***B*** has used the right key for encryption of the *ciphertext*	->	***B*** has successfully generated the right tempory key from ***A***'s public key and ***B***'s private key	->	***B*** has received the right public key of ***A***.

Confidentiality of the nonces is provided by encrypting these values with the Diffie-Hellman key. As nonces are confidential and are confirmed by the devices, the authenticity and integrity of these nonces are assured. The authenticity and integrity of the public keys are assured by the appended HMAC values in the protocol messages. All the sensitive values in the key-exchange scenario as well as in the validation scenario are protected by encryption. All the three criteria (confidentiality, authencity and integrity) for a secure protocol are met in the PFP, so we would consider it secure.

3.5 PFP Implementation

The message sequence of the PFP has been described in Section 3.4. This Section is more focused on the implementation of the PFP. The exact byte notation of message sequence, which is used in the implementation, can be found in [MAGNET-D433].

3.5.1 Operating Environment

The PFP application is written for the Linux Operating System and has been 'tested' on the INTEL i386 and ARM architectures. The PFP application can operate on top of the TCP layer, but it has mainly been developed for the Bluetooth Protocol stack. The application uses the connection-oriented L2CAP layer of the Bluetooth Protocol stack for communication. More information about the Bluetooth stack can be found in [BLUEZ].

It was decided in MAGNET that the PFP application should be executable on the Nokia 770 device. Figure 4 shows a picture of the Nokia 770. This small digital device is equipped with Bluetooth and 802.11 b/g wireless technology and is also provided with a small Linux distribution based on the Linux distribution Debian [DEBIAN].

In order to compile and execute the PFP application the following libraries are needed:

- BlueZ library, a library that implements the Bluetooth Wireless specification and can be used in Linux to establish communication with other Bluetooth devices.
- OpenSSL Library [OPENSSL], an open source library for implementing Secure Sockets Layers. This library also contains a general purpose cryptography library. The following functions of this cryptography library are used in the current PFP implementation:

- Encryption and decryption functions.
- All kinds of one-way functions, such as SHA-1, SHA-256 and HMAC.
- Diffie Hellman related functions for calculating the Diffie-Hellman keys.

- SQLite library [SQLITE], a small library that contains a small SQL database engine. When this engine is used, the databases are stored in a single file on the filesystem. A new SQLite database is created to store the permanent keys of the imprinted devices.
- GTK+ library [GTK], [GDK], a library for creating graphical user interfaces. The PFP application can be executed without the need of this library. The library is used in the PFP implementation to generate dialogs. Two dialogs are generated in the PFP application: a dialog to show the code, K to the user and another dialog for entering the code, K. In case the library is not used, the standard console input and output will be used.

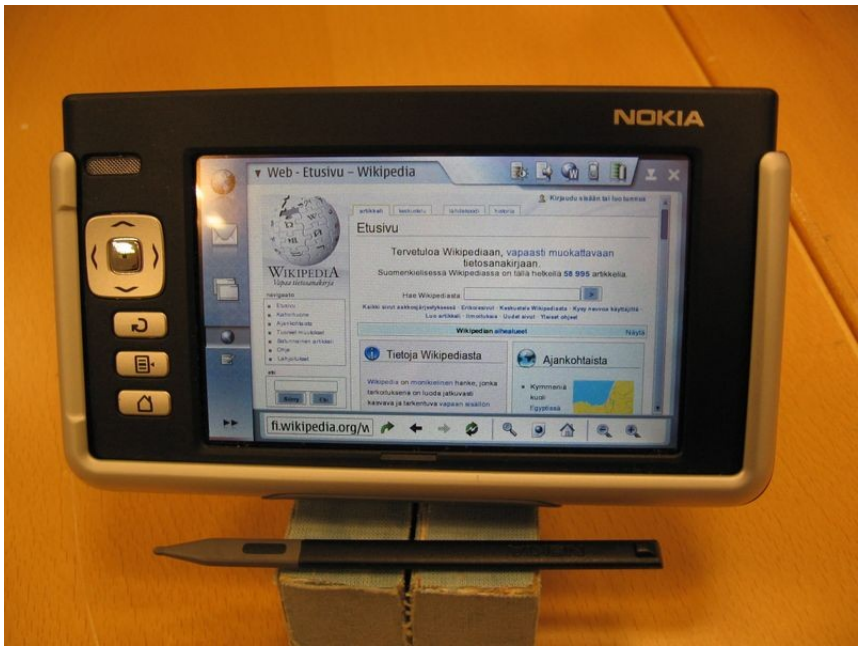

Figure 4: The Nokia 770

3.5.2 PFP Network Configuration

In order to get the PFP implementation working with Bluetooth, the main bluetooth configuration file of the Bluetooth Host Controller Interface Daemon (hcid) must be adapted. More information about hcid and the configuration can be found in the `hcid` and `hcid.conf` man page in Linux. We will describe how this configuration file is adapted below:

In the BlueZ protocol stack we must disable security, because we use the PFP for exchanging keys with other devices. Security on the physical layer for a bluetooth device is disabled as follows in the `/etc/bluetooth/hcid.conf` file:

```
# Authentication and Encryption
auth disable;
encrypt disable;
# Security Manager mode
security none;
```

In order to 'unhide' and to show this device for other devices, we must enable scanning and the same configuration file must be adapted as follows:

```
# Inquiring and Page scan
iscan enable;
pscan enable;
```

The command `hciconfig` can also be used, to enable scanning and/or to disable security. Only the settings are not restored, when restarting the computer. In case of the Nokia 770, this configuration file should be changed, because the device will disable the bluetooth network interface when it is in standby mode and when returning from standby mode it enables the bluetooth network interface and use the configuration file for the settings.

3.5.3 Compiling

Detailed information about compilation can be found at [LPROG]. This Section describes how we can compile the PFP source files and build the executables. The following must be done:

- First we must enter the directory, where the source files are stored.
- To build the executables:

  ```
  $ make
  ```

- To install the executables on your system:

  ```
  $ make install
  ```

Compiling of executables could not be done on the Nokia 770, because the device is small and does not have compiling tools installed. Therefore compiling must be done on another computer, which may have a different computer architecture and this is called cross-compiling.

Compilation for the Nokia 770 is more complex because:

- The Nokia 770 has an ARM processor, which is a different computer architecture than most other personal computers, therefore the executables are not compatible.
- The Nokia 770 is a small device and the user interface is more limited compared to other computers. Compilation of the source files on Nokia 770 requires a lot of user interaction.
- The Nokia 770 have not compiling tools installed.

A solution for this problem is compiling the source code on an other system, the host system, which can also compile source files for another computer architecture. This kind of compilation is called

cross-compiling. After the compilation the executables can be copied to the device and the Nokia 770 can execute the program.

