Implementation of Key Establishment Protocol based on Ultrasound Distance Bounding

Feng Kai

Kongens Lyngby
29th July 2006

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

This thesis is to study the threat in the key establishment protocols in wireless networks and to implement a secure key establishment protocol in embedded system. The confidentiality, integrity and authenticity becomes the major issues in the security of the key establishment protocols. Several existing solutions have been studied in this work. They use the help from public key cryptography or DH method to deal with confidentiality. Message digest provide integrity. The authenticity is provided by several different ways. Distance bounding authenticator has been focused in this work. The key establishment protocol with distance bounding is thoroughly studied and analyzed. The implementation of this protocol is also described. The test shows the protocol implementation in this work is secure.

Preface

This thesis was prepared at Informatics and Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering.

The thesis deals with secure key establishment protocol and implementation issues. The main focus is the Integrity and Authenticity of the message exchange.

Lyngby, July 2006

Feng Kai

Acknowledgements

I should thank my supervisor Srdjan Capkun who provided the guidance of my M.Sc. thesis at IMM DTU. I should also thank the Ph.D. student Kasper Bonne Rasmussen who helped me along the development.

I should thanks to Liu Ling, Hao Jia, Yang Jing, Lauge Ronnow, Lars Fox and Kasper Bonne Rasmussen who have read this thesis and provided feedback and suggestions for the refinement.

Special thanks to Mads Ingerslew Ingwar, Kasper Lindgaard Jensen and Soeren Kristian Jensen who studied my source code and discovered a bug.

Abbreviation

ADC	Analog-Digital Converter
AVR	Advanced Virtual RISC
ATPG	Automatic Test Pattern Generator
CBC	Cipher Block Chaining
CIA	Confidentiality, Integrity and Availability
CMOS	Complementary Metal-Oxide Semiconductor
CRC	Cyclic redundancy check
CTL	Computation tree logic
DES	Data Encryption Standard
DUT	Device Under Test
ECC	Error correction code
EMI	Electromagnetic interference
FIFO	First-in-First-out
FSA	Finite State Automaton
FSM	Finite State Machine
IR	Infrared
ISM	Industrial, Scientific and Medical
ISP	In-System Programmable
IV	Initialization Vector
JTAG	Joint Test Action Group (IEEE 1149.1 standard)
LFSRs	Linear Feedback Shift Registers
MAC	Message Authentication Code
MD	Message Digest
MITM	Man-In-The-Middle
MSB	Most Significant Bit
PWM	Pulse Width Modulation

RSSI	Received Signal Strength Indication
UART	Universal asynchronous receiver transmitter
US	Ultrasound
SECDED	Single Error Correction and Double Error Detection
SHA	Secure Hash Algorithm
SRD	Short Range Device
TCP	Transfer Control Protocol
VHDL	VHSIC Hardware Description Language
WEP	Wireless Equivalent Privacy
WLAN	Wireless Local Area Network

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background

The benefit from a laptop computer is not only the light weight, but also the mobility. The user can bring the computer inside a hand bag to participate in a conference. Without turning off the power, the user can freely move from a laboratory to a library; in the mean while, a session that downloads an important circuit diagram from another computer can be kept without being disturbed. The handhold devices such as PDAs, are also equipped with wireless LAN network and Bluetooth. With less powerful processor unit, such device is still capable of playing movies, read 'pdf' files and share information. The mobile phones with infrared and Bluetooth technology are not new. The user can access Internet directly from mobile phone without carrying a PC. With Bluetooth, the user can explore friends in the surroundings, exchange lecture notes and etc, without use of public wireless network.

Security problems can be found in any networks, especially in peer-to-peer network. In peer-to-peer network, there is no fixed network topology and no security infrastructure available. Eavesdropping is very easy on a wireless connection, since it is basically a broadcast channel. To protect the information, the user has to execute a protocol that authenticate each other, and develop a temporary secret key (as session key) between them. Further communication can be protected using encryption with the session key.

Nearly every wireless network technologies all have security measures. In 802.11 wireless local area network (WLAN), the information is protected with *WEP (Wireless Equivalent Privacy)*. The data packets are encrypted, and check sum is calculated to detect any modification in the packet. In Bluetooth, two parties can establish an *Initialization Key*, which is used further for *Link Key* generation.

Using a key to encrypt any information seems to be secure. However this is not the end of the story.

1.2 Existing problems

1.2.1 Few words about 802.11 and WEP

In 802.11 WLAN, WEP is used to have a security level equivalent to wired connections. It uses CRC (Cyclic redundancy check) for check summing and RC4 stream cipher. It first computes the CRC code over a message, and append at the end of the message to form a plain text. An Initialization Vector (IV) is then randomly chosen. The IV is 24 bits long. A key stream is computed using RC4 from IV and a shared key. The key stream is XORed with plain text message. Unfortunately several flaws have been found in WEP. Since IV is only 24 bits, and it is transmitted with data packet in clear text. An attacker can observe IV collision (so keystream reuse), to reconstruct the plain text message. A table (decryption dictionaries) of keystreams corresponding to each IV can be built by an attacker. By building such table, there won't be any privacy in the network. In addition, the attacker can also modify messages. The check sum provided by CRC-32 is flawed. It has been shown that the malicious modification made in the message cannot be detected by the check sum. The WEP is found to be vulnerable. There have been several paper addressing WEP vulnerability, the above is just a short description, reader may refer to the original full text in [1].

The public network may not be very secure, the user may establish a secret key and create a private secure channel themselves. A key establishment protocol should be executed.

1.2.2 Key establishment in Bluetooth

In Bluetooth, two devices first establish an Initialization key. This key is used to secure the communication for Link key generation. The Initialization key is established briefly in two steps, as shown in figure 1.1([2]).

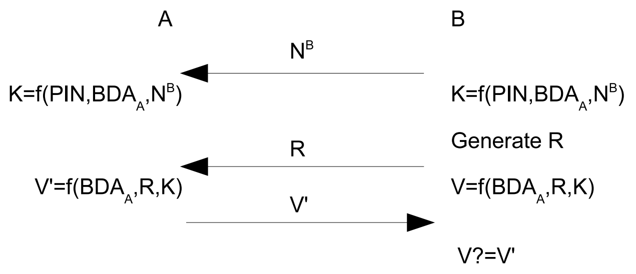


Figure 1.1: Establishment of Initialization Key in Bluetooth

There are two Bluetooth devices, A and B. Each has a Bluetooth Device Address (BDA), BDA_A and BDA_B respectively. Assume they share a secret pin code PIN. Device B first generates a random number N^B and sends to device A. Both devices compute the initialization key K from function $f(\cdot)$ over PIN, BDA_A and N^B. To verify the key, device B generates a random number R and sends to device A. Both devices compute using function $f(\cdot)$ over BDA_A, R and the key K, produce verification stream V and V' respectively. Device A sends V' to device B for verification. The key K is accepted only if $V = V'$. This initialization key K will be used to establish a Link key that is used to encrypt data in further communication. Unfortunately several attacks have been found in this key establishment protocol.

It can be seen clearly that the initialization key K is a function of PIN, BDA_A and N^B. The Bluetooth device address BDA_A is known from every device. The random number N^B is transmitted in clear text. The only thing left secret is the PIN. If the PIN is not available, it is in default set to zero. The PIN can be entered to each device by user. In case the attacker does not know the PIN, he can perform exhaustive search. Since the PIN code is a string that user may often use, to perform dictionary attack might be sufficient. The attacker eavesdrops the above communication shown in figure 1.1 and obtains V' (which equals to V). By guessing a PIN from his dictionary, and computes the key K', runs the verification and produces V''. If $V'' = V$, then he successfully obtains the key K. With the initialization key, the attacker will be able to eavesdrop the Link key generation protocol and obtain link key easily. Device A and B believe they are the only ones who have the secret key, however, the attacker also obtains a copy. He breaks the key establishment protocol from the very beginning.

The above short description shows security problems found in Bluetooth key establishment protocol. The details of this attack as well as several other kind of attacks in Bluetooth can be found in the original full text in [2].

1.3 Conclusion

In WEP, the IV collision hence keystream reuse and the checksum failure from using CRC code have demonstrated general problems found in security solutions that violate Confidentiality, Integrity and Authenticity. The problem found in Bluetooth further illustrates a specific problem in the key establishment protocols. To protect the communication in peer-to-peer network, secret key is necessary. To establish a secret key without any infrastructure as well as ensuring the security of establishment itself is a serious task.

This thesis is to address the secure key establishment in wireless network. In chapter 2, several key establishment techniques are described. Each technique has been analyzed from security point of view as well as their feasibility in embedded computing environment. Chapter 3, the use of distance bounding protocols in key establishment and other purposes are discussed. The distance bounding protocol itself can be used as an authenticator. In Chapter 4, a key establishment protocol based on distance bounding is implemented. Implementation issues are explained. The implementation is tested and described in Chapter 5. For the reader that wishes to review some basic concepts about security may refer to Appendix A.

Chapter 2

Key establishment techniques

This chapter introduces several key establishment techniques. The discussion starts from establishing a key with public key cryptography. Diffie-Hellman method is then described, which the two parties do not have any prior knowledge. Several variants that use DH method are described. Each technique is analyzed from security pointer of view.

2.1 Protection from Public Key Cryptography

A simple key establishment technique is to use public key cryptography [39]. Two parties A and B each hold a public and private key pair, (K^A_{PUB}, K^A_{PRI}) and (K^B_{PUB}, K^B_{PRI}). The public key is known to every one. To establish a session key, A could simply choose a key K_{AB}. He encrypts this session key with his private key K^A_{PRI}, and further encrypted with B's public key, that is:

$$Enc_{K^B_{PUB}}(Enc_{K^A_{PRI}}(K_{AB}))$$

The outer encryption ensures only the one holds private key K^B_{PRI} can see the message (confidentiality), which must be B himself. Since everyone knows B's public key, this message can be sent from any one. So the inner encryption ensures that the key K_{AB} is sent only by A (authenticity). By such a way, party A and B established a session key.

The downside of this scheme is obvious, the public key cryptography is very computational expensive. This technique is not feasible in many embedded devices, which have limited resources. In addition, the two communicating parties should already have public/private key pair, and each others public key has been authenticated through secure channel.

2.2 Diffie-Hellman

2.2.1 Description

One of the principle ideas to establish a key between two communicating parties is to use Diffie-Hellman (DH) technique [3]. The idea is to use the fact that computing some mathematical forward function is easy, whereas computing the inverse function is hard. For example, computing $y = \alpha^x$ is as simple as can be solved by hand, but computing $x = \log_\alpha y$ is much more difficult. For two parties, Alice and Bob, wish to establish a key. Each generates an independent random number X from the set of integers $[1, 2, \ldots, q-1]$, where q is a prime number. In addition, each computes Y by:

$$Y = \alpha^X \bmod q$$

where α is a fixed primitive element of finite field $GF(q)$. Each party puts his Y in public, but X must be kept secret. Y can be named 'public key information', since it is a piece of information about the key that is placed in public. So Alice has X_A, Y_A and Bob has X_B and Y_B. To establish a key between Alice and Bob, they exchange their public key information, that is: Alice gives Bob Y_A, and Bob gives Alice Y_B. For Alice, the key is derived by:

$$\begin{aligned} K_{AB} &= Y_B^{X_A} \bmod q && (2.1) \\ &= \left(\alpha^{X_B}\right)^{X_A} \bmod q && (2.2) \end{aligned}$$

For Bob, the key is

$$\begin{aligned} K_{AB} &= Y_A^{X_B} \bmod q && (2.3) \\ &= \left(\alpha^{X_A}\right)^{X_B} \bmod q && (2.4) \end{aligned}$$

The procedure of establishing a key using DH method can be shown in figure 2.1.

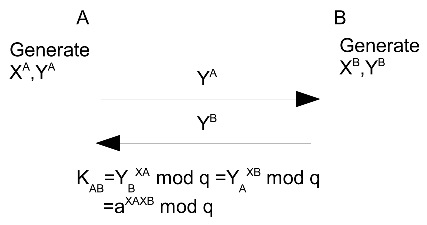


Figure 2.1: Key establishment - DH

2.2.2 Analysis

The DH method is found vulnerable to *Man-In-The-Middle* (MITM) attack. The fundamental problem of the DH method is that the receiver of the public key information cannot verify the sender. Consider the following scenario: Alice and Bob are establishing a key via wireless network, where Mallory notices the progress and interferes with the communication. One special skill Mallory has is to modify the data packet during transmission. As shown in figure 2.2, the

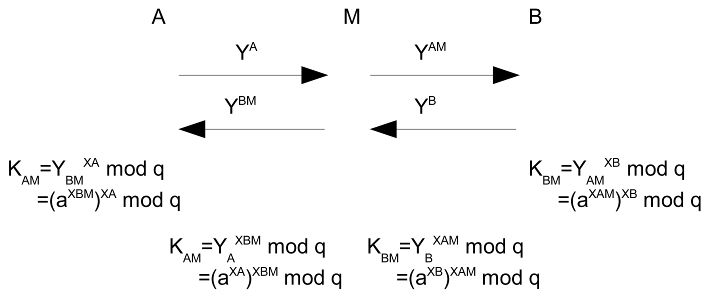


Figure 2.2: Analysis - DH

public key information Y^A is modified by Mallory into Y^{AM}, and Y^B is changed into Y^{BM}. Alice and Bob will proceed with the DH method as usual. But what Alice derived is:

$$K_{AM} = Y_{BM}^{X_A} \bmod q \tag{2.5}$$

$$= \left(\alpha^{X_{BM}}\right)^{X_A} \bmod q \tag{2.6}$$

and Bob gets:

$$K_{BM} = Y_{AM}^{X_B} \bmod q \qquad (2.7)$$

$$= \left(\alpha^{X_{AM}}\right)^{X_B} \bmod q \qquad (2.8)$$

Now the key derived by Alice K_{AM} is not the same as the one derived by Bob K_{BM} without even noticed. Even worse, Mallory has both keys. So Bob begins to send secret message to Alice by the key derived, K_{BM}. Apparently Alice can not read this message, but Mallory can. Mallory reads the message by key K_{BM} and make up a new message, encrypted with K_{AM} and sends to Alice. Now Alice receives a message, and believing it is from Bob. The communication between Alice and Bob with the DH key established provides no security at all.

2.3 Location limited Pre-Authentication

2.3.1 Basic Pre-authentication

As known from previous analysis, the DH method establishing a key between two communicating parties is a good start, but not really secure. One of the solution to such problem is proposed by Balfanz et al. in [4]. The idea is to use a 'Location limited channel' to bootstrap the authentication. They use the advantage that location limited channel can exchange data physically between the involving parties without third party intervention. Such channel is best found to be infrared, due to its directionality.

One basic protocol the author proposed is shown in figure 2.3. The protocol

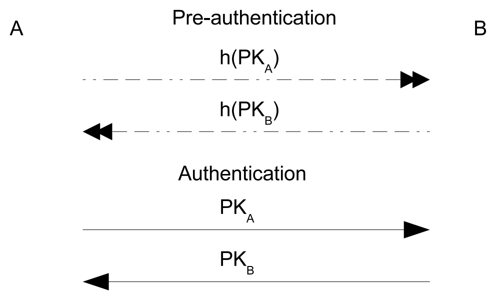


Figure 2.3: Basic Pre-authentication

assumes two parties A and B use public key cryptography. To establish a secure channel, they have to exchange their public key, PK_A and PK_B respectively. The protocol starts with *Pre-authentication* phase. In pre-authentication, two parties use location limited channel, such as IR, to exchange messages. Instead of sending the public key, party A hash the key PK_A and sends to B. The hash has to be a secure cryptographic hash function, so what has been transmitted is the digest of the key. Likewise B sends PK_B to A in the same way. The protocol continues with the public communication channel, e.g. wireless LAN. The following messages are basically to exchange each public key. Any key exchange protocol can be used. After public keys are exchanged, each party computes hash upon key received and matches with the hash received from pre-authentication.

2.3.2 Analysis of Basic Pre-authentication

The basic pre-authentication protocol uses cryptographic hash function to authenticate public keys exchanged via public access network. If there is any intruder to interfere with the key, the hash value will detect such unauthorized modifications (Integrity). The only question is if the hash received can be trusted.

Apparently that is what location limited channel used for. For example, using IR channel, the two communicating parties have to make their transceivers facing each other within certain distance range. Especially with IR that is directional sensitive, it makes the user to have physical insurance that it is the devices we are talking to (Authenticity). Such physical contact makes the MITM attack impossible.

This protocol uses public cryptosystem, however not every device can be equipped or afford such algorithm. A modified version of pre-authentication is proposed and shown in section 2.3.3.

2.3.3 Light weight Pre-authentication

A protocol that supports of one device without public key is proposed. This protocol used for situation where one party has limited computation resource, and not feasible for public key cryptography. The other party still uses public key cryptosystem. The protocol is shown in figure 2.4.

In pre-authentication phase, the party A computes the hash of the public key and sends to B. Party B prepares a secret that he wishes to inform A, denoted as S_B.

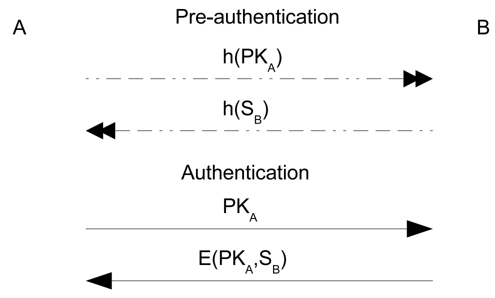


Figure 2.4: Light weight Pre-authentication

Party B hash the secret S_B. The above two messages, again exchanged using location limited channel. The protocol proceed with public wireless channel. Party A sends the public key PK_A to B. B has to verify this key with the hash found in pre-authentication. If the key is authenticated, he can continue to submit the secret to A. Since it is a secret, it has to be concealed by encrypting the secret with PK_A. Everyone can receive the encrypted secret on the wireless network but only the one holding the private key can decrypt the cipher, which is party A. A decrypts the message, obtains the secret S_B. He checks the integrity and authenticity by computing the hash of the secret and compares with the hash received in pre-authentication.