For the Nokia 770 there are two tools available, which aid in compiling the executables for the Nokia 770:

- Scratchbox, which is a cross-compilation toolkit for the ARM processor. This toolkit enables you to compile and to emulate programs for the Nokia 770, which has an ARM processor, on the host system.
- Maemo, which is a development platform for creating applications for the Nokia 770. A Maemo package can be added to the Scratchbox, such that the host-system has the same environment as on the Nokia 770.

Of course the implementation should also be capable to run on devices that do not have the Hildon library. Then it could be compiled without the Hildon library and uses the GTK+ library as replacement.

3.5.4 Executing PFP on the Nokia 770

In the MAGNET documentation [MD4.3.3], a detailed description on can be found on how to start the PFP application on a normal computer. How to run the PFP application on the Nokia 770 is more complex and therefore it will be briefly discussed below:

- Install Scratchbox and Maemo on the host system.
- Install the sqlite3 package for the Nokia 770 on the host-system, or compile and install the SQLITE 3 library in your scratchbox environment on your host-system.
- Compile the PFP application for the ARM processor in the Scratchbox environment.
- On the Nokia:

  Download and install the following packages from the Maemo website:

  - sqlite3
  - xterm
  - dropbear
  - maemopad
- Copy the PFP application with executables to the Nokia 770 flashcard.
- On the Nokia 770:
  - Start xterm
  - Login as root with default password '*rootme'*:

    ```
    $ dbclient root@localhost
    ```

Edit `/etc/bluetooth/hcid.conf` with Maemopad or with an other editor to disable security and to enable scanning
(See Section 3.5.2)

Copy the PFP application into a directory:

```
$ cp -R /media/mmc1/pfp/* /<somedir>/
$ cd /<somedir>
```

First initialize the PFP application and reset the imprinting database:

```
$ ./pfp_init <configpath> <devicename>
```

After initialization we could run the PFP server or the PFP client:

The Server:
Register the PFP application as service:

```
$ ./magnet_sdp -p 1111 "Magnet PFP Imprinting"
```

Start the PFP server:

```
$ ./pfp -B –path <configpath>
```

Afterwards, unregister the PFP service:

```
$ unmagnet_sdp
```

OR

The Client:
Run the PFP application as client

```
$ pfp -b <mac address> --path <configpath>
```

3.5.5 Analyzing the PFP implementation

Each function in the implementation is sufficiently commented and it is not difficult to understand the basic structure of the implementation. However it is recommended to examine the PFP message sequence before looking at the code. The PFP-application reports a lot of debug information on the screen. Still there are minor changes made by the author and therefore this debug information is needed.

The implementation of the PFP is mainly developed for bluetooth and unlike the IP version it can register the PFP service for service discovery using the Service Discovery Protocol (SDP). After registering the service devices are able to find the PFP service.

The protocol message syntax of the PFP implementation is based on the Extensible Authentication Protocol (EAP) [RFC 3748], which is an authentication framework supporting multiple authentication methods. The PFP is not 100% compatible with this framework, because the last EAP_Success message is sent with additional data. More information about the exact protocol syntax can be found in MAGNET delivery and in the implementation.

How messages are encrypted with the Diffie-Hellman key is not exactly specified in the message sequence in Section 3.4.1. The Diffie-Hellman key length is quite large (1536 bits), because the discrete logarithm problem is used for encryption. In order to guarantee the same security as normal symmetric encryption, it is required to use the complete Diffie-Hellman key length for encryption.

The PFP implementation uses the AES-128-CBC algorithm for encryption and therefore only keys of 128-bits are supported. In the implementation only the first 128-bits of the 1536 bits Diffie-Hellman key length is derived and used for data encryption.

Although I have no proof that a subset of the Diffie-Hellman is not secure enough, I believe (and also the author of PFP agrees) that the complete 1536 bits should be used to increase overall security because of the following points:

- The first 128 bits of the Diffie-Hellman key may not be random enough. For an attacker it may be easier to find patterns in the Diffie-Hellman keys.
- It could be that possible keys are easier to filter out, given that the attacker knows the public keys on both sides.

An implementation problem of using the whole key for encryption, is that many well-known symmetric encryptions do not support such large key length, therefore a conversion to a much smaller symmetric key should be made. For example, a hash such as SHA-256 or SHA-1, could be used for this type of conversion and the author of the PFP application is making these changes into the PFP implementation.

3.5.6 Known problems in the implementation

The are some problems executing the PFP protocol:

- The PFP implementation uses the OPENSSL library, which supports the SHA-256 hash function. Older libraries than version 0.9.8a have not implemented the SHA-256 function. In order to get older versions working, the SHA-1 function must be used to generate the permanent key. To use the SHA-1 function instead of the SHA-256 the following line must be uncommented in the file `pfp.h`

  ```
  #define NOSHA256 1
  ```

  Moreover, all other devices have to use the SHA-1 function instead of the SHA-256 for compatibility reasons. The Nokia 770 only has version 0.9.7 of the OPENSSL library, therefore to get it working with other devices, the SHA-1 function must be used on all the devices.
- Some new compilers compile source files with stack protection. As the PFP implementation is still in development and in debugging phase, the application detects in some situations at run time a buffer overflow. Of course these buffer overflow bugs have to be fixed. To compile without stack protection the following flag has to be added to the compiler options:

  ```
  -fno-stack-protector
  ```
- A dialog might not be shown for entering or for displaying the code to the user. Instead the standard console input and output will then be used. One of the following situations causes this problem:

- The PFP application starts another program for showing the dialogs and cannot find this executable. To solve this problem, check if the executables *server-pfp/server-pfp* and *client-fp/client-pfp* are in the right directory and check if the executables in the functions server_pac and client_pac of the file `eap_pfp.c` refers to the right directory.

  function server_pac:
  ```
  strcpy(command, "<path to server-pfp>/server-pfp ");
  ```

  function client_pac:
  ```
  strcpy(command, "<path to client-pfp>/client-pfp ");
  ```

- To display the dialogs, a xserver must be running and we must set the DISPLAY variable to the xserver. For the PFP it makes no sense to run it elsewhere, so the DISPLAY variable will be set to the first xserver on the localhost:
  ```
  export DISPLAY=:0
  ```

4 The implementation of a GUI for the PFP

The PFP application can only be started from the command line, which is not very user friendly, so a simple graphical user-interface is needed. The Nokia 770 also has a limited user-interface: it only has a few buttons and a touchscreen, so starting the PFP from the command line interface (CLI) is not very practical, especially when a lot of command line options have to be provided. A new Graphical User Interface (GUI) therefore simplifies the process of discovering devices in the neighborhood and the start of the PFP application for exchanging keys between devices. This chapter describes the development of this GUI.

4.1 Requirements

Because the user interface for the PFP is not that complex, there is only a short list of requirements for the user-interface.

- The user interface should be very simple, so that within a few clicks we can complete the process of imprinting.
- The user interface should be executable on the Nokia 770 and also on personal computers with Linux.
- The user interface should be able to scan for Bluetooth devices in the neighborhood.
- The user interface should be able to start the PFP server.
- The user interface should be able to start the PFP client connecting to a PFP server application.
- The user interface should be able to reset the imprinting database.

4.2 Operating Environment

This section will describe the operating environment needed to start and compile the PFP application.

As the PFP implementation is written for Linux, the new GUI should also be written for Linux. The implementation of the user-interface must also be compatible with the Nokia 770 environment, which has a different computer architecture than most computers.

The new GUI implementation makes use of the GTK+ library, which is a library for creating user-interfaces and which is supported on most Linux Distributions and also on the Nokia 770. Additionally, a special graphical user interface library for the Nokia 770, the Hildon library, is supported by this new implementation.

This Hildon library [HILDON] adds the look and feel of the Nokia 770 desktop environment to the applications that uses the GTK+ library for the user-interface. Applications written for the GTK+ library can easily be adapted to support Hildon. The Hildon library is only supported on the Nokia 770 devices, but there is a development environment available, which can emulate the Nokia 770 device and supports this Hildon library.

The GUI also uses the Bluez Library and its BlueZ utilities for scanning devices in the neighborhood and for configuring the bluetooth adapter.

4.3 Scratchbox and Maemo

With this cross-compiling toolkit and development platform, it is easy to compile and test your application on different computer architectures. Detailed information on how to setup the Scratchbox and Maemo can be found at the Maemo website [MAEMO].

To run the application and let it display on the emulated Nokia screen on the X server:

Login into the scratchbox environment:

```
$ /scratchbox/login
```

Select the right target, in our case the target for the intel processor:

```
[sbox-SDK_ARMEL: ~] > sbox-config -st SDK_I386
[sbox-SDK_I386: ~] >
```

Start the Screen for the emulated Nokia and, create a scriptname for example: start-xephyr.sh

```
#!/bin/sh
$target="SDK_I386"
prefix=/scratchbox/users/{$LOGNAME}/targets/{$target}/usr
export LD_LIBRARYPATH=${prefix}/lib;
exec ${prefix}/bin/Xephyr :2 -host-cursor -screen 800x480x16 -dpi 96 -ac
```

Start the script in the 'normal' environment:

```
$ scripts/start-xephyr.sh
```

Set DISPLAY variable to the emulated Nokia screen and start the desktop environment:

```
$ export DISPLAY=:2
$ /scratchbox/login af-sb-init.sh start
```

Now applications can be started on the display with the look and feel of the the Desktop environment of the Nokia 770:

```
[sbox-SDK_I386: ] > run-stand-alone ./pfp-gui
```

A snapshot of the screen on the host computer is shown in *Figure 5* and a picture of the Nokia 770 running the GUI is shown in Figure 6.

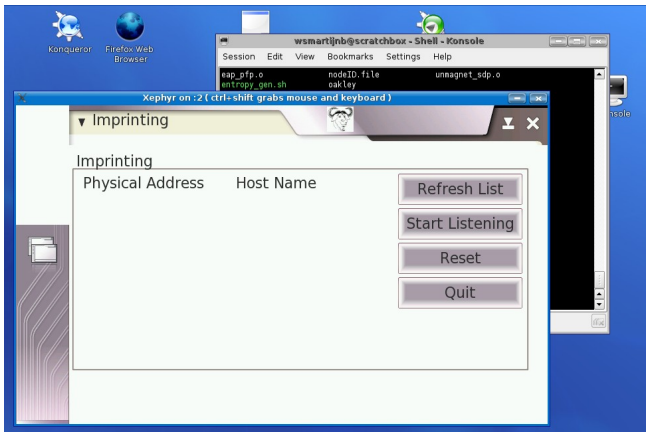

Figure 5: The developed GUI application in the MAEMO environment of the host-system.

Figure 6: The developed GUI application on the Nokia 770

4.4 Starting the GUI

In order to run the GUI on a personal computer with Linux as operating system, the command `pfp-gui` can be started:

```
$ ./pfp-gui
```

In order to get the GUI working on the Nokia 770:

- The PFP executables have to be copied to the Nokia 770.
- The PFP application is not available in the menu of the Nokia's desktop environment, and so a command line is needed to start the application.
- The configuration files of the hcid have to be adapted, so the device is enabled for scanning. (See Section 3.5.2, PFP Network Configuration)

A solution for these points would be to make a package for the Nokia, which adds the GUI application to the menu and sets up the configuration files for the blue adapter.

4.5 The Implementation

The user-interface can execute different commands:

- Scanning for devices, which is started by the button Refresh List, will scan for bluetooth devices in the Neighborhood using the program `hcitool`.
  An example output of the command `hcitool scan` is given below:

```
$ hcitool scan

Scanning ...
        00:14:A4:D4:84:56       connor-0
        00:12:D1:8E:7A:22       WMC_IT1
        00:12:D1:8E:74:6E       WMC_IT2
$
```

- Start Listening for PFP imprinting performed by the button "Start Listening", will register the PFP service and start the PFP server with the following programs:

```
$ magnet_sdp -p 1111 "Magnet PFP Imprinting"
$ pfp -B –path <configpath>
$ unmagnet_sdp
```

- Stops Listening, this will kill the child process, which executes the PFP program for listening the PFP imprinting.

- Connect to another device to start imprinting will be performed if the user double clicks on an item of the list of devices. The following program with arguments will be executed:
  `$ pfp -b <mac address> --path <configpath>`

- Initializing the PFP protocol and resetting the imprinting database will be performed by the button Reset and the following program will be executed:
  `$ ./pfp_init <configpath> <devicename>`

4.6 Overview of the implementation

The implementation basically consists of three parts:

- **GUI:** This part creates the user-interface and has functions for displaying and showing dialog windows. Furthermore it has functions to disable/enable different items in the user-interface, to create a list of imprinting devices and to add/remove items to the list.

- **Events:** This contains the functions, which are called when an event occurs. For example when a button is clicked or when an item on the list is clicked.

- **Commands:** This contains all the executable actions. For example, it contains functions for starting another process, for starting PFP imprinting and for scanning devices.

If an event occurs, it will execute the corresponding assigned callback function. Execution of this callback function must be handled quickly and should not block for too long, otherwise the thread can not react to other events. In the GUI implemenation, the blocking thread problem is solved by creating a new thread handling this event. The older thread will end the callback function and can react to other events.