2.3.4 Analysis of light weight Pre-authentication

This protocol provides a key establishment with single public key. The other party could be computational limited. The use of location limited channel possess of the same property described in the basic pre-authentication protocol in section 2.3.1. The hash ensures the integrity and location limited channel ensures authenticity of the message. Due to the nature of location limited channel, such as IR, the MITM attack is prevented.

Even though the two protocols proposed by the author prevents the MITM attack, the downside of the protocol is the usability. Not all devices are equipped with IR channels; some laptop computers may have, but most don't. In addition, due to location limited channel, the users are forced to move closely to the device to be used, which is not always very convenient.

2.4 Password Authenticated Key Exchange

2.4.1 Encrypted key exchange (EKE)

Instead of using additional communication channel with limited access range, a technique using password was proposed. The original protocol was called 'Encrypted Key Exchange' (EKE) by Bellovin and Merrit, reviewed in [5]. The two parties in a conference room wish to establish a key. They first agree upon a password that is written on a blackboard or on a piece of paper. Then type the password into the device (PC,PDA etc.). A password with sufficient length and well chosen combinations can be used. To be user friendly, the password has to be easily recognized. But such passwords are very few, and susceptible to dictionary attack. A stronger session key is derived from the weak password. The procedure that derives the session key from the password is shown in figure 2.5.

The two parties that wish to derive a strong key are A and B. Before everything starts, they decide a password P that is shared between A and B, no one else should know the password. Party A then generates a pair of asymmetric key E_A and D_A for encryption and decryption purpose. He also generates a random string C_A as a challenge string, and S_A as the key information. At the same time, party B also generates R as a symmetric key, C_B as a challenge string and S_B as the key information. The key establishment process possesses of several public known functions; they are $ENC_K(X)$, the encryption function based on key K of information X; $DEC_K(X)$, the decryption function based on key K of information X; $h(\cdot)$ is a one way function and $f(S_A, S_B)$ key establishment function based on key information S_A and S_B.

After generating adequate information, the communication starts. The protocol starts from party A by sending encrypted version of E_A by password P. Upon reception, party B obtains A's public key E_A by decryption. Party B encrypts his symmetric key R with E_A and further encrypted under password P. Party A decrypts the message with password P and decryption key D_A to obtain R. The third message in the protocol is by party A sending his key information S_A and a challenge C_A. The two items are encrypted under the symmetric key R. Party B decrypts the message and reveals the information. He also computes HC_A by function $h(C_A)$. Then HC_A, the challenge C_B and key information S_B are encrypted under key R and sent back to party A as response. Party A decrypts the message and verifies that $h(C_A) == HC_A$. If the two matches, it indicates that party B does know password P so he is able to obtain A's public key E_A. Party A then computes HC_B in the same manner, encrypted with key R and sends to B. Party B verifies HC_B after decryption. If the verification is

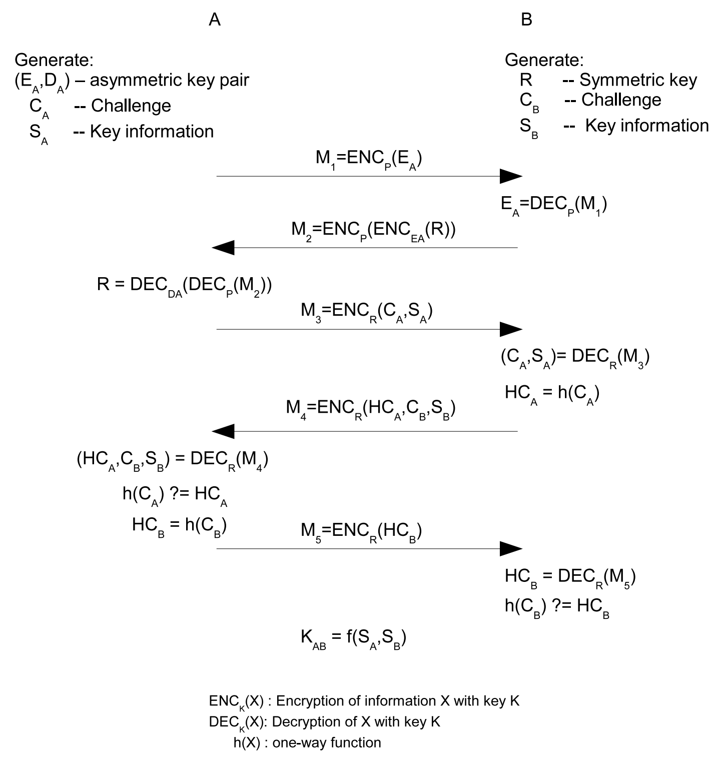


Figure 2.5: Encrypted key exchange - EKE

successful, then B is convinced that party A also knows the password, and he can obtain symmetric key R, so that the key information exchanged S_A and S_B was secure. The session key K is then generated for both parties by computing $f(S_A, S_B)$.

2.4.2 Analysis of EKE

The key establishment was based on information that are kept secret by encryption with password P in the first two steps, and with key R in the following three steps. The first two steps make the two parties agree on E_A and R. So the password is the essential key-point to this solution. But is the password really secure? As the author mentioned, the user friendly password can suffer from dictionary attack. So even if the attacker is outside the 'conference room', so he cannot see the password inside the room, he still can run through entire dictionary and try to find a match. In addition, to obtain the password is probably not difficult by current technology after all. A camera with telephoto lens, or wiretapping equipment makes it very easy to 'hear' and 'see' the password on the blackboard or paper. The user also needs to type the password into the device. The keyboard snooping attack is already not new technique to us. As soon as the password is obtained by the attacker, the MITM attack can be mounted right away.

When the password is exposed, the eavesdropper can obtain E_A. Since A uses asymmetric cryptosystem, the attacker cannot break the message M_2 and obtain R. Without R, the attacker cannot eavesdrop further communications and obtain key information. To break the asymmetric cryptosystem is very hard, so the attacker takes the alternative. When party A sends the message M_1, the attacker eavesdrops the message (obtain E_A) and then jams it (to prevent from being received by B). The attacker sends his message M_1' to party B. This message is given by $M_1' = ENC_P(E_A')$ (where E_A' is attackers encryption key). Before B sends the message M_2, the attacker could send his message M_2' first. This message is given by $M_2' = ENC_P(ENC_{E_A}(R'))$, where R' is attacker's symmetric key. Then party B sends M_2. Since $E_A \neq E_A'$, this message will only be understood by the attacker. Now the situation is A and the attacker shares E_A and R', whereas B and the attacker shares E_A' and R. It forms MITM attack.

In addition, this method requires asymmetric cryptosystem, which is very computational expensive.

2.4.3 Password authenticated DH key exchange

In paper [5], author also proposed a protocol that uses password authenticated key exchange together with DH method. The idea is same as described in section 2.4.1. The procedure is shown in figure 2.6.

The two parties should agree on a password P and type into the device. The public key information Y_A and Y_B should be generated. Private key information X_A and X_B should be kept extremely secure. Challenge C_A and C_B should also be generated as random. Party A should encrypt Y_A with password P and sends to B. Party B decrypts the message by P and obtains Y_A. He is capable of deriving the session key K by DH method. Party B should in turn gives A his public key information Y_B in the same manner encrypted by P. In addition, he also encrypts the challenge C_B with the session key K. Upon reception, party A first decrypts the message and gets Y_B, so that he can derive the session key K. With session key in hand, then he can decrypt the last part of the message and obtains the challenge C_B. As a reply to the challenge, party A encrypts the C_B together with his challenge C_A and sends to B. Party B decrypts the message, and verifies C_B. If the received C_B is correct, party B is convinced that the sender of this message indeed knew the password P and successfully derived the session key K. At the last, party B should reply to A with encrypted C_A. Party A also verifies C_A similarly.

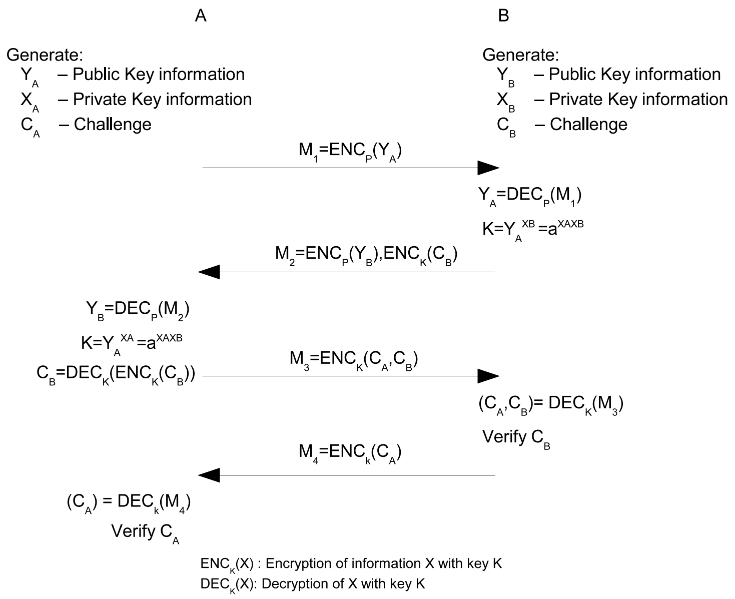


Figure 2.6: Password authenticated key exchange

2.4.4 Analysis of Password authenticated DH key exchange

This protocol uses the advantage of DH method, and avoid of using asymmetric cryptosystem in party A. But the same attack found in section 2.4.2 will also occur. If someone is outside the room and does not have visual contact with the password, he can still run a dictionary attack. If a password is hit, the attacker can decrypt all the messages he eavesdropped. But due to the DH method, the public key information is intended to be public. Even though the attacker can see the information in the message, he cannot derive the session key K. But this does not make this protocol very strong.

The message is authenticated by party B if he is convinced that sender has the password, likewise for A. The password can be exposed soon after it is decided. If it is exposed, the encryption at the first and second step provide no secrecy. It is exactly the same as DH method shown in figure 2.1, and suffers from the MITM attack shown in figure 2.2 on page 7.

2.5 Seeing is Believing (SiB)

2.5.1 Description

Inspired by Balfanz et al. with pre-authentication in [4], McCune et al. uses visual pre-authentication using camera phones and bar code to exchange key materials in [6]. For human to read and compare a long meaningless hash strings in hexadecimal number system is cumbersome. Instead, a bar code is generated based on the hash value, and it is captured by a digital camera on mobile phone. The procedure is shown in figure 2.7.

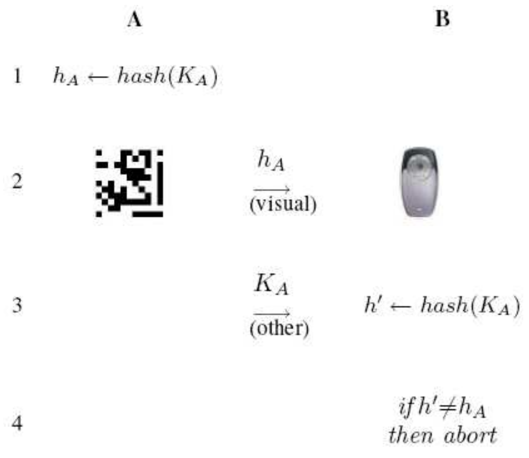

A
B
1
$h_A \leftarrow hash(K_A)$
2
$\xrightarrow[\text{(visual)}]{h_A}$
3
$\xrightarrow[\text{(other)}]{K_A}$
$h' \leftarrow hash(K_A)$
4
$if\, h' \neq h_A$
$then\ abort$

Figure 2.7: Seeing is Believing (SiB)

Two parties A and B wish to exchange their public key to establish a secure communication channel. They are all equipped with mobile phones that have digital cameras and LCD displays. Party A first computes the hash of this public key K_A resulting h_A. Then he transforms the hash value into a graphical bar code and shows it on the LCD. Party B then uses his digital camera to focus on A's LCD, and captures the bar code. The pre-authentication phase is finished here. Party A can use public wireless channel to send K_A to party B. If secrecy is still wanted on the public key, encryption or other key exchange protocol can also be used. Upon receiving K_A, party B computes the hash value and verifies with the one received through pre-authentication phase. The establishment failed if the two do not match.

2.5.2 Analysis of SiB

This novel approach for public key exchange uses the advantage of imaging capability in modern electronics. By capturing an image (a bar code), the device is ensured to communicate with the one intended to. If the other device is not compromised, the bar code displayed can be understood as a signature of that device (don't confuse this signature with 'Digital signature', here means something that device uniquely possessed). The third party has no way to modify such information. So MITM attack cannot be mounted in this proposal.

The requirement of this proposal is that the devices are equipped with cameras and displays. If one party does not have camera nor display, an unidirectional authentication also can be realized. The displayless device can be a printer or wireless access point (A.P.). The unidirectional authentication requires the displayless device to hold a public/private key pair. A bar code is computed and printed on a plate that is physically attached to the device housing.

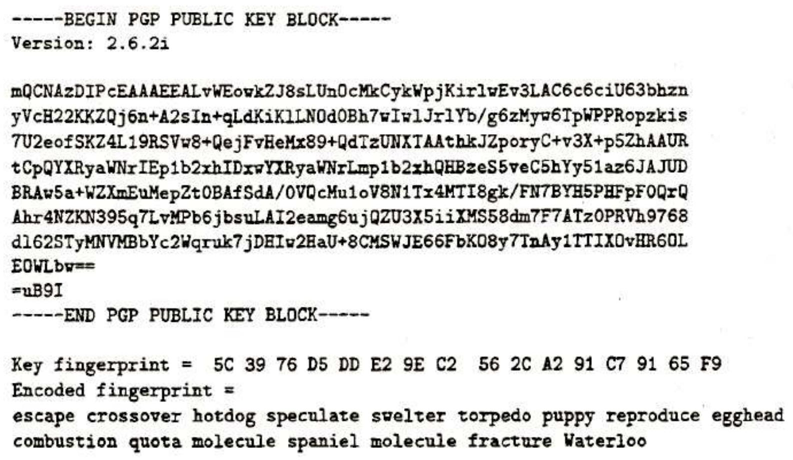
However such attachment might not be physically secure. It is like a vehicle licence plate. It can be manually removed, exchanged or swapped. A user may wish to establish a connection with a printer to print a confidential document. Assume the printer has no display, so bar code is attached to the housing. However this bar code plate was manually swapped by a spy with a fax machine. When the connection was established and believed to be secure, the user sends the document to print, which in fact was sent to the fax machine and transmitted out of the country.

For devices that are both equipped with cameras and displays, this proposal is secure. For the situation where only unidirectional authentication is possible, physical security of the bar code must be guaranteed. Further more, there are many devices that cameras are not standard components, such as PDA. This proposal will be difficult to invoke under insufficient light condition, such as darkness or smoke.

2.6 Loud and clear (L&C)

2.6.1 Description

The previous technique uses visual contact to authenticate public key materials. In this section, the idea of using audio channel to authenticate the message is described. The idea is the same as PGPfone in [7].



```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2i

mQCNAzDIPcEAAAEEALvWEowkZJ8sLUnOcMkCykWpjKirlwEv3LAC6c6ciU63bhzn
yVcH22KKZQj6n+A2sIn+qLdKiKlLNOdOBh7wIwlJrlYb/g6zMyw6TpWPPRopzkis
7U2eofSKZ4L19RSVw8+QejFvHeMx89+QdTzUNXTAAthkJZporyC+v3X+p5ZhAAUR
tCpQYXRyaWNrIEp1b2xhIDxwYXRyaWNrLmp1b2xhQHBzeS5veC5hYy51az6JAJUD
BRAw5a+WZXmEuMepZt0BAfSdA/0VQcMu1oV8N1Tx4MTI8gk/FN7BYH5PHFpFOQrQ
Ahr4NZKN395q7LvMPb6jbsuLAI2eamg6ujQZU3X5iiXMS58dm7F7ATzOPRVh9768
dl62STyMNVMBbYc2Wqruk7jDHIw2HaU+8CMSWJE66FbKO8y7TnAy1TTIXOvHR6OL
EOWLbw==
=uB9I
-----END PGP PUBLIC KEY BLOCK-----

Key fingerprint =  5C 39 76 D5 DD E2 9E C2  56 2C A2 91 C7 91 65 F9
Encoded fingerprint =
escape crossover hotdog speculate swelter torpedo puppy reproduce egghead
combustion quota molecule spaniel molecule fracture Waterloo
```

Figure 2.8: PGPfone

In figure 2.8 [7], a PGP public key block is shown. At the bottom of the figure, the key fingerprint is computed to verify the integrity. The fingerprint of the PGP key is further encoded into human language vocabulary to ease memorization. This is the idea of PGPfone. As can be seen, the PGPfone maps the fingerprint into words that does not necessarily make up into a meaningful sentence. In [8], a solution called 'Loud and Clear(L&C)' is developed. It uses the 'Text-to-Speech' engine to generate robust-sounding and syntactically correct English sentence mapped from hash of a public key.

The principle steps of key exchange using 'L&C' is shown in figure 2.9.

Two devices exchange public key. Party A starts by computing hash of his public key PK_A and obtain h_{KA}. Then he runs through 'L&C' library to generate a syntactically correct sentence S. Such sentence is output to the speaker by 'Test-to-Speech' technology. In the mean while, the public key PK_A is sent over

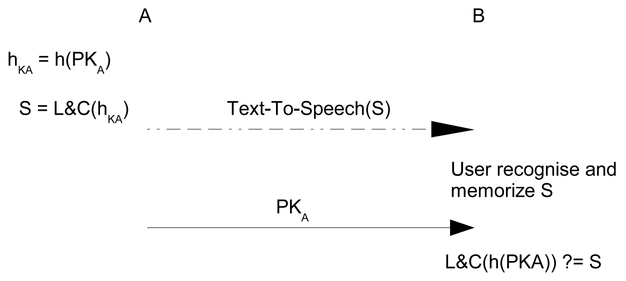


Figure 2.9: Laud and Clear

wireless link to party B. The party B's device should compute the same hash and run through the 'L&C' library, maps the hash into sentence. If the sentence mapped by B is the same as he heard, the public key from A is authenticated.