4.7 Analyzing the GUI Implementation

The GUI implementation performs its task well, however it is only able to scan bluetooth devices. As only the link-layer has the functionality to discover link connections, scanning devices using IP-layer is difficult and has to be researched. The UCL layer designed in MAGNET may solve this problem, but how this can be fit into PFP and the PFP-GUI still has to be researched [MD4.3.3] (see also problem described in Section 3.5.5)

5 Transitive PFP Imprinting

This Chapter describes an extension for the PFP for minimizing the number of user-interactions which is necessary to establish keys among pairs of devices. This extension is called Transitive PFP imprinting (TPFP). We give a description and specification of the TPFP in this Chapter.

The PFP requires the user to authenticate with the PAC each time, when a pair of devices in a network wishes to exchange keys. In our case the PAC will be the user and each imprinting procedure the user has to enter a code into one of the devices. When a network consists of N devices, a total of $N \cdot (N-1)/2$ pairwise keys have to be established and the user is bothered each time. For example if we only have 8 devices in our network, the user must interact 8*7/2=28 times with his devices to establish a full secure network.

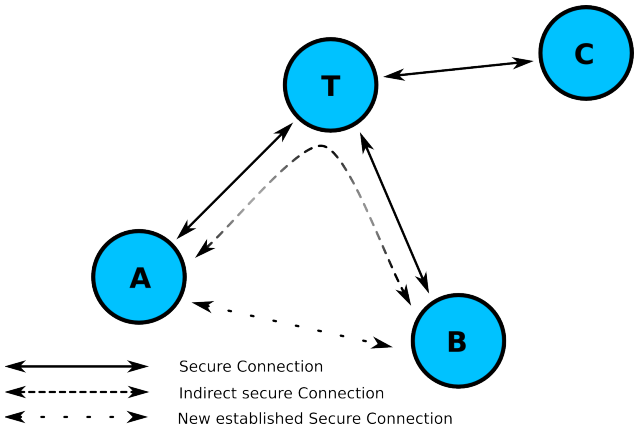

*Figure 7: Transitive imprinting, where two devices, **A** and **B** are able to establish a secure association with each other via an intermediate device, **T**.*

A solution for this user-authentication problem is to exchange keys via intermediate device with which both devices have already established a secure association. The intermediate device has not the role of a central device, but acts transitively and forwards the key exchange between the two parties wanting to imprint. This kind of imprinting has been introduced in MAGNET as transitive imprinting. Any specification about transitive imprinting procedure has not been realized and therefore this Chapter specifies a transitive imprinting protocol based on the existing PFP.

Figure 7 illustrates the general concept of transitive imprinting. We give a more description about this transitive imprinting procedure below:

Assuming that devices ***A*** and ***B*** wishes to establish a pairwise key and already having a secure association with common device ***T.*** ***T*** acts as an intermediate device for establishing keys between ***A***

and ***B*** [1]. ***T*** establishes a secure connection with ***A*** and ***B*,** therefore ***A*** and ***B*** have a mutual secure indirect connection and can exchange key parameters indirectly via ***T***.

After the exchange of the nonces and the public keys, a new key can be generated in the same manner as in the PFP application (Section 3.3). User-interaction for authentication is not needed anymore, because the end-devices ***A*** and ***B*** are able to authenticate with ***T***. The devices are able to detect if a received encrypted message is not coming from ***T*** or if it is modified.

5.1 Transitive Imprinting in the PN

Transitive imprinting could theoretically be used in any type of network. This Section describes how the transitive imprinting fits into the initialization of a personal network (PN). In the rest of the Chapter we will use the word node if we mean a device, which is part of a secure network or wants to be part of this secure network.

The first secure association of a node in the PN must always be established by manual imprinting, because the node have not a trust relationship with any node yet and therefore authentication can only be provided by the user. After the manual imprinting, the node transitive imprints with all the possible nodes and in the end it registers itself into the PN.

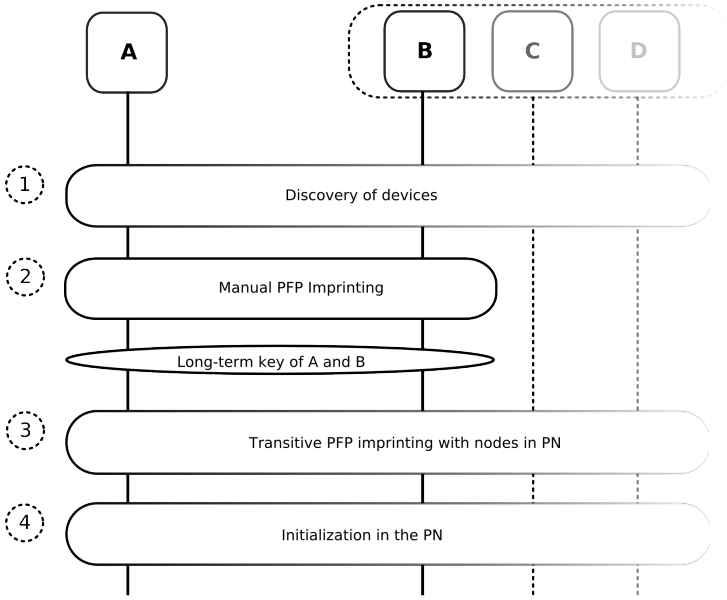

Figure 8: Overview of imprinting a new node A with nodes in the PN.

1 Of course A and C, or B and C can also establish a key with intermediate device T

Figure 8 gives an overview of the PN initialization:

Assuming that node ***B*** has already registered itself into the personal network and therefore has already secure associations with other nodes in the PN. The user wishes to manual imprint node ***A*** with node ***B*** and subsequently ***A*** can registered itself via ***B*** into the PN:

1. **Discovery of devices:**
   A discovers nodes that are in range and these nodes are potential partners for manual imprinting. In our case, ***A*** discovers that ***B*** is suitable for imprinting and displays ***B***'s address to the user. This first discovery procedure is accomplished by using the PFP GUI described in Chapter 4.
2. **Manual PFP imprinting:**
   The user starts the manual imprinting procedure on the both nodes: he starts the PFP server on ***B*** and he connects the PFP client, ***A***, to ***B***. After this PFP imprinting procedure, ***A*** and ***B*** have established a pairwise key.
3. **Transitive PFP imprinting with nodes in PN:**
   After the imprinting via intermediate node ***B, A*** can transitive imprint with all the other nodes in the PN, but first ***A*** has to discover ***B***'s imprinted nodes. The message sequence about the discovery can be found in Section 5.2. After this discovery, ***A*** establishes secure associations with these imprinted nodes and will continue to discover imprinted node and transitive imprint with them until ***A*** has trust relationships with all the possible nodes.
4. **Initialization in the PN:**
   After the establishment of secure connections, ***A*** is able to register itself into personal network. How this is exactly been accomplished is not in the scope of this report and therefore it will not be discussed.

5.2 Example of Transitive Imprinting

An example network illustrating the transitive imprinting is given in Figure 10. ***A*** represents a node, registering itself into the personal network. Other nodes already have secure connection with some of the other nodes.

The following assumptions are made, when discussing the example of Figure 10:

- All the nodes can communicate to the other nodes in this example network. How communication is accomplished, is not important for this discussion.
- The pair of nodes with a solid arrow between them, are the pairs having a secure association.
- The dotted arrows between ***A*** and other nodes are the secure associations, which are created in this example.

Figure 10 illustrates the sequence of transitive imprinting with all the other nodes in this network:

1. First node ***A*** manual imprints with node ***B*** and then ***A*** asks ***B***, which nodes ***B*** already has a secure association with. The example network tells us that ***B*** has a trust relationship with ***C*** and

A. A secure association between ***A*** and ***B*** already exists and therefore ***B*** will only reply the result: ***C.***

2. ***A*** has learned the existence of secure association between ***B*** and ***C*** and therefore ***A*** is able to transitive imprint with ***C*** via ***B***. After the imprinting with ***C***, ***A*** continues discovering the imprinted nodes and learns from ***C*** that ***A*** can transitive imprint with ***D*** and ***E***.
3. ***A*** knows that ***C*** already has imprinted with node ***D*** and therefore it can transitive imprinting with ***D*** via ***C***.
4. ***A*** also knows that ***C*** has already imprinted with node ***E*** and therefore it can transitive imprint with ***E*** via ***C***.

Finally ***A*** has secure associations with all the possible nodes. The message sequence in the discovery of imprinted nodes is described in the next section.

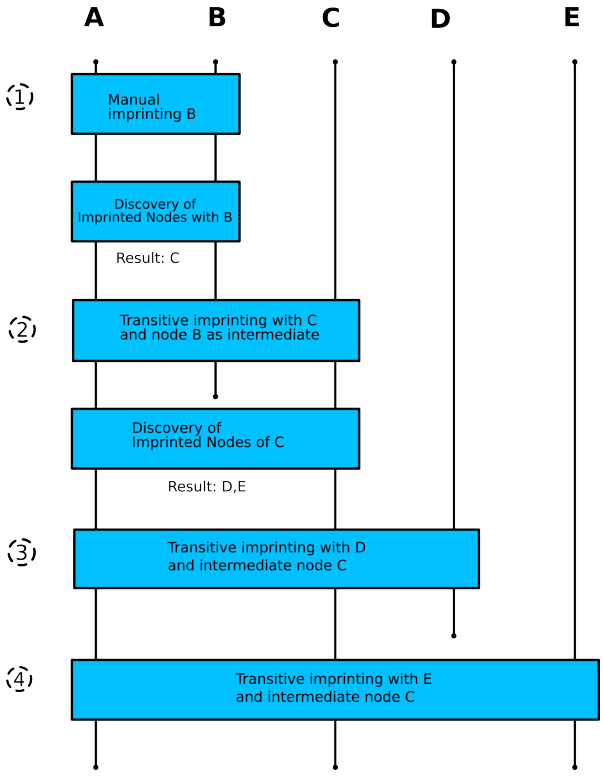

Figure 9: The imprinting and discovery sequence of node A in the example network illustrated in Figure 10

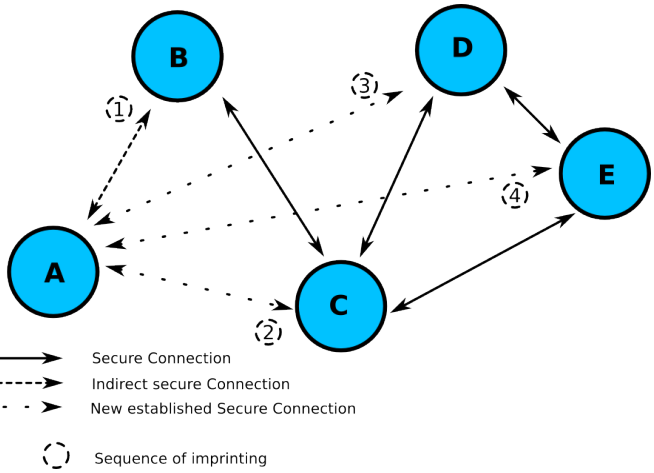

Figure 10: An example of network, where some nodes have secure associations with other nodes and where the sequence of imprinting with node A is illustrated.

5.3 Discovery of transitive and imprinted nodes

In the process of transitive imprinting with other nodes, the initiator must be informed which nodes already have a secure association with this intermediate node. The procedure of this kind of discovery is described in this section and the message sequence of such discovery is illustrated in Figure 11.

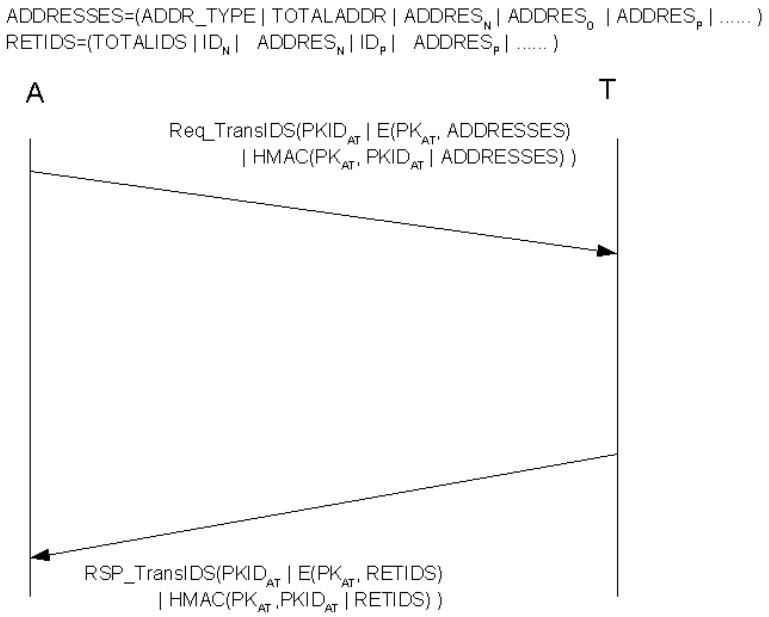

Figure 11: Message sequence for the discovery of imprinted nodes.