2.6.2 Analysis

This technique uses the audio channel to transfer information to authenticate the public key. For devices such as PDA and laptop computers, sound generation through on board speaker is rather easy to detect and recognised by human. Certainly this does require an advanced research and development of 'Text-to-Speech' technology and the 'L&C' library. This method however, won't work with person who has hearing disability. Further more, this solution will not be very trustworthy if the loud speaker is external instead of embedded. Since external speaker is usually wired or sometime wireless, wiretapping or MITM attack can be mounted between the speaker and the voice generator (i.e. PDA, laptop).

2.7 Short string comparison (MANA)

2.7.1 MANA I&II

The solution in [10] provides several proposals using short string comparison to manually authenticate public key information. The initial idea was found by Maher in US patent 5,450,493 [9]. The MANA protocol is an improvement over Maher's idea. The procedure of MANA I is shown in figure 2.10.

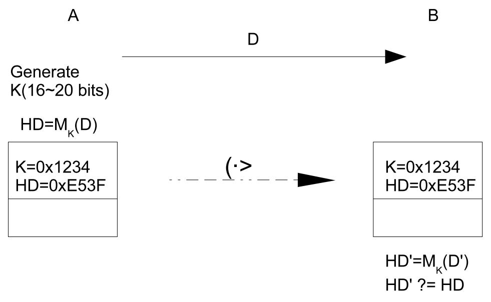


Figure 2.10: MANA I

The procedure involves party A and B. They wish to exchange public key information D. Party A first sends D to B over any means, such as wireless connection without any protection. In order for B to authenticate D, party A generates a key K that is 16˜20 bits long. Then he uses a function $M_K(\cdot)$ to compute a check-value over D and obtain HD. The function $M_K(\cdot)$ can be an universal hash function family. Party A types in the key K and the check-value HD to B's device, or B can take a look at A's display and types in himself. With key K in hand, party B recomputes the check-value upon the received D (noted as D' in the figure). He obtains the check-value HD' and compares with the HD that was typed in by A. If the two matches, the D is authenticated, abort otherwise.

Another variant was developed for devices do not have a keypad. In this case, the key and the check-value cannot be manually typed in. The solution is to transfer the key K also in wireless channel. Both devices then compute check-value and displayed on the screen. The user has to visually verify his key and check-value against the others. This version of the solution is called 'MANA II'.

2.7.2 Analysis of MANA I&II

The MANA I solution uses a stronger authenticated channel [11] to provide check-value. A stall-free transmission is a property of stronger authenticated channel. A face to face conversation is a stall-free transmission, and hence stronger authenticated, whereas E-mail is not. The user A typed key and check-value personally into B's device. Such check-value cannot be tampered, unless A is dishonest or the device is already compromised, which we assume that is not the case. The attacker is not constrained to find the second preimage. By knowing D from wireless connection, and observing the check-value, the attacker can try to compute another message $\hat{D}$ that is different from D, but $M_K(D) = M_K(\hat{D})$. As has been analyzed in [10], the probability of such a successful attack is less than 2^{-13} for 16-bit keys and check-values, and less than 2^{-17} for 20-bit keys. The attacker has to perform excessive attempt with very small chance to succeed.

2.7.3 MANA III

This solution is aimed at devices that have keypad, but simple displays such as seven-segment display. This solution also uses MAC (Message Authentication Code) with key K to compute check-value. The procedure is shown in figure 2.11.

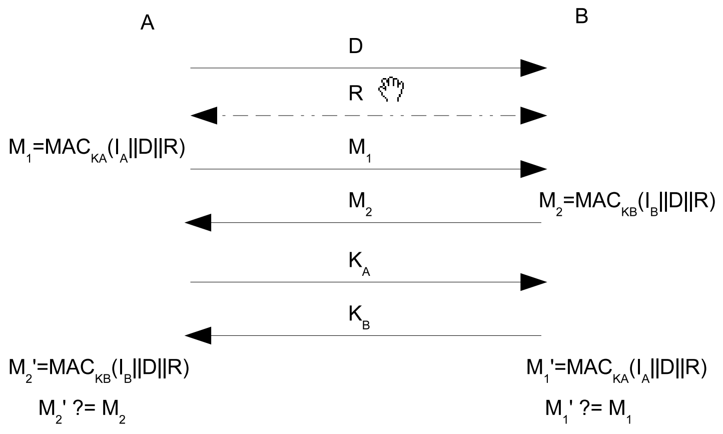


Figure 2.11: MANA III

Party A sends public key information D to B via wireless connection. Then he generates a random string R and typed into both A and B's device manually by hand. Party A computes the MAC value over I_A, D and R with key K_A, where $||$ denotes for concatenation in the figure, and I_A is the identifier for A. He sends the MAC value M_1 to B. At the same time, party B performs similar operation, computes M_2 over I_B, D and R sends to A, where I_B is the identifier for B. Both parties exchange the key K_A and K_B. The last step, party A verifies M_2 and party B verifies M_1. If both are authenticated and display 'SUCCESS', the public key information D is accepted, abort otherwise.

2.7.4 Analysis of MANA III

Since the attacker does not know R, which is typed in by keypad, it is very difficult to perform an attack on this solution. To simply modify D into some other value basically does not work. The M_1 sent by party A will detect such modification. To modify or even make up and replace both D,M_1 and key K_A can neither break this solution. Since the lack of knowledge of R, to find same MAC value from different R is rather difficult. To work out R from MAC in the absence of the keys, as the author mentioned, is infeasible.

2.8 DH with Short string comparison (DH-SC)

2.8.1 Description

A key agreement protocol using string comparison based on DH method is proposed by Cagalj et al. in [14]. The protocol achieves optimal trade-off between security and usability. The protocol is shown in figure 2.12.

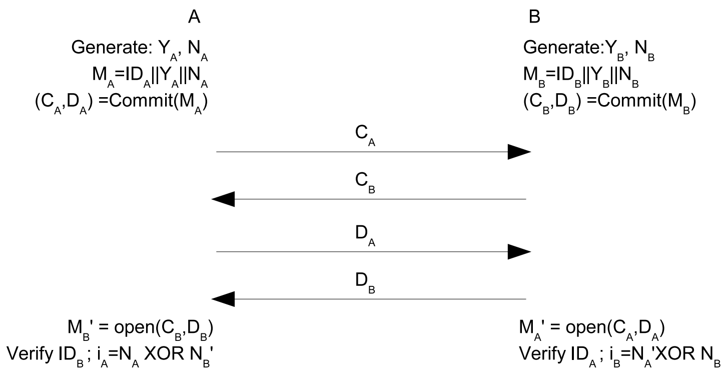

A
B
Generate: Y_A, N_A
M_A=ID_A||Y_A||N_A
(C_A,D_A) =Commit(M_A)
Generate:Y_B, N_B
M_B=ID_B||Y_B||N_B
(C_B,D_B) =Commit(M_B)
C_A
C_B
D_A
D_B
M_B' = open(C_B,D_B)
Verify ID_B ; i_A=N_A XOR N_B'
M_A' = open(C_A,D_A)
Verify ID_A ; i_B=N_A'XOR N_B

Figure 2.12: DH with short string comparison

The protocol agrees upon a key between party A and B. Each at the first place, generates a public key information Y_A and Y_B respectively. A random number that is k bits long is also generated, noted as N_A and N_B. Each party produces a message, M_A and M_B by concatenating Y and N leading with their identity tag, ID_A or ID_B. Each party commits to the message and obtains a commitment-opening pair (C, D) . The commitment possesses of hiding and binding property. It is infeasible to find a $\hat{D}$ that is different from D, so that $(C, \hat{D})$ pair reveals another message $\hat{M}$ that is different from M. In the following, two parties exchange commitment C_A, C_B as well as the opening value D_A and D_B. With (C_B, D_B) in hand, party A is able to open the B's message and denoted as M'_B. The notation M'_B is to distinguish the received copy from the original value. Similarly, party B opens A's message M'_A. Both parties should first verify the ID tag from the message, abort the protocol if altered. Then party A computes $N_A \oplus N'_B$ and denoted as i_A, where $\oplus$ indicates XOR. In turn, party B computes $N'_A \oplus N_B$ and denoted as i_B. If $i_A = i_B$, then the message is authenticated, both A and B accept public key information Y_B and Y_A.

2.8.2 Analysis

The DH-SC protocol is analyzed through a MITM attack scenario. The attack is shown in figure 2.13.

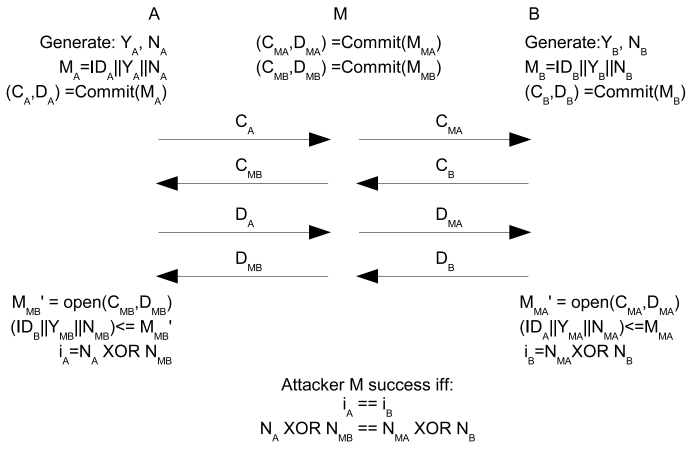

A
M
B
Generate: Y_A, N_A
M_A=ID_A||Y_A||N_A
(C_A,D_A) =Commit(M_A)
(C_{MA},D_{MA}) =Commit(M_{MA})
(C_{MB},D_{MB}) =Commit(M_{MB})
Generate:Y_B, N_B
M_B=ID_B||Y_B||N_B
(C_B,D_B) =Commit(M_B)
C_A
C_{MA}
C_{MB}
C_B
D_A
D_{MA}
D_{MB}
D_B
M_{MB}' = open(C_{MB},D_{MB})
(ID_B||Y_{MB}||N_{MB})<= M_{MB}'
i_A=N_A XOR N_{MB}
M_{MA}' = open(C_{MA},D_{MA})
(ID_A||Y_{MA}||N_{MA})<=M_{MA}
i_B=N_{MA}XOR N_B
Attacker M success iff:
i_A == i_B
N_A XOR N_{MB} == N_{MA} XOR N_B

Figure 2.13: MITM Attack in DH-SC protocol

This analysis introduces the third party M as Mallory. He makes up some message M_{MA} and M_{MB}, also computes the commitment pairs (C_{MA}, D_{MA}) and (C_{MB}, D_{MB}) respectively. When party A commits and sends C_A to B, Mallory reads this message and then prevents it from being received by B. He then transmits his version of the commitment denoted as C_{MA} to B. From B's point of view, this commitment is still sent from A. In turn, B commits to his message and sends C_B to A. Similarly, Mallory intercepts this transmission, and provides his version C_{MB} to A. Likewise, the opening value D_A is replaced by D_{MA} and D_B is replaced by D_{MB}.

The two legal parties A and B now open the message and perform authentication. The attack will be successful if two parties cannot recognise the change made in the commitment; that is A's string i_A matches with B's i_B. In order to have $i_A = i_B$, the XOR operation computed in both parties have to be the same; that is:

$$N_A \oplus N_{MB} = N_{MA} \oplus N_B$$

It can be seen easily, the only condition for such equation to hold is $N_{MB} = N_B$ and $N_{MA} = N_A$. Since at the time Mallory commits to his message M_{MA} and M_{MB}, he does not have any knowledge about N_A and N_B. In order for Mallory to have $N_{MB} = N_B$ and $N_{MA} = N_A$, he has two choices; either wait for opening value is published, which will be too late for him to attack the protocol; or Mallory has to make a wild guess. The probability for $N_{MA} = N_A$ depends on the length k, i.e. $P_{SUCC} = 2^{-k}$. For $k = 16$, the chance to be success is only 0.0015%. For a well chosen value k, the probability for the attacker to success will be significantly small.

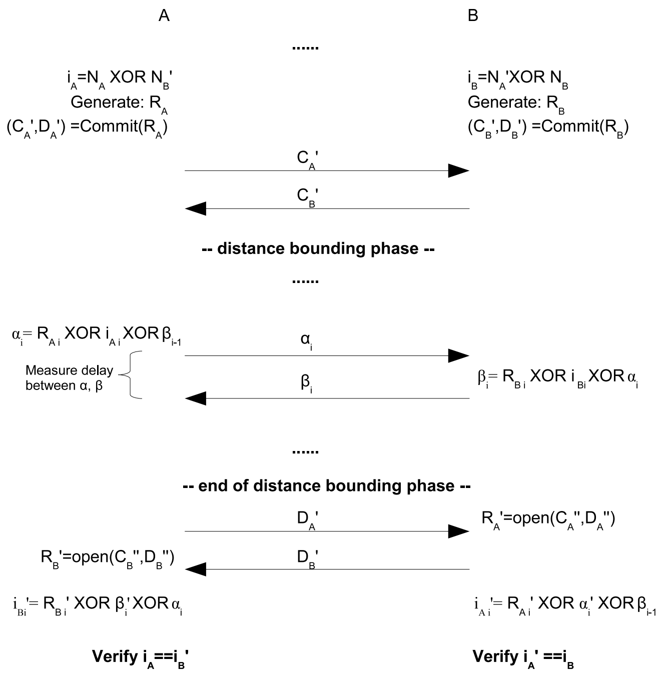
2.9 DH with Distance bounding (DH-DB)

2.9.1 Description

The previous DH method with short string comparison shown in figure 2.12, requires the user A and B to visually compare strings i_A and i_B. This method is secure, however not very user friendly. A new procedure has been proposed to automate such tedious procedure. It is DH method with distance bounding protocol by Capkun et al. in [14]. The protocol is shown in figure 2.14. The procedure to exchange commitment C_A, D_A and C_B, D_B is the same as DH-SC protocol, therefore not shown here. Only the procedure to compare i_A and i_B is briefly explained.

The idea is for two parties further exchange their i_A and i_B. Each party prepares a k bits random string, R_A and R_B respectively. They commit R and obtain (C'_A, D'_A), (C'_B, D'_B). They exchange the commitment C'_A and C'_B. The distance bounding phase can be started. In each step, party A computes

$$\alpha_i = R_{Ai} \oplus i_{Ai} \oplus \beta_{i-1}$$



A
B
......
i_A=N_A XOR N_B'
Generate: R_A
(C_A',D_A') =Commit(R_A)
i_B=N_A'XOR N_B
Generate: R_B
(C_B',D_B') =Commit(R_B)
C_A'
C_B'
-- distance bounding phase --
......
α_i= R_{Ai} XOR i_{Ai} XOR β_{i-1}
α_i
Measure delay between α, β
β_i
β_i= R_{Bi} XOR i_{Bi} XOR α_i
......
-- end of distance bounding phase --
D_A'
R_A'=open(C_A'',D_A'')
R_B'=open(C_B'',D_B'')
D_B'
i_{Bi}'= R_{Bi}' XOR β_i' XOR α_i
i_{Ai}'= R_{Ai}' XOR α_i' XOR β_{i-1}
Verify i_A==i_B'
Verify i_A' ==i_B

Figure 2.14: DH-DB protocol

where i=1,2,...,k, and $\beta_0 = 0$. Party B computes

$$\beta_i = R_{Bi} \oplus i_{Bi} \oplus \alpha_i$$

In such a way, α hides i_A by R_A and β, and β hides information about i_B by R_B and α. For party B, each bit in α is a challenge, and β is the response. Likewise, the previous bit β_{i-1} is the challenge for party A, and α_i is the response. Each should send out response as soon as possible. Party A measures the time between α_i and β_i; party B should measure the time between β_{i-1} and α. When all k rapid bit exchanges are finished, A and B will have an estimation of the distance between them. If the value found by distance bounding phase matches with the true distance, then the protocol continues, abort otherwise. The following steps of the protocol is to decommit by exchanging D'_A and D'_B. Each party opens and obtains R_B and R_A, reveals i. For party A,

$$i_{Bi} = \alpha_i \oplus \beta_i \oplus R_{Bi}$$

and for party B,

$$i_{Ai} = \alpha_i \oplus \beta_{i-1} \oplus R_{Ai}$$

Now party A and B can verify the equality of i_A and i_B locally by their computer. This protocol uses automated method to compare two strings to authenticate message. It also uses the location information to *verify the sender of the message*. In this section, only the basic idea is explained, and short version of the protocol is demonstrated. In chapter 3, the use of distance bounding protocol will be thoroughly studied, further details and analysis will be explained.

2.10 Shake them up

2.10.1 Description

The previous protocols discussed are based on DH method, which use secret exponent to compute public key information. Such method cannot always be applied. For example, for devices with very limited CPU resource, memory and power, such method can be overkill. A protocol proposed by C.Castelluccia et al. called 'Shake them up' in [12] can be used for such situation. The protocol proposed is inspired from the protocol developed by Alpern and Schneider. The protocol made by Alpern et al. requires $4n$ messages to be exchanged to establish a n bits secret key. The 'Shake them up' protocol further optimizes the number of messages into n. The protocol proceed in following steps:

1. Party A selects $n/2$ random bits, $R_A[1], R_A[2], \ldots, R_A[n/2]$
2. Party B selects $n/2$ random bits, $R_B[1], R_B[2], \ldots, R_B[n/2]$
3. Party A forms $n/2$ messages, $m_A[1], m_A[2], \ldots, m_A[n/2]$ where the source address of the message is:
$$src = \begin{cases} A & \text{if } R_A[j] = 1 \\ B & \text{if } R_A[j] = 0 \end{cases}$$
4. Party B forms $n/2$ messages,$m_B[1], m_B[2], \ldots, m_B[n/2]$ where the source address of the message is:
$$src = \begin{cases} B & \text{if } R_B[j] = 1 \\ A & \text{if } R_B[j] = 0 \end{cases}$$
5. Party A and B send their message to each other in n steps. For the i^{th} step, only the message $R_A[j]$ or $R_B[j]$ is sent. The order between A and B is purely random.