Node ***A*** is the initiator and ***T*** is the intermediate node and these nodes have a secure association with each other. The message sequence of the discovery of imprinted nodes is described below:

First ***A*** scans the for nodes and retrieves a list of all the devices in the neighborhood and their corresponding addresses.

1. ***A*** must determine, which nodes are capable of transitive imprinting. ***A*** knows that ***T*** has the transitive imprinting service enabled and sends the message Req_TransIDS to node ***T***.

   A -> **T**: *Req_TransIDS (PKID$_{AT}$ | E(PK$_{AT}$, ADDRESSES) | HMAC(PK$_{AT}$, PKID$_{AT}$ | ADDRESSES))*
   This message is encrypted with the pairwise key between ***A*** and ***T***, **PK$_{AT}$**. The message contains the addresses with which ***A*** wishes to establish a secure association. These addresses corresponds to the nodes in the neighborhood. The type of addresses can be indicated by the message field ADDR_TYPE.

2. When ***T*** retrieves the message Req_TransIDS, ***T*** decrypts the message and retrieves the addresses from the message. Subsequently ***T*** searches these addresses in the local database, whether it has already imprinted with these addresses or not. The result is the addresses / Magnet_ID pairs of the nodes, where ***T*** has imprinted with.

   In case there is an empty address field in the message and the field TOTALADDR is zero, T will reply with *all* the imprinted addresses and corresponding Magnet_ID pairs.

   Subsequently ***T*** returns the encrypted message Rsp_TransIDS. This message contains the address and Magnet_ID pairs retrieved by the local database.

   T -> **A**: *RSP_TransIDS ($PKID_{AT}$ | $E(PK_{AT}, RETIDS)$ | $HMAC(PK_{AT}, PKID_{AT} | RETIDS)$)*
3. ***A*** receives the message RSP_TransIDS, decrypts the message and retrieves the address / Magnet_ID pairs. ***A*** has now learned which nodes can do imprinting via intermediate node ***T***.

   Assurance of the authenticity of the messages is achieved by appending a HMAC. The receiving node discards the message if the retrieved HMAC is not the same as the calculated HMAC.

Advantages of discovery procedure described above is that:

- The client, ***A*** is responsible for giving the addresses to ***T*** and ***T*** only returns these addresses and corresponding MAGNET_IDs.
- This discovery procedure is also used to map addresses to Magnet_IDs. Received addresses and corresponding Magnet_IDs are stored into ***A***'s local database such that ***A*** can do mapping between addresses and Magnet_IDs.

A disadvantage of this discovery procedure is that:

- When ***A*** wishes to transitive imprinting with for example node ***Z***. ***A*** is not certain, which node can act as intermediate node with ***Z*** and therefore ***Z*** has to 'ask' the all imprinting nodes until it finds such intermediate node.

5.4 Message Sequence of transitive PFP imprinting

This section describes, the message sequence of the proposed PFP transitive imprinting procedure. The following assumptions have been made:

- Node ***A*** and Node ***B*** are the nodes wanting to establish a secure association.
- Node ***T*** is the intermediate node with which ***A*** and ***B*** have already imprinted.
- The pair ***AT*** and the pair ***BT*** already have a secure assocation with each other
- ***B*** and ***T***, have the service for transitive imprinting enabled.
- ***A***, is a client that supports transitive imprinting.

Figure 12 illustrates the message sequence of transitive PFP imprinting and this message exchange is described below:

1. ***A***, the TPFP client, connects to node ***T*** and than ***T*** sends an empty Req_TransBegin message to ***A*** and the TPFP procedure is started.
   T -> **A**: *Req_TransBegin()*
2. ***A*** receives the empty Req_TransBegin message from ***T***, generates a new nonce, N_a and returns the message Resp_TransBegin. This message is encrypted with the key $\mathbf{PK_{AT}}$ and contains the values: N_A, the source and the destination Magnet_Ids (ID_A and ID_B) and ***A***'s public key (g^a mod p).

   T -> **A**: $Resp\_TransBegin(PKID_{AT} | E(PK_{AT}, ID_A | ID_B | N_A) | g^a \bmod p | HMAC(PK_{AT} , g^a \bmod p))$
3. ***T*** receives, decrypts the message and retrieves the IDs.
4. ***T*** checks if it has already imprinted with ***B*** and if so it maps the ID_B into ***B***'s address and connects to the ***B***'s corresponding address. Subsequently ***B*** sends a empty Req_TransBegin message to ***T***.

   B -> **T**: Req_TransBegin()
5. ***T*** forwards the unencrypted message Resp_TransBegin from ***A*** and encrypts it with the key of ***T*** and ***B***, $\mathbf{PK_{TB}}$.

   T -> **B**: $Resp\_TransBegin(PKID_{TB} | E(PK_{TB}, ID_A | ID_B | N_A) | g^a \bmod p | HMAC(PK_{BT} , g^a \bmod p))$
6. ***B*** receives the message Resp_TransBegin from ***T*** and retrieves the values from the message. Subsequently ***B*** generates a new nonce, N_B, and can now calculate the temporary key between ***A*** and ***B***, $\mathbf{TK_{AB}}$, according to Diffie-Hellman. Afterwards ***B*** generates a new permanent key from the nonces and the temporary key. (see Section 3.3, PFP key generation)
7. ***B*** replies the message Req_Transcont to ***T*** and this message contains ***B***'s nonce and also the ***B***'s public key. Subsequently ***A***'s nonce is added to the message for confirmation. Then the message is sendt by ***B*** and is forwarded by ***T*** in the same manner as the Resp_Transbegin message.

   B -> **T**: $Req\_TransCont(PKID_{TB} | E(PK_{TB} , ID_B | ID_A | N_A | N_B) | g^b \bmod p | HMAC(PK_{BT} , g^b \bmod p))$
   T -> **A**: $Req\_TransCont(PKID_{TA} | E(PK_{TA} , ID_B | ID_A | N_A | N_B) | g^b \bmod p | HMAC(PK_{AT} , g^b \bmod p))$
8. When ***A*** receives this Req_Transcont message, ***A*** can generate the temporary key $\mathbf{TK_{AB}}$ and subsequently generates the permanent key $\mathbf{PK_{AB}}$ in the same manner as ***B*** did. (see Section 3.3, key generation)
9. For confirmation of ***B***'s nonce, ***A*** indirectly sends via ***T*** the message Resp_TransCont and encrypts the nonce with the new generated permanent key, $\mathbf{PK_{AB}}$.

   A -> **T**: $Resp\_TransCont(PKID_{AT} | E(PK_{AT},ID_A | ID_B) | E(PK_{AB}, N_B))$
   T -> **B**: $Resp\_TransCont(PKID_{BT} | E(PK_{BT},ID_A | ID_B) | E(PK_{AB}, N_B))$
10. ***B*** receives the message and decrypts it with his newly generated permanent key, $\mathbf{PK_{AB}}$ and checks if the receive nonce as the same as the ***B***'s nonce, which has been sent. Finally, ***B*** indirectly sends a EAP_Success message to ***A*** and the key-exchange is successfully ended.

B -> **T**: *EAP_Success()*
T -> **A**: *EAP_Success()*

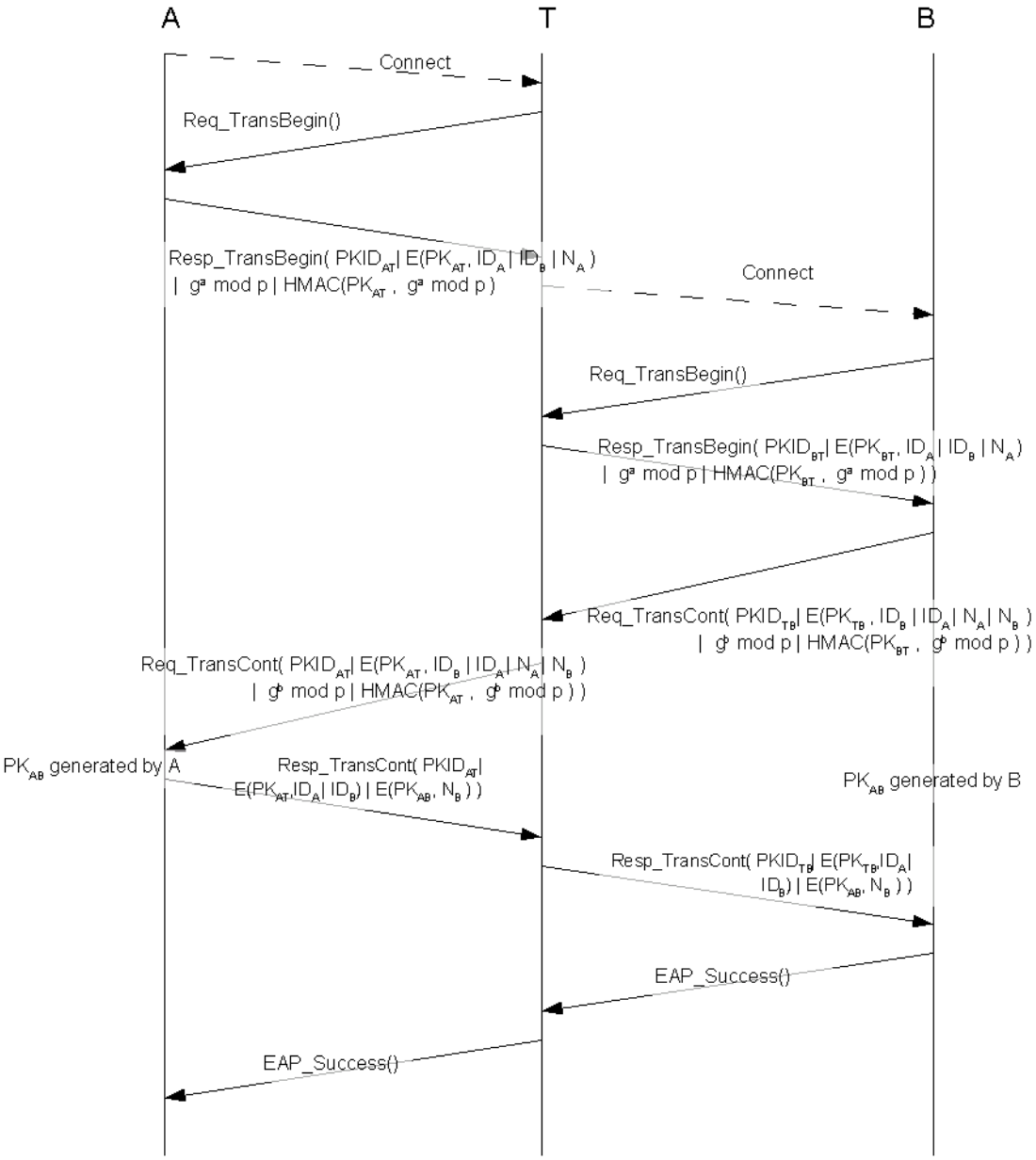

Figure 12: The message sequence of the transitive PFP

5.5 The PFP Implementation

There are three parties involved in the imprinting procedure and therefore also the TPFP application should also be capable to act as three separate instances: The TPFP client, the TPFP service as intermediate node and the TPFP service as imprinting node.

Although the implementation is far from completed, a beginning of the TPFP application has been developed:

- Mapping Magnet ID's to address. In PFP and TPFP nodes are identified by an Magnet_ID and mapping to physical and / or IP-address is required for connecting to other devices (see Section 3.4). Mapping is accomplished by adding an extra database, which contains all the Magnet_IDS and corresponding addresses. Furthermore the database contains entries of the transitive node and their discovered imprinted nodes.
- State diagram and events. Each separate instance has its own state-diagram and for each instance an implementation of its corresponding state-machine has been implemented. The state-machine changes from states when certain events occur. Each message has its own corresponding event and a number of additional events exist for error handling.
- The basic structure of implementation. The basic structure of the application has been implemented: several files already containing some basic functionality. For example: functions for initialization and making a blue tooth connection, functions for generating keys. A number of these functions are copied from the original PFP and adapted to fit in the TPFP context.

5.5.1 Further Work

Although some basic framework for the TPFP exists, much of the transitive imprinting has to be developed. The following points still have to be implemented before the TPFP is operational:

- Functions that handles received messages. Each distinguish message is handled by different function.
- Functions that sends a specific messages. Each type of message is differently created and sent and most of these messages must be encrypted.
- An implementation that discovers transitive node and their imprinted nodes, and adds these entries into the mapping database. In the current implementation we have not implemented any functionality yet for discovering the imprinted nodes.
- Testing the TPFP application.

5.5.2 Analyzing the TPFP

In the TPFP the three criteria for a secure protocol are provided by the same security mechanisms applied in the PFP. Therefore we will not discuss these three criteria in detail. However the HMACs of the public keys are now encrypted with the pairwise key of the imprinted node and intermediate node. The three devices involved in the transitive imprinting procedure do not need a PAC for authentication and the user is not involved anymore.

6 Conclusion

This thesis describes the PFP and two contributions developed and / or specified in this bachelor assignment. The first contribution is a GUI for the PFP and the second contributions is a specification of an extension for the PFP, called TPFP.