6. Upon reception of the message, A and B checks whether the source address is correct to determine the current bit value for the key K_{AB}.

An example is shown in figure 2.15([12]). Two devices wish to derive a secret key to secure further communication. The key size is 8 bits, each need to prepare 4 random bits so in total 8 message exchanges.

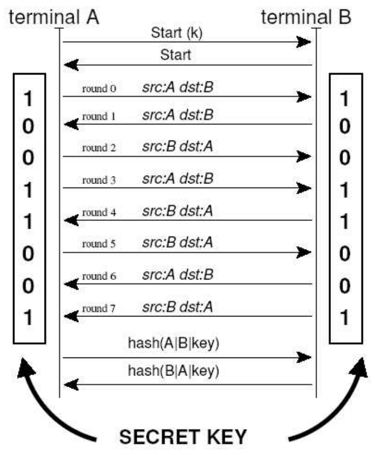


Figure 2.15: Shake them up

The protocol starts by device A sending a START message, which contains ‘k’ that is the size of the key, and the address of A. Party B should reply another START message containing the address of B. The following message exchanges progress by time slots. In the first time slot, device A looks at his random bit $R_A[1] = 1$, he marks a ‘1’ in this secret key string and sends a message containing correct source address,i.e. ‘src=A,dst=B,data=NULL’. The message only have source and destination address without any data payload. Upon reception, device B knows the message is sent from A, since he didn’t send any message. He checks the source address that is correct, he also marks a ‘1’ in the secret key string. In second time slot, B decided by random to send a message. He looks at his random bit $R_B[1] = 0$, he marks a ‘0’ in the secret key string, and make up a message with wrong source address, i.e. ‘src=A,dst=B,data=NULL’. Upon reception, A knows the message is sent from B, and the source address is wrong so he also marks a ‘0’ in his secret key string. The following messages proceed in the same manner. When 8 messages are exchanged, each device should compute a hash and send to the other device for verification.

This protocol did not use cryptography to protect the secret. The attacker is allowed to eavesdrop the communication. But the idea to protect the information is similar to what is done in cryptography. One technique to achieve confidentiality is by confusion. In this protocol, the confusion is realized by *Source Indistinguishability*. The protocol executant should not expose who sent which message. The source indistinguishability is provided by two major contributions, i.e. *Temporal indistinguishability* and *Spatial indistinguishability*. The randomness of true message sender provides temporal indistinguishability. The attacker cannot determine who should send the next message. Whereas spatial indistinguishability is bit difficult to realize. The attacker can measure the signal strength and distinguish the source. The solution of this problem is to move the device around, turn them up and down, and hold in hand shake them up during key exchange. The movement provided by shaking randomizes the device location, so signal strength analysis cannot be performed.

2.10.2 Analysis

The confusion provided during secret key exchange makes the attacker has no idea of who actually send which message. However, in the first two messages exchange, the device A also have no insurance of device B is the one he selected. MITM attack might be attempted. The solution to such problem is to define a signal level threshold. The author discovered that when two devices are placed $2cm$ apart and when they touch each other, the signal level are significantly different. The closer they are, the higher signal level they have. By defining a signal level threshold, the device in the proximity that exceed this threshold can continue with the key exchange. That is when a user wishes to establish a connection between two devices and moved them closer to each other.

Even though the protocol provides confusion to the attacker, there will still be information leakage. Each radio transceiver exhibits some unique characteristic, such as radio central frequency. Such characteristic is determined during manufacture, it is like in nature that no two tree-leaves are exactly the same. The crystal clock oscillator generates carrier signal. One clock will be different from the other, and hence the central frequency of the carrier. By further investigation, each device's unique characteristic known as 'signature' or 'signal fingerprint' can be recognised. The attacker can pin-point the exact sender of any message. In this case, the protocol will not be secure. On the other hand, it also require the attack has a very advanced RF signal analyzer.

2.11 Conclusion

This chapter studied several proposals of key establishment with many different techniques. The strong cryptography protected key establishment is secure and difficult to break, but also very difficult to implement in embedded system with limited resources. The DH method instead, expose only public partial key information. The key is established through the use of public key information together with another partial information that is kept secret. This method is feasible in embedded system, but many implementations are vulnerable to attacks. The message and entity authentication has been used in the key establishment. Use of location limited channel prevents the attacker from interception; provides the legal devices with appropriate physical contact. A secret such as a password can be used to authenticate the entity, and derive a stronger key from the weak password. Audio and images have also been used to authenticate legal parties. Short strings can be computed and used for visual comparison, so to mutual authenticate two devices with a very high probability. Physical distance from the message receiver to the message sender can be found from distance bounding. Such distance authenticator provides a very reliable authentication. In the mean time, it automates the key establishment process. A key establishment protocol without cryptographic protection has also been proposed. The designer uses temporal and spatial indistinguishability to confuse the eavesdropper.

The techniques described in this chapter all have different measures to counter attack. The specialty they have, also exhibits a special requirement for implementation. Not all devices are equipped with infrared, keypad, especially cameras. Very few embedded system can afford ‘Text-to-Speech’ engine. In contrast, distance authenticator verifies the sender of the message. The distance can be measured by time-of-flight with radio or ultrasound wave, which can be implemented in many devices that has radio or ultrasound transceiver.

In next chapter, the use of distance bounding protocol in key establishment as well as in many other protocols to authenticate entities will be explained.

Chapter 3

Distance Bounding Protocols

The previous chapter has described several key establishment techniques. This chapter will focus on distance bounding protocols and the use in the key establishment. The distance bounding with rapid single bit exchange (DB-BE) is described in section 3.2. Followed by distance bounding word exchange (DB-WE) in section 3.3. The DH method used together with distance bounding is in section 3.4. The echo protocol designed in Berkeley and the DBP protocol from Cambridge are also discussed in section 3.5 and 3.6. The so called MAD protocol used in SECTOR is shown in section 3.7. Each protocol has been analyzed. First, this chapter starts from an appetizer, which is a simple protocol as an introduction.

3.1 Appetizer

3.1.1 Protocol Description

The first protocol designed in this chapter is a simple key exchange protocol that is inspired by string comparison protocol according to the work in [13] [14]. The protocol is shown in figure 3.1. Bob prepares the message M containing the public partial key information, Y as noted in previous chapters. Bob commits to

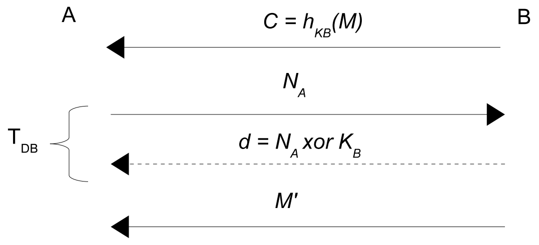


Figure 3.1: Appetizer

the message by standard commitment scheme. The most simple implementation of commitment is using hash function, $h_{K_B}(\cdot)$. For an input message, it will give an unique hash value. The commonly known hash functions are MD4, MD5, SHA etc. The commitment scheme using cryptosystem can be used as well.

Bob initiates the protocol by sending the commitment C to Alice via radio. Then the protocol enters the second phase for Bob to decommit. This phase is started by Alice sending a random number N_A, a sequence of arbitrary 0s and 1s , as a challenge (authenticator). N_A should be the same size as K_B. Upon reception of the entire nonce, Bob computes XOR between N_A and K_B as $N_A \oplus K_B$ and denoted as d. The decommitment d is sent via ultrasound. The time between the challenge and the arrival of the response is measured by Alice and denoted as T_{DB}. The last phase of the communication is sending the message by Bob in clear, and denoted as M'.

When the above process finished, Alice will compute h_{K_B} over the message M' and denoted as C'. She compares C' with C. If they match, the protocol succeed, abort otherwise.

3.1.2 General security analysis

This protocol intends to provide Alice the public key information contained in a string M from Bob. MITM attack can be attempted in this protocol. The attacker Mallory could simply tamper the communication. He prevents C to be received from Alice (e.g. corrupt the message), and sends his version of the commitment C_M to A. By XOR N_A with his hash key K_M and replaces the message string to M_M, the key derived by A is in fact between A and the attacker Mallory.

However, Alice will not be fooled. The extra measure to prevent the intruder is the distance measurement, namely T_{DB}. Alice records the time when N_A was sent, and stops her watch when the response d is received. From the knowledge about the physics, she can properly compute the distance (radio or ultrasound) message travelled, so to determine the location of the actually sender. Alice roughly estimates the distance between herself and Bob (who she intends to communicate), denoted as D_{AB}. If the distance she measured does not match with the true distance D_{AB}, the intrusion is detected and discard all the message she had received.

The attacker also noticed the distance measurement in the protocol, he did not give up. Instead, he like to cheat. There are several cases to be considered. The attacker is geographically further away from Alice than Bob, or he is in somewhere between them. In the first situation, the attacker has to perform distance shortening attack; whereas in the second, he has to perform distance enlargement attack. These two kinds of attacks can be very successful. The details of the attack is explained in the coming two sub-sections.

3.1.3 Analysis - Distance shortening

To understand distance shortening attack, consider the simpler situation only involves Alice and Bob, but Bob is the dishonest prover (so to remove the third role in the discussion). Alice receives key information from Bob, but Bob would like to cheat on the location where he actually is. He decided to appear with shorter distance on Alice's screen. The protocol shown in figure 3.1 can be decomposed further into more detailed steps as shown in figure 3.2.

The first and the last message sent by Bob is non-important in this attack. The attack will appear in the challenge and response steps, and therefore noted differently in this figure to have timing visualization. Alice sends the challenge N_A bit by bit over radio link. Assuming radio signal travels in the air at speed of light, i.e. $c = 3 \times 10^8 m/s$. Upon reception of the entire challenge, Bob computes the response $N_A \oplus K_B$, denoted as d, and sends back to Alice over ultrasound. The ultrasound travels much slower than radio, but provides a more precise distance measurement. The processing time between the reception of the nonce and sending of the response is defined as Δp. As shown in the figure, T_0 denote the time in which first challenge bit is sent. Alice can send out the second bit of the challenge after the first bit has been completely sent, which depends on the channel bit rate $b_r(\,bits\,per\,second)$. Therefore,

$$T_i = T_{i-1} + \frac{1}{b_r} \qquad (3.1)$$

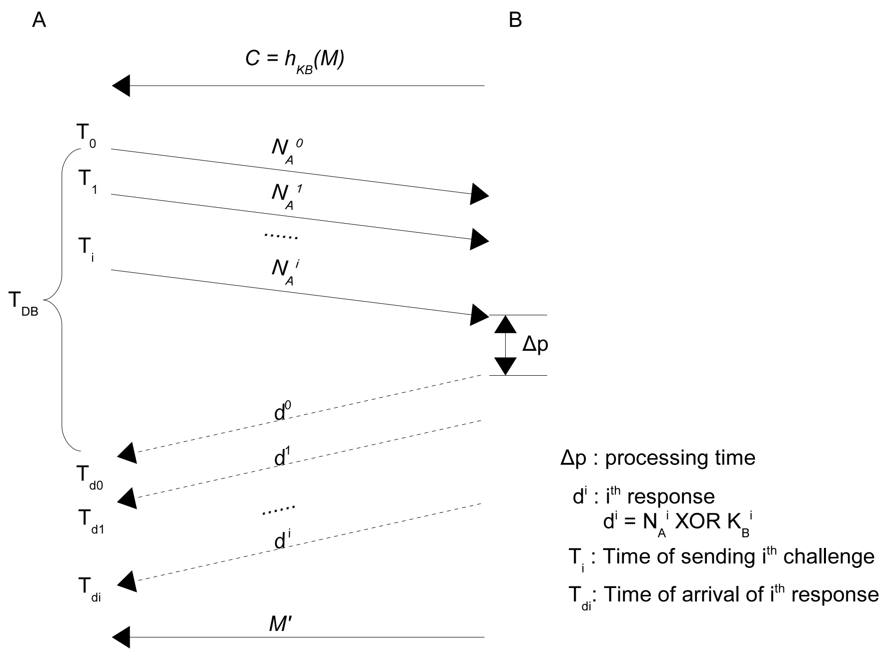


Figure 3.2: The appetizer with expanded details

where b_r is the bit rate over radio link. As in commercial product such as Chipcon CC1000 will operate at $20kbps$. In other words, it takes $50us$ until the next bit can be transmitted. Similarly, the data transmission over ultrasound channel behaves in the same way, but much more slower than radio channel. The sound signal travels in air at speed of $340m/s$, defined as s. T_{di} denotes the arrival time of the i^{th} response bit (shown in the figure). In contrast to radio links, the bit rate of ultrasound communication is much slower than in radio. In this work, the bit rate over ultrasound b_s is about $16bps$, or $60ms/bit$ (details please refer to section 4.4 Data transmission over ultrasound).

$$T_{di} = T_{di-1} + \frac{1}{b_s} \tag{3.2}$$

For the sake of simplicity, the processing time Δp is assumed to be negligibly small. In order for Alice to determine Bob's location, she must record T_0 and T_{d0} and calculates T_{DB}.

$$\begin{aligned} T_{DB} &= T_{d0} - T_0 && (3.3) \\ &= \frac{l}{c} + \frac{n}{b_r} + \frac{l}{s} && (3.4) \end{aligned}$$

where, l is the distance between Alice and Bob, n is the nonce length. Equation 3.3 gives a clear model of what contribute into the time measurement. $\frac{l}{c}$ indicates the time takes for the first challenge bit to travel over the distance l. $\frac{n}{b_r}$ indicates the time for the entire n-bit nonce to be sent. $\frac{l}{s}$ indicates the time for the first response bit to travel over distance l. The three quantities can be denoted as t_1, t_2 and t_3 respectively. The distance can then be found by:

$$\begin{aligned} T_{DB} &= t_1 + t_2 + t_3 && (3.5) \\ T_{DB} &= \frac{l}{c} + \frac{n}{b_r} + \frac{l}{s} && (3.6) \\ T_{DB} - \frac{n}{b_r} &= \frac{l}{c} + \frac{l}{s} && (3.7) \\ \left(T_{DB} - \frac{n}{b_r}\right) \cdot c \cdot s &= l(s+c) && (3.8) \\ l &= \frac{\left(T_{DB} - \frac{n}{b_r}\right) \cdot c \cdot s}{s+c} && (3.9) \end{aligned}$$

Equation 3.9 shows the calculation of the distance based on timing measurement. However, equation 3.3 and 3.5 is based on the assumption that one event occurred after another in a consecutive (sequential) manner. Meaning t_3 cannot occur until t_2 is finished. Unfortunately such assumption is not always valid. A dishonest prover, Bob in this case, could send out the first response earlier before t_2 elapsed. He could even start sending before verifier (as Alice) sent the challenge (before t_1 starts). Now, the previous model 3.5 is not valid. t_3 overlaps on t_1 and t_2. From verifier's point of view, t_3 is shortened or even disappeared. The entire timing measurement T_{DB} is much smaller than the true value, hence, the distance calculation shown in equation 3.9 is telling no more the truth.

The prover has successfully broken the distance measurement in this protocol, but he still has to make sure the response bit is correct. Since the attacker needs to send the response earlier than the challenge is arrived, he does not have the correct challenge. What he can do is to make a wild guess. To guess for one challenge bit, he has 50% chance to be correct. To guess n challenge bits, he only has $2^{(-n)}$ chance to succeed. Unfortunately, this number doesn't always reflect of what is actually happening in real implementation.

As mentioned, the ultrasound communication device has a much slower bit rate than the radio device (be aware, here is talking about bit rate instead of the velocity of the signal). Such as in this work, to send one bit in ultrasound takes about $60ms$, whereas in radio link takes only $50us$. To send a 64-bit challenge in radio takes $3.2ms$. Assume the attacker need to send out his response at the same time as Alice sends the challenge. The attacker flips a coin to guess the first challenge bit and then sends the response. It is clear that it takes $60ms$ to send the first response bit. To send the second response bit, he needs to flip another coin. But he noticed, during the first $60ms$, all other radio challenge bits have already been received (which only takes $3.2ms$). What the attacker needs to do is using the challenge, and sends the response!

For n bits challenge, it supports to have success probability of only $2^{(-n)}$. But in real implementation, what he need is to guess the first bit of the challenge. During the sending of the first response bit, all other challenge bits come for free. The attacker only need to make ONE guess, the probability to be success becomes 50%!! The attacker reduced the distance, and has a very high probability to break the authentication. This protocol is not secure any longer.

3.1.4 Analysis - Distance Enlargement

Similar to distance shortening attack, the prover can also mount distance enlargement attack. Referring to figure 3.2 on page 32, instead of sending response earlier, the prover postpones the reply. The timing measurement T_{DB} now has an offset. The distance calculation will be a much larger result. By adjusting the delay offset, the prover can be at any location he wishes.

Such attack can be used in the situation where Alice and Bob is trying to exchange key, and Eve is sitting (physically) somewhere in the middle. Eve can invoke such attack to successfully establish a key with Alice, to whom the distance corresponds to Bob's location.

For such kind of attack, the solution is for the legally involved two parties to look around the vicinity, if no one is in between, then such attack can be avoided. This introduces the new concept "Integrity Region" [14].