Security

One of the thesis's objectives was to analyze the security aspects of the PFP. Three criteria for a secure protocol should be met: confidentiality, authentication and integrity. After analyzing the PFP message exchange, we conclude that the PFP provides enough security mechanism for ensuring each of these three criteria in the protocol. Also in the TPFP specification, we have also carefully applied these security mechanism to limit vulnerability for attacks.

Implementation

Although the PFP application operates well and can be used to exchange keys successfully, still it cannot be deployed on large scale. The reason is that the source code needs carefully analyzed against security flaws by several security experts to prevent the vulnerability of the implementation as much as possible.
The PFP and PFP-GUI operates well on the bluetooth protocol stack and the PFP is also capable to operate on top of the IP layer. However there are a number of problems, when we want to scan devices in the neighborhood on IP layer communication and solutions for these problems have to be researched.
A basic framework for the TPFP implementation already exists, but most of the functionality has to be implemented. Furthermore we would recommend to implement TPFP using both the L2CAP bluetooth and TCP sockets. As the PFP uses both sockets and these sockets are comparable, not much of the TPFP implementation has to be adapted to have both functionalities.

Scalability

Scalability of the user-interactions for key-establishment in the PFP is a problem. The TPFP specified in this thesis solves the scalability problem by using a trusted intermediate device for imprinting, so that no user-interaction is required. User-interaction is only needed for the first imprinting and afterwards devices can imprint by using the TPFP service. The combination of PFP, GUI and TPFP simplifies the process of imprinting between several pairs of devices.
However much overhead is spent on key establishment when using pairwise keys, because the existence of many pairs of devices. However further research have to be done on how we can limit overhead for key establishment. A solution is to establish keys on demand and only a minimum of keys are established. However minimum pairwise secure associations limits the connectivity to other devices.

7 References

[BLUEZ] "BlueZ - Official Linux Bluetooth protocol stack", http://www.bluez.org/,

[DEBIAN] " The Universal Operating System", http://www.debian.org/,

[FS-PC] Niels Ferguson and Bruce Schneier, "Practical Cryptography", John Wiley & Sons, april 2003

[GDK] GDK Reference Manual (GDK 2.6.10-1.osso11), https://stage.maemo.org/svn/maemo/projects/haf/doc/api/gdk/index.html

[GTK] GTK+ Reference Manual (GTK+ 2.6.10-1.osso11), https://stage.maemo.org/svn/maemo/projects/haf/doc/api/gtk/index.html

[HILDON] Hildon Reference Manual (0.12.14), http://www.maemo.org/platform/docs/api-bora/hildon-docs/hildon-libs/index.html

[HU-NIPV6] Christian Huitema, "IPv6 The New Internet Protocol"

[LPROG] John Goerzen,"Linux Programming Bible", April 2000, Hungry minds

[MAEMO] Application Development platform for Nokia Internet Tablet Products, http://www.maemo.org,

[MAGNET] "IST - My personal Adaptive Global NET", http://www.ist-magnet.org/,

[MD2.1.2] Marina Pertova, Matthias Wellens, Simon Oosthoek, etc. , "Overall secure PN architecture", MAGNET 2.1.2, 31st October 2005.

[MD4.3.3] Neeli Prasad, M. Imine, Sepideh Fouladgar, etc. , "Implementation and Evaluation of the Level Security Architecture", MAGNET 4.3.3, December 31st 2005.

[MI-TSS] E.F. Michels, "Lecture Nodes Telematics System Security", University of Twente, November 2003

[NH-ADPN] I.G. Niemegeers and S.M. Heemstra de Groot, "Research Issues in Ad-Hoc Distributed Personal Networks", Kluwer International Journal of Wireless and Personal Communications, Vol 26, No. 2-3, pp 149-167, 2003

[RFC 3748] B. Aloha, L. Blunk, J. Volbrecht and etc, "Extensible Authentication Protocol (EAP)", RFC 3748, June 2004

[SBOX] Scratchbox, http://www.scratchbox.org/,

8 List of Abbreviations

AES	Advanced Encryption Standard
CLI	Command Line Interface
GUI	Graphical User Interface
DH	Diffie-Hellman
EAP	Extensible Authentication Protocol
GTK	The GIMP Toolkit
HMAC	The Keyed-Hash Message Authentication Code
IP	Internet Protocol
MAC	Message Authentication Code
MD5	Message Digest Algorithm 5
NFS	Network File System
RFC	Request for Comments
RSA	Rivest, Shamir, Adleman
PAC	Proximity Authentication Channel
PAN	Personal Area Network
PN	Personal Network
PFP	Personal Area Network Formation Protocol
SDP	Service Discovery Protocol
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TK	Temporary Key
TPFP	Transitive PFP

9 Appendix A: Notation Message Sequence

Notation:	**Description:**
p	a very large prime
g	g is 2=<g=<p-2 Where the following properties must hold, such that: for every i, 1<i<=p-1 there must be a power j, that $i=g^j \bmod p$ holds.
a	The private key of device A.
$g^a \bmod p$	The public key of device A.
$g^{ab} \bmod p$: or TK_{AB}	The Diffie Hellman shared key between devices A and B.
SHA1()	160 bits hash function
SHA1($g^a \bmod p$) *or* ID_A	Magnet_ID, which is used to identify the devices
SHA256()	a 256-bits hash function.
$SHA256()_{32}$	The first 32 bits of the 256-bits one-way hash function.
N_A:	Number used once, random value generated by A. When A and another device B, this value is used to generate a permanent key between A and B.
PK_{AB}	Permanent key between A and B, is generated of N_A ,N_B and $g^{ab} \bmod p$. PK_{AB}=SHA_256 (E($g^{ab} \bmod p$, $N_A \mid N_B$) when $N_A <= N_B$ *or* PK_{AB}=SHA_256 (E($g^{ab} \bmod p$, $N_B \mid N_A$) when $N_B < N_A$
$PKID_{AB}$	The identifier for the permanent key. It is used to identify the permanent key without publishing the key itself. The PKID is generated from N_A N_B, where N_A and N_B are used to generate the permanent key. $PKID_{AB}$=SHA_256 (N_A , N_B)$_{32}$ when $N_A <= N_B$ *or* $PKID_{AB}$=SHA_256 (N_B , N_A)$_{32}$ when $N_B <= N_A$
$M_1 \mid M_2 \mid M_3$	Conjunction of the messages M_1, M_2, M_3.
$E(K,M_1\|M_2\|M_3)$	Messages M_1,M_2,M_3 encrypted with key K.
HMAC(M,K)	a hashed MAC encryption of message, M with key, K
AnyMessage(M_1)	Message AnyMessage with data M_1