3.1.5 Analysis - Integrity Region

The 'Integrity Region' is a 3-D space (two spheres) that is each centered at one legitimate party with radius d that is equal to the distance between the two, shown in figure 3.3 [14]. There are two possibilities that an attacker can be. Either he is inside the spheres, or outside. For attacker who is outside the sphere, he has to play distance shortening attack to pretend being correct position. If

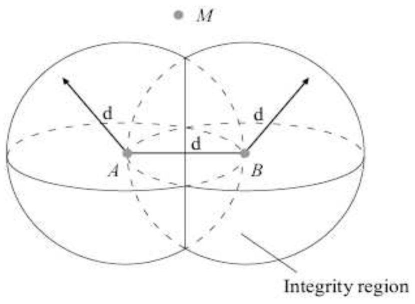


Figure 3.3: Integrity Region

the attacker is inside the spheres, he need to play distance enlargement attack.

For correct implementation of distance authenticator, the distance shortening attack cannot be performed successfully with probability less than 2^{-n}, where n is the size of the random string. However the distance enlargement attack is easier, the attacker obtains the challenges for free. Since the attacker has to be within the spheres, this attack can be defeated if the legitimate parties can make sure that no one else is physically inside the spheres.

In this protocol, the two parties look around and visually make sure that there is no third party within the 'Integrity Region', the distance enlargement attack is avoided. For distance shortening attack, a correct implementation of distance authenticator can be found in next section: 'Distance bounding - Bit exchange'.

3.2 Distance bounding - Bit exchange (DB-BE)

3.2.1 Protocol description

The protocol is shown in figure 3.4. Two parties, Alice and Bob wishes to exchange key. Bob commits to the secret message M, by standard commitment scheme $h_{K_B}(\cdot)$.

Bob initiates the protocol by sending the commitment C to Alice. Now the protocol enters the distance bounding phase. In distance bounding phase [15], Alice first generates an n bits random sequence N_A. In each step, she sends one bit N_A^i from the sequence to Bob as a challenge, and expecting the response.

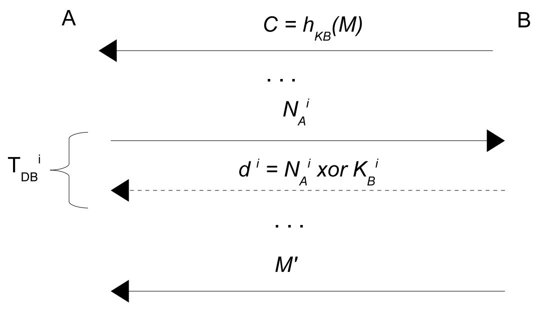


Figure 3.4: Distance bounding protocol - Bit exchange

Upon the reception of N_A^i Bob should compute:

$$d^i = N_A^i \oplus K_B^i \tag{3.10}$$

and send the response back to Alice immediately. Alice should record the time when challenge is sent and the response is received, denoted as T_{DB}^i. Alice and Bob will repeat such steps. The distance bounding phase is finished when every bit in K_B is XORed with N_A and received by Alice. The time T_{DB}^i is measured in each steps. At the last step of the communication, the secret message M can now be sent to Alice in clear, denoted as M'.

Since Alice possesses of the random sequence N_A, she can derive K_B from d as decommitment.

$$K_B = d \oplus N_A \tag{3.11}$$

She now computes $C' = h_{K_B}(M')$ and compares with previously received C. If $C' == C$, then the key is agreed, otherwise the process fails.

3.2.2 Analysis - Distance shortening

This protocol is an enhancement based on Protocol Appetizer, the procedure is shown in figure 3.4. As discussed in section 3.1.3, the location verification is based on the timing measurement contributed from the three factors shown in equation 3.5 and 3.6 on page 33. The problem is the factors that being modelled in the equation do not, by nature, exhibit a sequential ordering. The dishonest prover can simply guess the first challenge bit, and obtain the rest challenge bits in the mean time. Even worse, this is the only timing measurement being used for location verification. So, that protocol can be seen as an one-phase distance bounding protocol. The nonce length doesn't weaken the attack.

In this protocol, the attention to such weakness has been paid. The protocol uses the advantage of the lengthy nonce. The verifier challenges the prover bit by bit, one bit at a time. The prover should reply to each challenge-bit individually. One challenge bit should be sent only after the previous response bit has been received (this is different from the appetizer). Each challenge and response pair is composed into a distance bounding step. There will be N distance bounding steps in the protocol, in which N is the number of bits in the random nonce. The timing should be measured in each step and contributes to the final distance verification. A graphical demonstration is shown in figure 3.5 to give timing visualization.

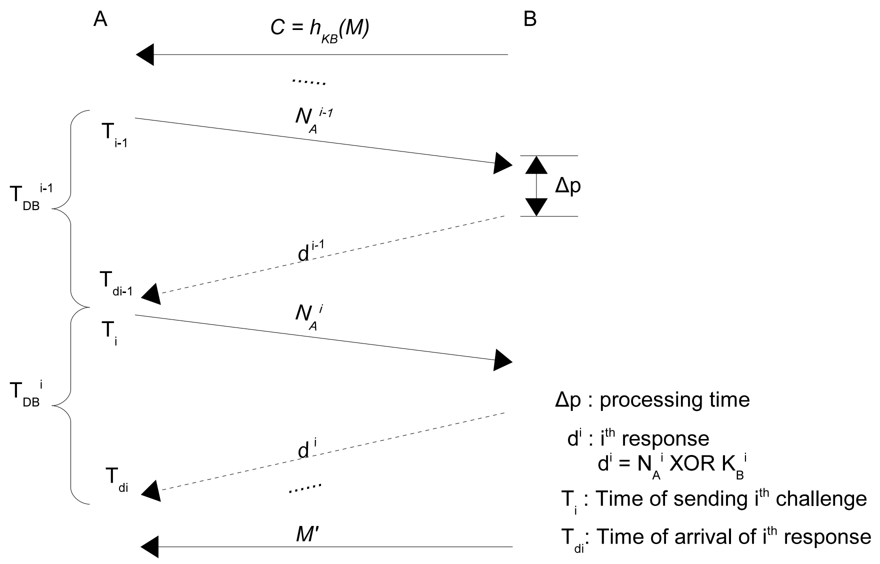


Figure 3.5: Expanded details

The timing measurement for each phase is denoted as T^i_{DB}, in which $i \in \{0, n-1\}$. T^i_{DB} can now be modelled as:

$$\begin{aligned} T^i_{DB} &= t_1 + t_2 + t_p + t_3 \\ &= \frac{l}{c} + \frac{n}{b_r} + \Delta p + \frac{l}{s} \end{aligned} \tag{3.12}$$

It is the time for the challenge to travel through the medium, t_1; the time for the entire bit to arrive at the destination, t_2. The prover's processing time,

t_p, and the time for response to travel back to the verifier, t_3. t_1 and t_3 are the same as before. n is the number of bit in each distance bounding step, in this protocol, n is one. The processing time is assumed to be negligibly small. In this protocol construction, the distance shortening attack can also be attempted. The dishonest prover may send the response bit prior the challenge bit has been received, so he has to guess the challenge. But in this protocol, the next challenge bit is not revealed until the previous challenge is replied, he has to guess the challenge bit for each distance bounding step. For each step, the prover has 50% probability of succeed in the random process. But for N bits challenges, and hence N steps of distance bounding, the probability that an attacker survived is reduced to $P = 2^{-N}$.

3.2.3 Analysis - Distance Enlargement

Similar to section 3.1.4, the distance enlargement attack can be attempted in this protocol. The prover may delay his response to impersonate someone further away. The solution is already given in previous section, and hence not explained again here.

3.2.4 Analysis - Signal encoding attack

Besides distance shortening by purely chance, another form of attack can be explored in ultrasound physical signal encoding scheme.

As explained in the previous section, the data rate of the ultrasound transmission is, in this work, about $60ms$ per bit, or 17bps (details please refer to section 4.4 Data transmission over ultrasound). In order to send a logic one, 8 cycles of ultrasound pulses, in total $200us$ are sent at the beginning of $60ms$ period. To denote a logic zero, the channel will be silenced for $60ms$. Similarly in the receiver, it looks into $60ms$ interval (defined as *sampling interval*). A logic '1' is concluded if there is ultrasound signal detected, otherwise logic '0' is received. In the protocol, there is a requirement of measuring the arrival of the response by the prover. However by this encoding of logic '0' and logic '1', the arrival of '0' cannot be timed. Fortunately, the data transmission format solved such problem. The format is, for convenience, shown below.

During transmission of the ultrasound data, a synchronization prefix is sent prior the data payload. The synchronization prefix is always '1'. The first bit of the data payload, which is designated to be the response bit, is following the prefix. The rest payload bits are for miscellaneous purpose. Such transmission

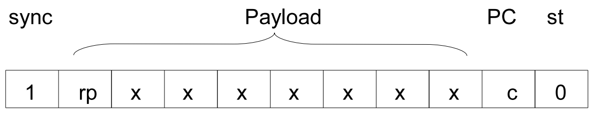


Figure 3.6: Ultrasound packet format

format uses the synchronization bit to provide a potential opportunity to encode the response bit. The encoding scheme can be summarized in table 3.1.

Encoding		Value
sync	response bit	
1	0	0
1	1	1

Table 3.1: Signal encoding scheme

From the receiver, he sees a '1-0' pattern to interpret a logic zero as a response, and '1-1' as a logic one. This encoding scheme makes sure that the verifier can time both the arrival of '1' as well as arrival of '0'.

Although this technique satisfies the functional requirement, it does violate the security need. The physical signal of the encoding is shown in figure 3.7.

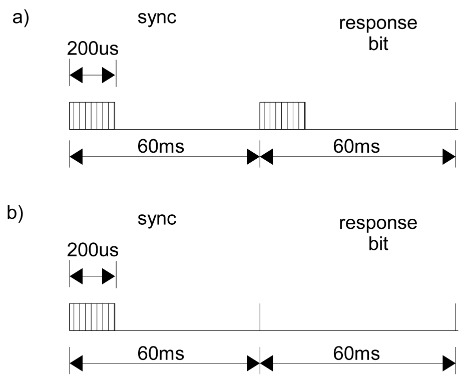


Figure 3.7: Physical signal representation. a)encoding of logic '1', b) encoding of logic '0'

The actual ultrasound pulses last only for $200us$. There are more than $59ms$ channel idle time to avoid echoes reflected from vicinity (also due to hardware limitations). In the receiver, to provide flexibility, it does not restrict the ultrasound pulses to appear at the beginning of the $60ms$ *sampling interval*. Because of this, a cooperative attack can be mounted to this protocol. Based on the current encoding scheme, the dishonest prover Bob could simply reply a 'zero' to the verifier at arbitrary time according distance he wishes to cheat. The 'team member' Mallory, a malicious attacker, he listens to the radio message sent from Alice and inject ultrasound pulses into the medium. The attack is illustrated in figure 3.8.

After Bob commits to the verifier Alice, a challenge bit will be sent to Bob. The challenge bit is denoted as N_A^i. Bob was prepared to cheat on the distance, so he sent the response d^i much earlier. In order to pass the authentication of the commitment scheme, he could guess the challenge bit, as being done in the attacks of the previous protocols. But now in this protocol, Bob won't survive, because there are many challenge bits he has to guess on. There is neglectablly small probability such kind of attack succeed. But due to the encoding scheme, Bob doesn't have to concern about the value of the challenge bit, what he need to do is simply sent a logic '0' to the air, and Mallory will do the rest. Mallory is the one that stood in the middle of Alice and Bob. He can listen to message sent by both parties. Mallory particularly interested in the challenge N_A^i, since the correct response sent by the prover also depends on the challenge. If the current challenge N_A^i equal to K_B^i, then the response should be 0 as it was sent by Bob. If the challenge is different from K_B^i, the response should be 1, and Mallory should now be active and inject an ultrasound pulse into the channel.

The figure only shows signal injection in one distance bounding step. Signal P is what Mallory injected into the data, which is originally sent by Bob. P is found by:

$$P = \begin{cases} 1 & \text{if } N_A^i \neq K_B^i \\ 0 & \text{otherwise} \end{cases} \tag{3.13}$$

Assume the true distance between Alice and Bob is $100m$, and denoted as D, the distance Bob pretends to be is $10m$ from Alice, denoted as l. The speed of sound s is $340m/s$. The time takes for Bob's ultrasound message to arrive at Alice is t_{BA}:

$$t_{BA} = \frac{D}{s} = \frac{100m}{340m/s} = 294ms \tag{3.14}$$

The time for message sent from the legitimate party called Carol, l meters away from Alice, is found by:

$$t_{CA} = \frac{l}{s} = \frac{10m}{340m/s} = 29.4ms \tag{3.15}$$

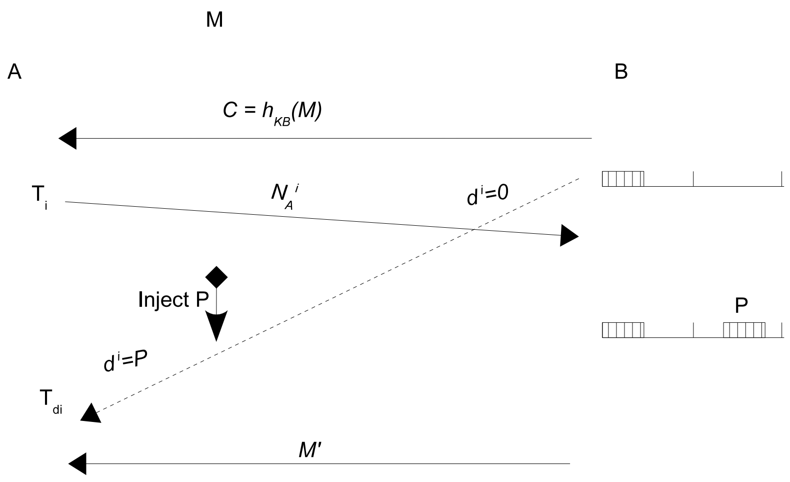


Figure 3.8: Cooperative attack on Protocol II by injection

$$t_B = t_{BA} - t_{CA} = 294ms - 29.4ms = 264.6ms \tag{3.16}$$

Therefore, Bob has to sent the response 264.6 ms earlier in order to pretend to be Carol's position, denoted as t_B (Assuming radio message does not take time). Bob now knows precisely when to sent his message to break the protocol. The question is where should Mallory stand. Since the ultrasound receiver doesn't restrict the ultrasound pulses to be at the beginning of the sampling interval, so there are plenty of time for Mallory to react and inject pulses into the message.

$$t_M = 120ms - 200us = 119.8ms \tag{3.17}$$
$$D_M = t_M \cdot s = 119.8ms \cdot 340m/s = 40.7m \tag{3.18}$$

Equation 3.17 and 3.18 shows the Mallory's location. As shown in figure 3.7, two bits encode one bit logic value, which is $120ms$ (each bit is $60ms$). The ultrasound signal lasts for $200us$. The attacker can wait for $119.8ms$ to inject an ultrasound signal. During this time, Mallory can determine the correct response. This time correspond to distance of $40.7m$. So in order for Mallory to perform signal injection attack, he should stand at most $40.7m$ away from Alice. The graphical representation of each party's geographical position is shown in figure 3.9. As indicated in the figure, both Bob and Mallory are outside the radius l between Alice and Carol , but the protocol still can be broken.

The problem of such attack is from the fact that logic '0' is represented by absent of ultrasound signals. It can be easily modified. This problem is originated from

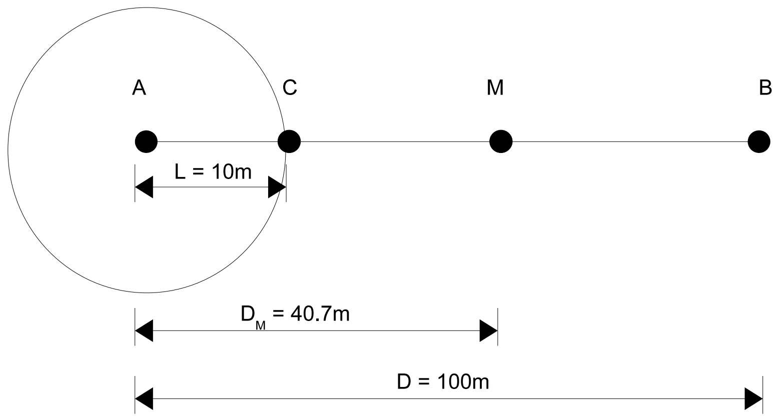


Figure 3.9: Geographical position of the attack

the hardware construction. The solution is to implement an extra ultrasound hardware transceiver, a better signal encoding such as 'Manchester code' can be implemented. In Manchester encoding, the logic value is represented by a transition, i.e. logic '1' is encoded by -5V to +5V transition, whereas logic '0' is encoded by +5V to -5V transition. Once the logic value is sent through the transceiver, it is very hard to modify.

3.3 Distance bounding - Word exchange (DB-WE)

3.3.1 Description

This protocol, shown in figure 3.10, is very similar from the previous two. The first protocol described in section 3.1 is efficient with less overhead. The second protocol described in section 3.2 is more redundant, but more secure. The protocol designed here understand the previous protocols as two extremes, and takes the advantages of both two.

The procedure of this protocol does not differ very much from the previous. After Alice received commitment from Bob, she will send nonce to Bob as challenge. In the previous protocols, the nonce is either sent entirely to Bob at once, or sent one bit at a time and another bit after receiving the previous response.

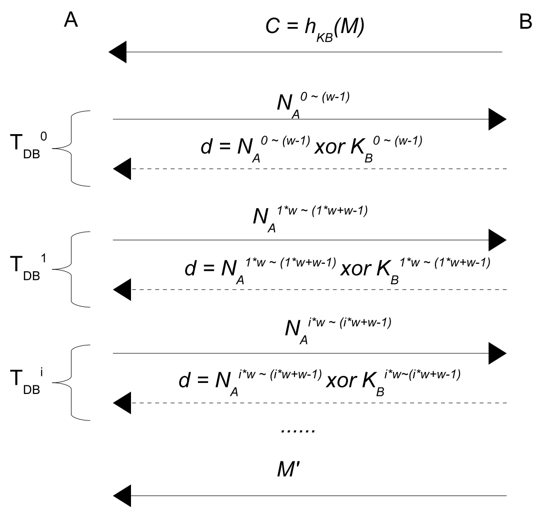

A
B
C = h_KB(M)
N_A 0 ~ (w-1)
T_DB 0
d = N_A 0 ~ (w-1) xor K_B 0 ~ (w-1)
N_A 1*w ~ (1*w+w-1)
T_DB 1
d = N_A 1*w ~ (1*w+w-1) xor K_B 1*w ~ (1*w+w-1)
N_A i*w ~ (i*w+w-1)
T_DB i
d = N_A i*w ~ (i*w+w-1) xor K_B i*w~(i*w+w-1)
......
M'

Figure 3.10: Distance bounding - Word exchange

In this protocol, the N bits random nonce is divided into "words", each word is composed of several bits. The word size is defined as W. As shown in figure 3.10, assuming word size is defined as $W = 8$. Alice will send the first 8 bits of N_A to Bob, i.e. N_A^0 ˜ N_A^7. After receiving W bit challenge, Bob should send corresponding bits in K_B XORed with the challenge and reply to Alice as soon as possible. The time between the challenge and the response is recorded, and denoted as T_{DB}^0.

The next challenge word can be sent by Alice, and expecting the response. The challenge and the response word can be formalized as:
The i^{th} challenge:

$$N_A^{i*W} \sim N_A^{(i*W+W-1)}, \qquad i = 0 \ldots \frac{N}{W} \tag{3.19}$$

The i^{th} response:

$$[N_A^{i*W} \sim N_A^{(i*W+W-1)}] \oplus [K_B^{i*W} \sim K_B^{(i*W+W-1)}], \qquad i = 0 \ldots \frac{N}{W} \tag{3.20}$$

where i is non-negative integer from 0 to $\frac{N}{W}$, and N is the length of the nonce, W is the word size.

The challenge and response steps will be repeated until Bob decommit all the bits in K_B. There will be i rounds distance bounding in this protocol. The time taken between challenge and response is denoted as T^i_{DB}. Notably the nonce size N should be integer multiple of word size W.

At last, the message M will be sent by Bob in clear in the same manner as the previous protocols.

After the communications are finished, Alice will check the Integrity of the message the same as described in the previous protocols. The timing measurement will be used to verify the location of Bob. In case of no violation is found, the message is accepted, failed otherwise.

This protocol can be seen as a generalization of the previous two. It has the same nonce size as Protocol Appetizer, but less distance bounding rounds, and hence less communication overhead. In security point of view, this protocol is less stronger than Distance bounding - bit exchange. This protocol can be trimmed to have minimum overhead, i.e. word size W equal to nonce length N, which will result to be protocol Appetizer. It can also be trimmed to have maximum security, i.e. word size W equal to 1, which is the second protocol.

3.4 DH with Distance bounding (DH-DB)

3.4.1 Description

The DH key exchange with distance bounding protocol has been introduced in section 2.9. This protocol is based on DH-SC string comparison, and further extended with distance bounding. The advantage of DH-SC is to strong authenticate public key information through human string comparison. The user compares two strings displayed on each other's screen. To reduce such tedious process, the user exchanges two strings with each other, and compare by computers. The strings are hidden by random nonce, and exchanged in a bit-by-bit manner. By measuring the time during rapid bit exchange, the distance between the two parties can be determined. This distance value will be used to verify the sender of the message.

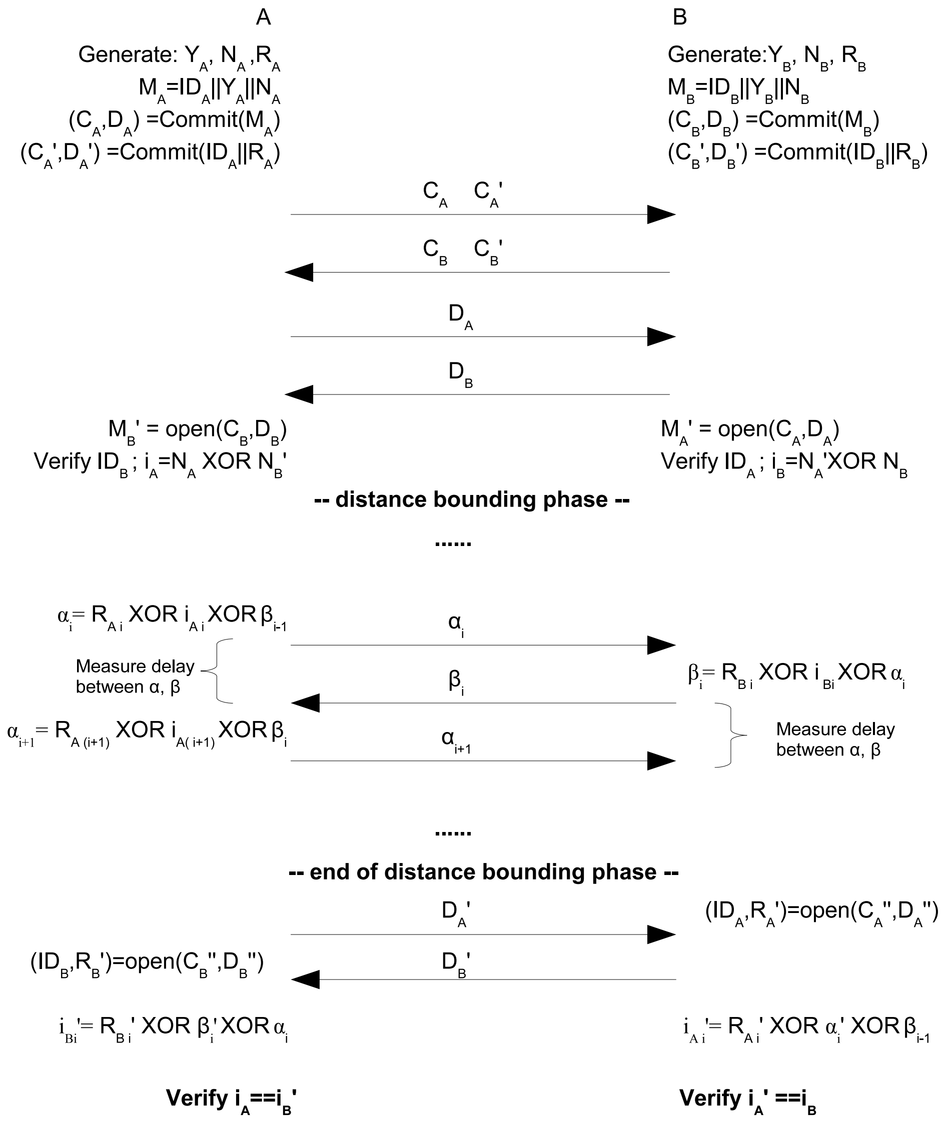

A
B
Generate: Y_A, N_A, R_A
M_A=ID_A||Y_A||N_A
(C_A,D_A) =Commit(M_A)
(C_A',D_A') =Commit(ID_A||R_A)
Generate: Y_B, N_B, R_B
M_B=ID_B||Y_B||N_B
(C_B,D_B) =Commit(M_B)
(C_B',D_B') =Commit(ID_B||R_B)
C_A C_A'
C_B C_B'
D_A
D_B
M_B' = open(C_B,D_B)
Verify ID_B ; i_A=N_A XOR N_B'
M_A' = open(C_A,D_A)
Verify ID_A ; i_B=N_A' XOR N_B
-- distance bounding phase --
......
α_i= R_Ai XOR i_Ai XOR β_i-1
α_i
Measure delay between α, β
β_i
β_i= R_Bi XOR i_Bi XOR α_i
α_i+1= R_A(i+1) XOR i_A(i+1) XOR β_i
α_i+1
Measure delay between α, β
......
-- end of distance bounding phase --
D_A'
(ID_A,R_A')=open(C_A'',D_A'')
(ID_B,R_B')=open(C_B'',D_B'')
D_B'
i_Bi'= R_Bi' XOR β_i' XOR α_i
i_Ai'= R_Ai' XOR α_i' XOR β_i-1
Verify i_A==i_B'
Verify i_A' ==i_B

Figure 3.11: DH with distance bounding (DH-DB)

The detailed procedure of DH-DB is shown in figure 3.11 [14]. The same as DH-SC protocol, Alice generates public key information Y_A, nonce N_A and concatenates with ID_A into message M_A. She commits to the message and obtains (C_A, D_A) pair. Bob does the same. In addition, another nonce R_A is generated, and committed with ID_A into (C'_A, D'_A) pair, likewise for Bob. Alice and Bob exchange (C, D) pair, open the message and compute i_A and i_B respectively. Notice that the commitment C'_A and C'_B is also exchanged in the above steps. However, the decommitment D'_A and D'_B should NOT be sent by now, the reason will be explained in the coming analysis(section 3.4.2).

Distance bounding phase will start from this point. The distance bounding repeats for k steps, where k is the size of nonce R_A as well as N_A. Each step, Alice computes:

$$\alpha_i = R_{Ai} \oplus i_{Ai} \oplus \beta_{i-1}$$

where i=1,2,...,k, and $\beta_0 = 0$. Alice sends α_i to Bob as a challenge. Bob should in turn compute:

$$\beta_i = R_{Bi} \oplus i_{Bi} \oplus \alpha_i$$

and send β back to Alice immediately. Alice measures the delay between α and β. Alice should not relax, β_i is a challenge to Alice as well. She must compute the next bit of α and sends back to Bob right away. In turn, Bob also measures the time between β and α. After repeat for k times, distance bounding phase is terminated. Each party should now decommit and exchange D'_A and D'_B. They opens the message and obtain R'_B and R'_A. Reveals i'_B and i'_A as:

$$\begin{aligned} i'_{Bi} &= R'_{Bi} \oplus \alpha_i \oplus \beta'_i && (3.21) \\ i'_{Ai} &= R'_{Ai} \oplus \alpha'_i \oplus \beta_{i-1} && (3.22) \end{aligned}$$

Alice can compare i_A with i'_B, and Bob compares i'_A with i_B, which can be done by computer. They discard the exchanged key if the comparison failed. In addition, Alice and Bob should verify the distance between them against the value obtained from distance bounding phase. If the measurement corresponds with the true distance, they can accept the public key exchanged, and derive the session key by DH method; abort otherwise.

3.4.2 Analysis - Distance shortening attack

The distance bounding is to determine the location of the actual message sender. Distance shortening is an obvious attack on such protocol. For Alice, the distance is determined by measuring the time between α and β. To reduce the time, the attacker has to reply β earlier. As can be seen, the response β is a function of the challenge α. Without knowing α, the attacker cannot compute correct response β. To know α, the attacker has to know R_A. It is very difficult (almost impossible) for an attacker to obtain R_A before distance bounding phase terminated. The *hiding* property of commitment scheme ensures that only (C', D') pair can open the the message R. To reveal R from C' is not feasible. Therefore, the decommitment D' is exchanged only after distance bounding is terminated.

3.4.3 Analysis - MITM attack

The DH method with string comparison (DH-SC) has been analyzed in section 2.8.2 on page 22 and concluded to be secure. Any attacker intercepts the message will be detected by string comparison. In DH with distance bounding protocol (DH-DB), the string comparison is not performed by human visual inspection. MITM might be attempted. To analyze such situation, figure 3.12 demonstrates such attack.

Assume Alice and Bob exchange public key information using DH-DB protocol. Mallory tampered the messages sent by each party. He successfully exchanged the messages containing his version of the public key with Alice as well as with Bob. So that Alice hold string i_A, Mallory hold corresponding $i_B^M = i_A$. Bob hold i_B and Mallory hold corresponding $i_A^M = i_B$, but $i_A \neq i_B$. Alice and Bob generate R_A and R_B respectively. They commit to R obtain (C'_A, D'_A) and (C'_B, D'_B). In turn, Mallory generates R_B^M and R_A^M, and commitment $(C_B^M{}', D_B^M{}')$ and $(C_A^M{}', D_B^M{}')$. During distance bounding, Alice sends challenge α to Bob, but Mallory jams this message so to prevent it to be received by Bob. He replies β^M to Alice. Similarly, Mallory sends α^M to Bob, pretends to be Alice and expects β from Bob. After k steps of distance bounding, each send his decommitment and reveals R. Alice verifies the string $i_A == i_B^M{}'$ and Bob verifies $i_B == i_A^M{}'$. Certainly both device will display ‘SUCCESS’. The attacker played MITM attack, and passed string authentication. Alice and Bob did not exchange key with each other, but in fact exchanged public key with Mallory. However, the protocol did not fail. There is one more step for distance authentication.

Alice and Bob look at the screen, the distance measured during distance bounding phase does not correspond to the true distance between them. In stead of accepting the key just exchanged, they decide to abort. The key should only be accepted if the verification shows ‘SUCCESS’ and the distance matches with the true value.

As explained in the previous analysis, distance shortening attack on distance

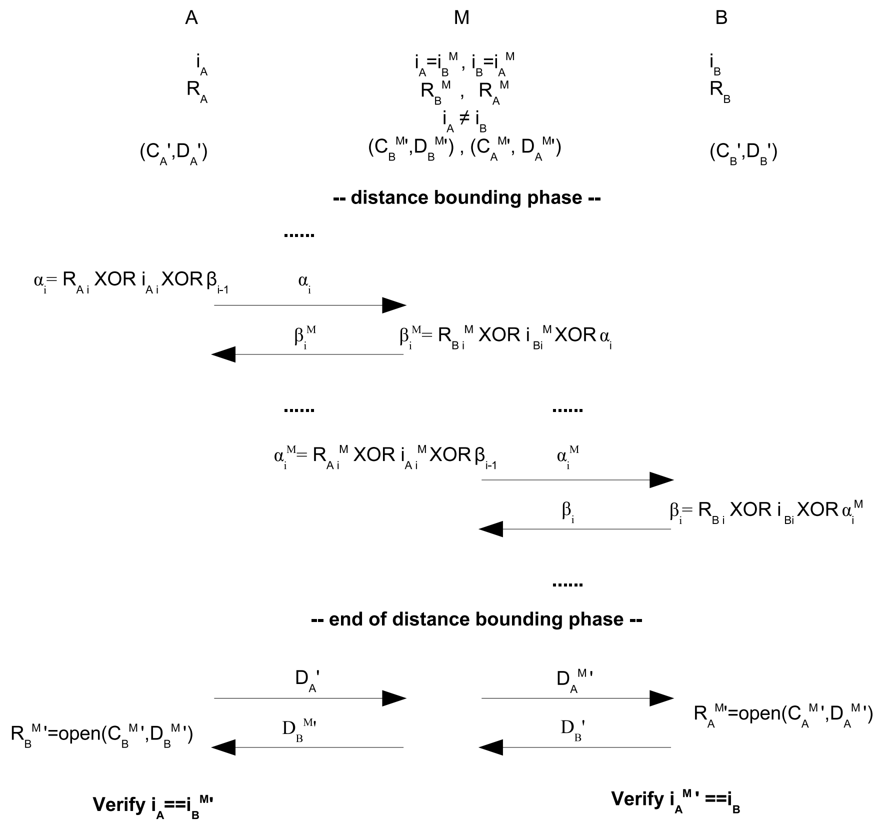

A
M
B
i_A
R_A
$i_A=i_B^M$, $i_B=i_A^M$
R_B^M, R_A^M
$i_A \neq i_B$
i_B
R_B
(C_A', D_A')
$(C_B^{M\prime}, D_B^{M\prime})$, $(C_A^{M\prime}, D_A^{M\prime})$
(C_B', D_B')
-- distance bounding phase --
$\alpha_i = R_{Ai}$ XOR i_{Ai} XOR β_{i-1}
α_i
β_i^M
$\beta_i^M = R_{Bi}^M$ XOR i_{Bi}^M XOR α_i
$\alpha_i^M = R_{Ai}^M$ XOR i_{Ai}^M XOR β_{i-1}
α_i^M
β_i
$\beta_i = R_{Bi}$ XOR i_{Bi} XOR α_i^M
-- end of distance bounding phase --
D_A'
$D_A^{M\prime}$
$R_A^{M\prime}$=open$(C_A^{M\prime}, D_A^{M\prime})$
$R_B^{M\prime}$=open$(C_B^{M\prime}, D_B^{M\prime})$
$D_B^{M\prime}$
D_B'
Verify $i_A == i_B^{M\prime}$
Verify $i_A^{M\prime} == i_B$

Figure 3.12: MITM attack of DH with distance bounding (DH-DB)

bounding protocol is not possible, any one far from Alice and Bob cannot pass final verification. Distance enlargement can be avoided if no one is between Alice and Bob.

3.5 Berkeley Echo

3.5.1 Description

As it has been shown in the previous protocols, the security of the protocol, also relies on the distance measurement between the verifier and the prover.

A protocol proposed in [16] called 'Echo' protocol securely verifies the location claims. As stated in the original paper, the protocol proposed is focused on in-region verification instead of secure location determination. In the former topic, the protocol verifies the location claimed, accepts or rejects that claim. In the latter topic, the verifier attempts to securely discover physical location of the prover [16]. This protocol takes into account of the security influenced by the processing time of the device and the communication time. The protocol constructs a *Verifiable region* called Region Of Acceptance (ROA) that ensures the prover only has negligibly small success probability. The later section will show that in some cases the ROA can be reduced to be so small, which makes the protocol infeasible.

This protocol uses techniques that similar to others previously described, which the verifier sends several challenge bits and expecting the prover to respond immediately. However, in real systems, there will always be some processing delay, denoted as Δ_p. The prover need to inform the verifier that the processing delay experienced. The verifier has to subtract the time he measured with the processing delay declared, to yield a time value corresponds to the distance. The problem occurs when a dishonest prover with an advanced device, which has almost no processing delay, but claims a false value of Δ_p. The verifier has no way to verify the amount of processing delay claimed, therefore, the prover may cheat the location by $\Delta_p \cdot s$ meters, where s is the speed of sound. The solution to such problem was to reduce the verifiable region by $\Delta_p \cdot s$ amount. Resulting 'Region of Acceptance' (ROA) as:

$$ROA = R - \Delta_p \cdot s \tag{3.23}$$

and denoted as ROA(v,Δ_p), where R in this equation is the original verification radius. It is a circle centered at verifier v. Now, if the claimed location of the prover, is within the ROA, the verifier can verify such claim.

Such problem also occurs with communication time. The radio signal travels at the speed of light. But when radio signals are used in communications, the data transmission time will depend on the data rate of the communication hardware. In real system, especially in small embedded systems with limited power resources, to provide a reliable communication, the radio channel data rate is considerably slow. In this work, Chipcon CC1000, provides data rate of 20$kbps$. This figure is sometimes comparable with ultrasound channels [16]. Even though both channels' data rate are slow, there will be a one that is faster than another. Assuming the verifier sends challenges in a faster channel, and prover responds in a slower channel, and the nonce size is N-bit. The prover could guess few number of challenge bits $m(m < N)$ and send responds earlier. To guess m bits correctly, he has 2^{-m} success probability. During the transmission of m bits data, all N challenge bits are exposed. Depending on the distance he wishes to cheat and the data rate of the slower channel, m can be

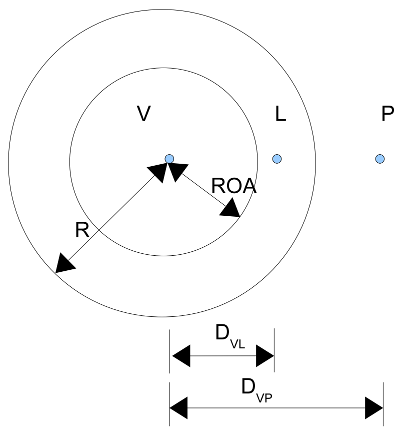


Figure 3.13: Berkeley Echo - Region of Acceptance

as small as 1 bit, the success probability now increased to 50%. The problem arises when prover's channel is slower than the verifier and try to sacrifice some response bit m to gain knowledge of $N - m$ bits of the challenge. The more he wishes to cheat on distance, the larger m will be, and hence the less success probability is. The solution proposed by [16] categorize the distance corresponds to data transmission time to be non-verifiable and contribute to the calculation of ROA. The equation for finding ROA is then:

$$ROA = R - \Delta_p \cdot s - \left(\frac{N}{b_s} + \frac{N}{b_r}\right) \cdot s \tag{3.24}$$

where b_s is the data rate of prover's channel sending reply with unit of bps, and usually ultrasound; N is the number of challenge (or response) bits. Again, R is the original verifier region. The proposed method guarantees that the location claimed to be in ROA is verifiable. The prover's success probability fully depends on the size of challenge N. In further analysis shown in section 3.5.2 on page 50, the proposed method does ensure the security, but the ROA can be reduced to be very small. The geographical view of the situation is shown in figure 3.13, where V is verifier, P is the prover,L is the location claimed [16].

3.5.2 Analysis

The fundamental procedure of the key establishment protocol was shown in figure 3.2 on page 32. For convenience the communication part concerning the

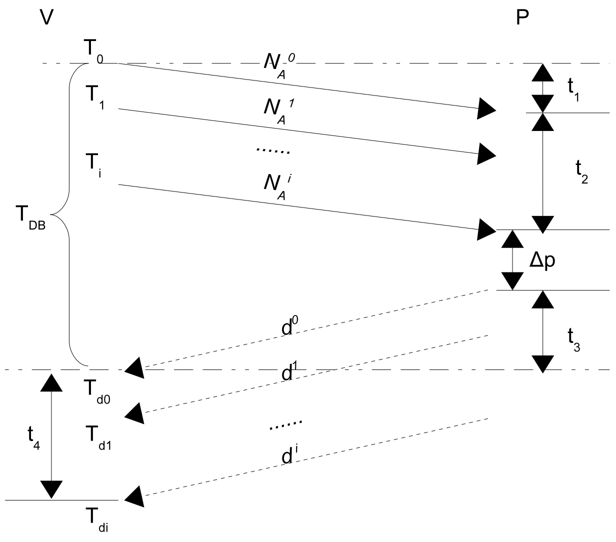


Figure 3.14: Timing details for legitimate parties

distance bounding is shown again here in figure 3.14. The timing relevant to this discussion is marked in the figure. Assuming the verifier sends message using radio, prover sends response using ultrasound. Further assume radio channel data rate is faster than the ultrasound.

In figure 3.14, t_1 is the time for the first challenge bit travels over D_{VP} (the distance between two parties); t_2 is the time to send entire radio message; Δ_p is the prover's processing time; t_3 is the time for first response bit to travel over D_{VP}, and t_4 is the time to send entire ultrasound message. Each quantity can be found by:

$$t_1 = \frac{D_{VP}}{c} \tag{3.25}$$

$$t_2 = \frac{N}{b_r} \tag{3.26}$$

$$t_3 = \frac{D_{VP}}{s} \tag{3.27}$$

$$t_4 = \frac{N}{b_s} \tag{3.28}$$

$$\Delta_p \in \mathbb{Z}^+ \tag{3.29}$$

where, c is the speed of light, s is the speed of sound in m/s, b_r and b_s are the radio and ultrasound channel data rate in bps, and $\mathbb{Z}^+$ is non-negative integers.

It has been explained how a dishonest prover can cheat within processing time and data transmission time, and this protocol has modelled the distance corresponding to such time to be non-verifiable. The Region of Acceptance was introduced to be a reduction from R by processing time and packet transmission time, shown in equation 3.24 on page 50.

Assume a prover P is located outside the verification range R, but claimed to the verifier V at location L within R. A timing diagram is shown in figure 3.15. t_1 to t_3 and Δ_p is as previously defined in figure 3.14 and T_i,T_{di} and T_{DB} was defined in figure 3.2. Further introduce t_4 as packet transmission time over ultrasound. In order to demonstrate the effectiveness of the protocol, P is assumed to be just outside R, and L is exactly on the edge within ROA. So the following formulae hold:

$$D_{VP} = D_{VL} + D_{LP} > R \tag{3.30}$$

$$D_{VL} \leq ROA \tag{3.31}$$

$$D_{VL} \leq R - (\Delta_p \cdot s) - \left(\frac{N}{b_s} + \frac{N}{b_r}\right) \cdot s \tag{3.32}$$

$$D_{VL} \leq R - (\Delta_p + t_2 + t_4) \cdot s \tag{3.33}$$

$$D_{VL} + (\Delta_p + t_2 + t_4) \cdot s \leq R \tag{3.34}$$

The distance between the verifier V and prover P is composed of distance between V and L, D_{VL}, and distance between L and P, D_{LP}. Since D_{VL} is known to be within ROA, by putting equation 3.30 into 3.34:

$$D_{VL} + (\Delta_p + t_2 + t_4) \cdot s \leq R < D_{VP} \tag{3.35}$$

$$D_{VL} + (\Delta_p + t_2 + t_4) \cdot s < D_{VL} + D_{LP} \tag{3.36}$$

the distance between prover P and the location claimed L, D_{LP}, can be found by the time for ultrasound to travel, and denoted as t_{LP} (as marked in the figure), therefore:

$$D_{VL} + (\Delta_p + t_2 + t_4) \cdot s < D_{VL} + t_{LP} \cdot s \tag{3.37}$$

$$(\Delta_p + t_2 + t_4) \cdot s < t_{LP} \cdot s \tag{3.38}$$

$$\Delta_p + t_2 + t_4 < t_{LP} \tag{3.39}$$

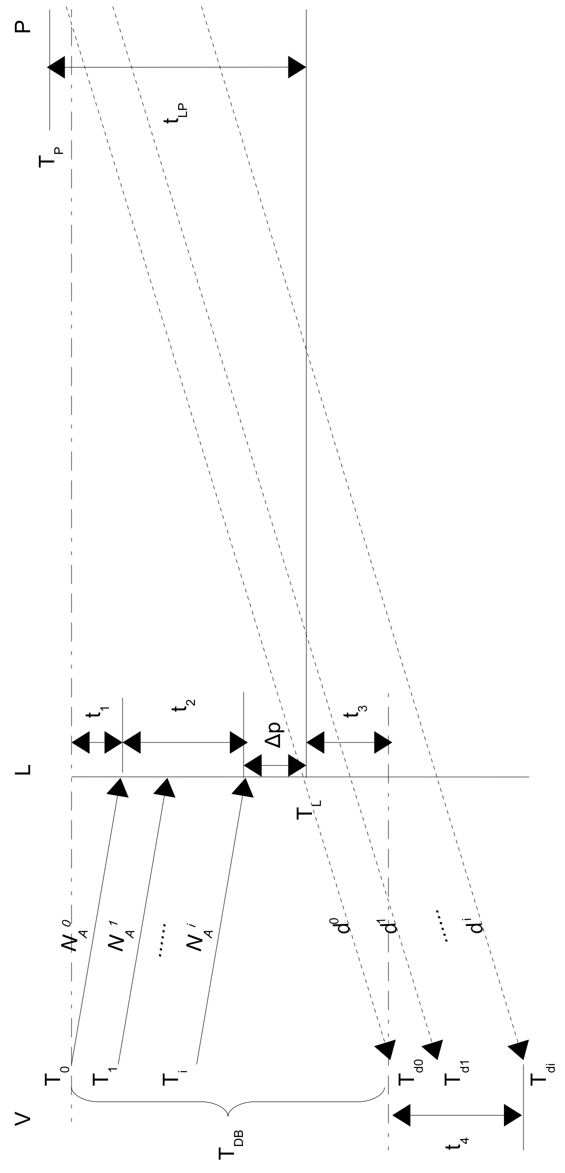


Figure 3.15: Berkeley Echo with security analysis

In order for prover P to pretend being at L, his ultrasound packet has to arrive at L at time instant T_L as noted in the figure. The time takes for an ultrasound message to travel between P and L is denoted as t_{LP} found from equation 3.39, then the time for P to start sending the packet is:

$$T_P = T_L - t_{LP} \tag{3.40}$$

where T_P is the time instant that prover start to send the packet. By rearranging equation 3.40:

$$T_L - T_P = t_{LP} > \Delta_p + t_2 + t_4 \tag{3.41}$$

It can be seen from figure 3.15:

$$T_L = t_1 + t_2 + \Delta_p \tag{3.42}$$

So, put equation 3.42 back into 3.41:

$$\begin{aligned} t_1 + t_2 + \Delta_p - T_P = t_{LP} &> \Delta_p + t_2 + t_4 && (3.43) \\ t_1 - T_P &> t_4 && (3.44) \end{aligned}$$

Assume t_1, the time for radio signal to travel between verifier V and L, to be zero.

$$\begin{aligned} -T_P &> t_4 && (3.45) \\ T_P &< -t_4 && (3.46) \\ |T_P| &> t_4 && (3.47) \end{aligned}$$

Again, T_P is the time when P start sending the packet. Since t_4 is the time spent to send the entire packet, it can't be negative, therefore T_P has to be negative. That indicates the prover has to send the responses before time '0'. By taking numerical values of T_P as equation 3.47, T_P is larger than the packet transmission time. In other words, in order for P to pretend being at L, he has to send entire packet before the verifier sends the first challenge. By doing so, the protocol ensures the challenges are exposed only after the cheater sent the response. The cheater can provide a false position in this case, but he cannot provide correct response. Since the cheater has no chance to gain knowledge about the challenge before he sends any response bit, the success probability to break the authentication is only 2^{-N}.

The above argument proves that this protocol does effectively verify the location claimed by the prover, when the location claimed is within the Region of Acceptance. Claims will be refused otherwise.

However the size of ROA may not be realistic. As shown in equation 3.24 on page 50, assuming the processing time is bounded and cannot affect the ROA significantly, ROA will be decrease if N increases. ROA is also affected by ultrasound channel data rate. The slower the data rate, the smaller ROA will be. As what is usually seen, the ultrasound channel is rather slow. In this work, the ultrasound transmission achieves about $17bps$. This means for $N = 16$ bits, and assume no processing delay, the verifier must has R of 321 meters in order to have 1 meter ROA (assuming radio channel is very fast) i.e:

$$\begin{aligned} ROA &\approx R - \frac{N}{b_s} \cdot s \qquad (3.48) \\ ROA &\approx 321m - \frac{16b}{17bps} \cdot 340m/s \\ ROA &\approx 321m - 320m \\ ROA &\approx 1m \end{aligned}$$

A MATLAB program has been written to demonstrate the use of this protocol, and to illustrate the deficiency of the protocol. The MATLAB program source code is shown in Appendix F. The results are shown in the following.

Figure 3.16 shows the simulation setup. The verifier is placed in the center, coverage the range of 500 meters in radius. The prover is just outside the coverage, 501 meters away from the verifier. The location prover claimed is just inside ROA, which is 176 meters. The radio channel is much faster than the ultrasound. The size of the nonce (challenge string) used for authentication is 16 bits. The ROA(P) is the ROA only consider processing delay, found from

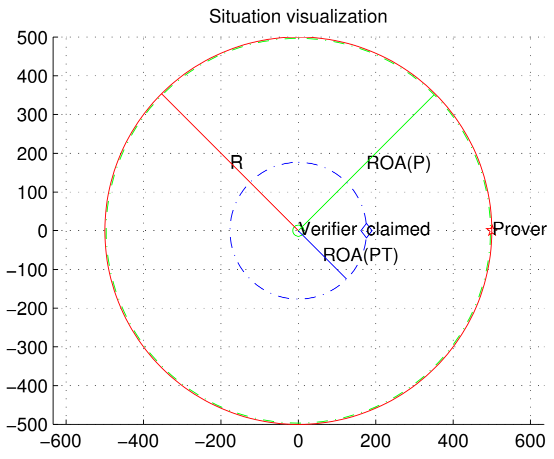


Figure 3.16: MATLAB simulation of Berkeley Echo protocol

equation 3.23; the ROA(PT) is the ROA that includes both processing delay and transmission time, found from equation 3.24. In the following discussion, 'ROA' refers to ROA(PT).

As shown from figure 3.17, in order to have ROA of 100 meters, the verifier's radius must be at least 424 meters.

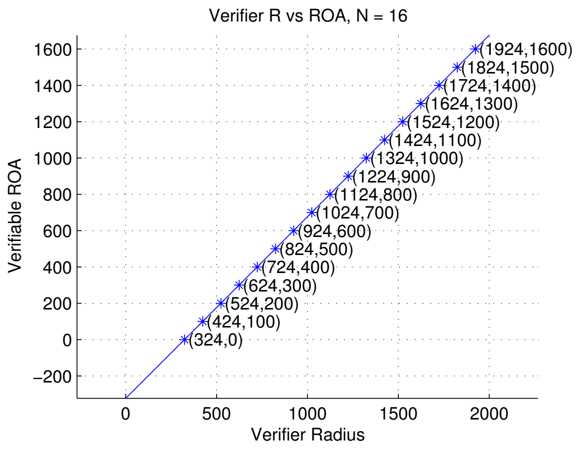


Figure 3.17: Verifier ROA and radius R

Figure 3.18 further demonstrates the effect of ROA influenced by the size of N and bit rate. The axis are the nonce length N (bits), verifier radius R (meters), and the ROA shown as the vertical axis. There are four planes in the figure. Each plane indicates different bit rate under evaluation. As can be seen, the larger N is, the smaller ROA is going to be. The rate of change is determined by the data rate. The slower the data rate is, the greater the ROA is affected. The plane for data rate of 5bps is much more steep than the one with 17bps.

Figure 3.19 shows the effect by using this protocol. The prover P and verifier V are $500m$ away. P claims to be at L, which is $176m$ away from V. The vertical axis is the time line. At time 0, V sends a challenge string to P. P has to reply to the challenge. It is very clear that, in order to cheat on the distance, the prover has to send the entire packet of response (on the P side) before the first challenge bit is being sent (on the V side). In such a way, the prover has no chance to gain knowledge of challenges. The probability of success by breaking the message authentication is a function of nonce size, as $2^{(-N)}$. The larger N, the smaller success probability.

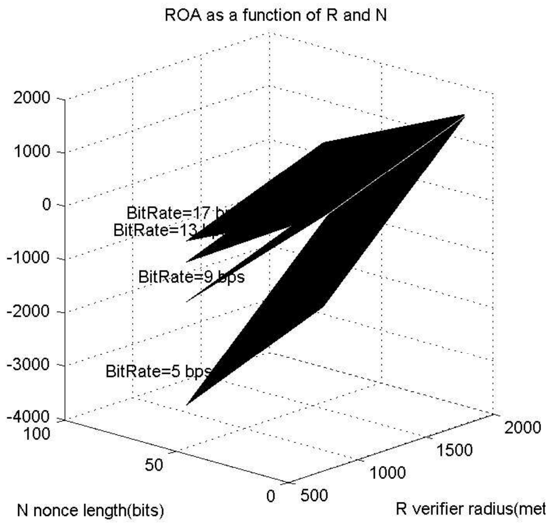


Figure 3.18: ROA as a function of N and data rate

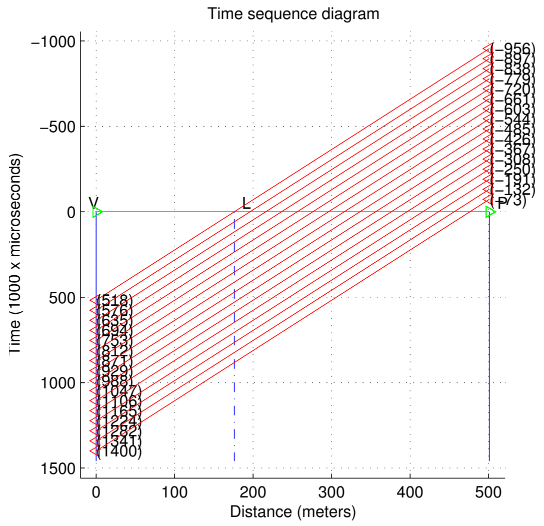


Figure 3.19: The time sequence diagram

3.6 Cambridge DBP

3.6.1 Protocol description

Similar to section 3.5.1, a distance bounding protocol used for RFID tags has been proposed in [17]. This protocol is to prove to the verifier that a RFID token is located not more than a distance upper bound from the verifier. Like other protocols, this protocol also uses the cryptographic function $h(\cdot)$, i.e. a pseudo random function in this case, and distance bounding for location verification. This protocol has three phases. The procedures of the protocol is shown in figure 3.20 [17].

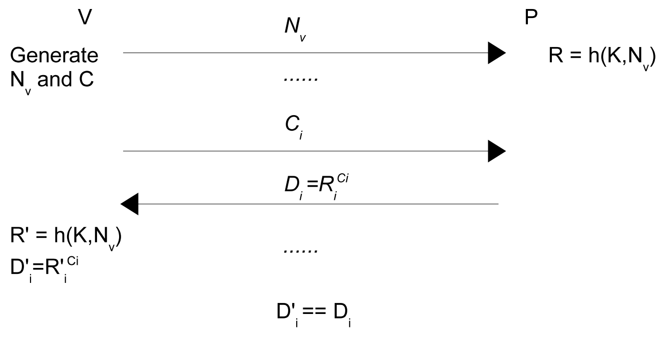


Figure 3.20: Cambridge DBP

Two parties are involved in the execution of the protocol. A verifier V, i.e. an RFID reader, and a prover P, i.e. an RFID token. Both parties share secret key K, and both agree on the pseudo random function $h(\cdot)$. In the first phase of the protocol, the verifier V sends a nonce N_V to the prover P. The prover will compute R by using function $h(\cdot)$ with secret key K and N_V. Rather than producing an n bits sequence as in the previous protocols, this protocol generate a $2n$ sequences, denoted as $\mathbf{R}$. The result $\mathbf{R}$ can be represented as a 2-by-n matrix. Each row has n bits sequence, and there are two rows.

$$\mathbf{R} = \begin{pmatrix} R_1^0 & R_2^0 & R_3^0 & \dots & R_n^0 \\ R_1^1 & R_2^1 & R_3^1 & \dots & R_n^1 \end{pmatrix}$$

The protocol enters second phase, i.e. the distance bounding. The verifier generates a random string C and resets a counter i. It sends one bit C_i to prover as a challenge and expecting a response. Unlike the original distance bounding in [15] and other related protocols, this protocol does not compute simple cryptographic function, such as 'XOR'. Instead it uses the challenge C_i and counter i as a entry to the matrix $\mathbf{R}$. i is used to select the column, C_i is used to select the row. After received the challenge, the prover selects the correct entry, and sends it back to the verifier as the response. Both parties increment the counter i. The time between the challenge and the response is measured. The action required for the prover to perform can be very fast and efficient. Since the matrix can be, in computer system, a two-dimensional array. To fetch an item only needs to read a memory location. The hardware overhead taken to access the first memory location is the same as accessing the last. The distance bounding repeats n times. In the last phase of the protocol, the verifier stores the responses as D. It also computes the pseudo random function $h(\cdot)$ with K and N_V and denoted as matrix $\mathbf{R'}$. By using C_i and i to address the item in the matrix, a vector D' can be found. Verifier compare D' and D, success if they are the same, failed otherwise.

3.6.2 Analysis - Distance shortening

This protocol has many similarities with what previously introduced. The pseudo random function $h(\cdot)$ with secret key K performs message authentication. The pseudo random function is assumed to be collision resistant. The protocol invokes the distance bounding with rapid single bit exchange. The time between the challenge and the response is proportional to the distance between the two. The generation of the response is somewhat different. The challenge C_i and the current counter i acts as an index to the entry of the response matrix $\mathbf{R}$. The counter i indicates the column that is selected, which is known to both parties; the challenge C_i will determine which row to select. Since the response matrix contains binary numbers, the i^{th} column of the matrix forms a column vector, $\begin{pmatrix} R_i^0 \\ R_i^1 \end{pmatrix}$ and there are only four possibilities:

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

It is easy to see that the uncertainty comes only when the rows of the responses are different. When the rows of the matrix are the same, the response bit will be independent of the challenge. Therefore, it gives the attacker a space to cheat. since the generation of the response matrix is a pseudo random function, the likelihood for the binary number from two rows depends on the distribution of the function itself. When the row vector R^0 is entirely the same as R^1 then the response is completely independent of the challenge. There will be no insurance of whether the prover is cheating or not. On the other hand, R^0 is the inverse, (logical NOT function) of R^1, it is similar as using 'XOR' in the previous protocols.

In the response matrix $\mathbf{R}$ with n columns, and if m columns are having the same rows, the dishonest prover can cheat on the distance in m columns without effort. For the rest of the columns, the cheater has to guess what challenge is. The probability that a prover can cheat successfully is reduce to $2^{(n-m)}$.

3.6.3 Analysis - Man in the middle attack

A MITM attack can be attempted with this protocol. An attacker only has knowledge about nonce N_V, since it is transmitted in the open air, and he has access to the specification of the function $h(\cdot)$. The attacker has no knowledge about the secret key K shared with prover and the verifier.

The attacker cannot compute the correct response matrix $\mathbf{R}$, he has to gain the knowledge from the prover. This can be realized by attacker accelerating the clock signal sending anticipated challenge C'_i before the verifier's challenge C_i [17]. The prover will reply to the attacker corresponding response D_i. If $C'_i = C_i$, the attacker can use D_i as reply, and the probability of $C'_i = C_i$ is one half. In case that $C'_i \neq C_i$, the response got from the prover $R_i^{C'_i}$ will not provide any value to the attacker. He has to guess $R_i^{C_i} \in \{0, 1\}$. The probability of success will be again one half. In order word, the attack fails only when $C'_i \neq C_i$ and he guessed the wrong response, which gives probability $\frac{1}{4}$. For n-bit challenges, the attack succeed with probability $\left(\frac{3}{4}\right)^n$.

3.7 SECTOR-MAD

3.7.1 Description

Another protocol developed in [18] introducing 'SECTOR' a set of mechanisms for secure verification of the time of encounters between nodes in multi-hop wireless networks. The 'SECTOR' is used to prevent wormhole attacks in ad hoc networks. The wormhole attack was introduced by Yih-Chun Hu et al. in [19]. In [18], Mutual Authentication with Distance Bounding protocol (MAD) was proposed. The protocol uses the distance bounding technique to determine the upper bound on the physical distance between two parties. Thus prevents false encounter certification caused by wormhole attacks. Even though this protocol is not proposed particularly for the purpose related to this work, the protocol does provide a good use of distance verification and message authentication. The distance verification and message authentication are the major techniques

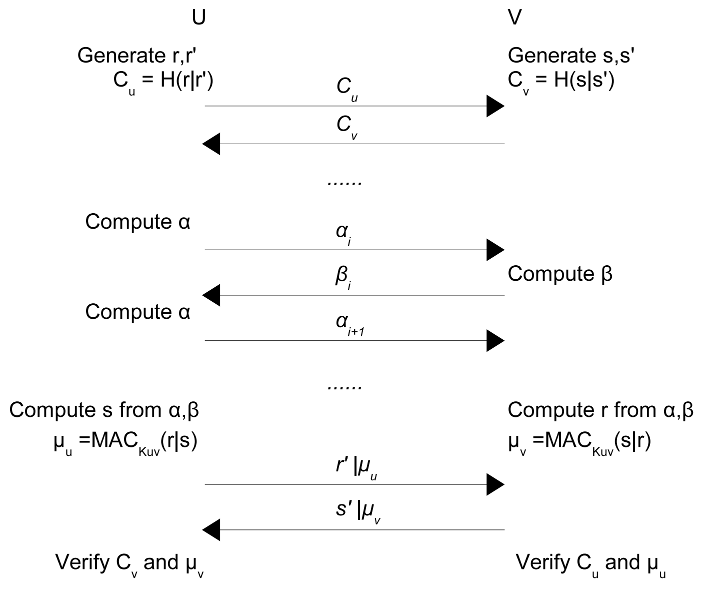


Figure 3.21: MAD protocol

used in this work.

This protocol is composed of three phases, initialization phase, distance bounding phase and authentication phase. The demonstration of the protocol is shown in figure 3.21 [18].

In the initialization phase, two parties each generates two random numbers, (r,r') for party U, and (s,s') for party V. They are represented as binary strings. r and s are with length l, whereas r' and s' are length l'. The two parties will commit the random numbers to each other, by using any commitment scheme. In [18], collision resistant one-way hash function H is used, i.e.:

$$U: \quad c_U = H(r|r') \tag{3.49}$$
$$V: \quad c_V = H(s|s') \tag{3.50}$$

c_U and c_V will be exchanged via communication channel.

The protocol enters distance bounding phase. In each steps, U will send V one bit denoted as α_i, and V will also send U one bit β_i. i.e.:

$$\begin{aligned} U: \quad \alpha_1 &= r_1 \\ \alpha_i &= r_i \oplus \beta_{i-1} \qquad (i = 2, \ldots, l) \\ V: \quad \beta_i &= s_i \oplus \alpha_i \qquad (i = 1, \ldots, l) \end{aligned}$$

For the first bit of α, it is just r_1 as a challenge to V. In return, V will XOR α_i with s_i as a response. U measures the time taken between sending of α_i and reception of β_i. Similarly, the β_i has just been received by U acts as a challenge. U should not hesitate, he should perform XOR of r_i with β_{i-1} and send the result back to V. These steps are repeated l times. After l steps of distance bounding, the protocol enters final authentication phase.

The authentication phase is mainly to authenticate that s obtained by U was initially committed by V; at the same time, r got by V was the same U committed. So that both U and V knows they ran the distance bounding with each other. For U to authenticate s, he has to compute all the bits of s from α and β. It is known that:

$$\begin{aligned} [\alpha_1, \alpha_2, \ldots, \alpha_l] &= [r_1, r_2, \ldots, r_l] \oplus [0, \beta_1, \ldots, \beta_{l-1}] \\ [\beta_1, \beta_2, \ldots, \beta_l] &= [s_1, s_2, \ldots, s_l] \oplus [\alpha_1, \alpha_2, \ldots, \alpha_l] \end{aligned}$$

More formally,

$$\alpha_i = r_i \oplus \beta_{i-1}, \qquad i = 1, 2, \ldots, l; \quad \beta_0 = 0 \tag{3.51}$$

$$\beta_i = s_i \oplus \alpha_i, \qquad i = 1, 2, \ldots, l \tag{3.52}$$

Therefore, s and r can be found by U and V:

$$V: \quad \bar{r_i} = \alpha_i \oplus \beta_{i-1}, \qquad i = 1, 2, \ldots, l; \quad \beta_0 = 0 \tag{3.53}$$

$$U: \quad \bar{s_i} = \beta_i \oplus \alpha_i, \qquad i = 1, 2, \ldots, l \tag{3.54}$$

and denoted as $\bar{r_i}$ and $\bar{s_i}$ respectively to indicate as a copy after transmission.

In order for U to authenticate $\bar{s}$, message authentication code (MAC) is used. In addition, the protocol also provide a way to check the integrity of (s, s') and (r, r'), namely using the hash function $H(\cdot)$. U recomputes $\bar{c}_V$ from $H(\bar{s}|s')$ by himself and compares with the one received, c_V. Similarly, V can compute $\bar{c}_U$ from $H(\bar{r}|r')$. However, r' and s' has not been exchanged. The r' and s' are exchanged with MAC in the following way.

Based on $\bar{r_i}$ and $\bar{s}$, U and V compute a message authentication code (MAC):

$$V: \quad \mu_V = MAC_{K_{uv}}(v|u|s_1|\bar{r_1}|\ldots|s_l|\bar{r_l}) \tag{3.55}$$

$$U: \quad \mu_U = MAC_{K_{uv}}(u|v|r_1|\bar{s_1}|\ldots|r_l|\bar{s_l}) \tag{3.56}$$

where K_{uv} is the secret key between V and U. U sends μ_U together with r' to V and V sends μ_V together with s' to U. The communication ends at this point.

Now U has $(\bar{s}, \bar{s}')$, he can check c_V and μ_V.

$$U: \quad \bar{c_V} = H(\bar{s}|\bar{s}') \tag{3.57}$$

$$\bar{\bar{\mu_V}} = MAC_{K_{uv}}(v|u|\bar{s_i}|r_i), \qquad i = 1, 2, \ldots, l \tag{3.58}$$

s and s' are authenticated by U only when $c_V = \bar{c_V}$ and $\mu_V = \bar{\bar{\mu_V}}$. As can be seen from equation 3.50, and 3.57, the former holds if and only if $s = \bar{s}$ and $s' = \bar{s}'$, and latter holds from 3.55 and 3.58 if and only if $s = \bar{s}$ and $r = \bar{r}$:

$$c_V = \bar{c_V} \quad \text{iff} \quad s = \bar{s} \quad \text{and} \quad s' = \bar{s}' \tag{3.59}$$

$$\mu_V = \bar{\bar{\mu_V}} \quad \text{iff} \quad s = \bar{s} \quad \text{and} \quad r = \bar{r} \tag{3.60}$$

In other word, since party U and V has secret key K_{uv}, the μ_U computed from MAC can be used for V to authenticate r; and μ_V can be used for U to authenticate s. To authenticate r', V can compute hash over $(r|r')$ and check with C_U; and to authenticate s', U can compute hash over $(s|s')$ and check with C_V.

Similarly, to authenticate (r, r'), V needs to check c_U and μ_U:

$$V: \quad \bar{c_U} = H(\bar{r}|\bar{r}') \tag{3.61}$$

$$\bar{\bar{\mu_U}} = MAC_{K_{uv}}(u|v|\bar{r_i}|s_i), \qquad i = 1, 2, \ldots, l \tag{3.62}$$

r is authenticated by V, if the following equations are satisfied:

$$c_U = \bar{c_U} \quad \text{iff} \quad r = \bar{r} \quad \text{and} \quad r' = \bar{r}' \tag{3.63}$$

$$\mu_U = \bar{\bar{\mu_U}} \quad \text{iff} \quad s = \bar{s} \quad \text{and} \quad r = \bar{r} \tag{3.64}$$

3.7.2 Analysis - Distance shortening

Two parties meet each other in the ad hoc network. The MAD protocol can be invoked to authenticate each other. The execution style of distance bounding is rather similar to the protocol shown in section 3.4. In each step, only involves one bit of challenge and response. The response β_i is a function of the challenge α_i. The prover V seen from U cannot send response earlier without knowing the challenge. If the prover guessed a challenge, the success probability for each distance bounding step will be $\frac{1}{2}$. For l bits, where l is a big integer e.g. 128 bits, the success probability will be only 2^{-l}, which is considerably small. Since the protocol is symmetrical, the same property applies to U. The detailed reasoning and analysis of distance shortening attack please refer back to section 3.1.3 and 3.2.2.

3.7.3 Analysis - Man in the middle

The MITM attack can be attempted in this protocol. In this section cooperative attack is discussed, i.e. two intruders are in the attack scenario. Assuming the commitment scheme used in the protocol is secure, this discussion will focus only on distance bounding.

The attack introduces two intruders $\bar{V}$ and $\bar{P}$ denoting fraudulent verifier and prover. $\bar{P}$ will pretend to be P and prove to the verifier V, similarly, $\bar{V}$ will communicate with P. The consequence is that honest prover P believes its identity has been verified by V, and further communication can proceed between P and V, however this is not what actually happened. It seems for P and V that they are communicating with each other, but in fact, $\bar{V}$ and $\bar{P}$ are standing in the middle. Message sent from one honest party can be dropped, modified, forged and injected. An example, also easily performed, of such attack is shown in [15]. The fraudsters $\bar{V}$ and $\bar{P}$ complete the distance bounding with one party entirely, then with another. For conventional Man-in-the-Middle attack, there is no restriction on the intruder's physical position.

By using distance bounding, the party that involved in the protocol cannot shorten the physical distance between the parties. If the distance between fraudulent prover $\bar{P}$ and verifier V is longer than P to V, then as explained in section 3.7.2, $\bar{P}$ cannot shorten the distance with success probability greater than 2^{-l}. In other word, $\bar{P}$ and $\bar{V}$ has to be physically between V and P. Now the problem can be simplified. The intruder basically cannot broke the protocol from somewhere further away than the honest prover. For situation where the intruder is physical between the V and P, a solution proposed in [14] can be used. The concept 'Integrity Region' has been explained in section 3.1.5. The 'Integrity Region' is a 3-D space (two spheres) that is each centered at one party with radius d that equal to the distance between the two parties. During the execution of the protocol, the two parties look around and visually make sure that there is no third party within the 'Integrity Region', then the protocol just executed is reliable.

The original use of the protocol is for secure tracking of node encounters. The MITM cannot be performed either. The message authentication code computed in the protocol uses the secret key shared between the two parties. Without knowing the key, the intruder's message will not be authenticated.

3.8 Conclusion

This chapter demonstrates several protocols that use distance bounding protocol to authenticate the participant. An appetizer introduces the use of distance information to verify the sender of the message with one round distance bounding in key establishment. This protocol leads to 50% attack success rate. To obtain better security, Distance bounding with bit exchange protocol is developed. This protocol has an n rounds distance bounding phase. The bigger n is, the harder to cheat. After n rounds distance bounding, the receiver and the message sender are bounded by the distance between the two. By visual inspection, the user can verify if the participant of the protocol is the one he believed. The attacker only has $2^{(-n)}$ probability to break the protocol. If the n is chosen large enough, the probability to be attacked can be significantly small. To speed up the protocol execution, distance bounding with word exchange uses the same idea, but with less security insurance. The DH-DB protocol has been proposed by Capkun et al. The protocol is developed based on DH-SC. The string comparison has been automated with help from distance bounding. The user mutually authenticate each other within one protocol run. The probability of a successful attack is also less than $2^{(-n)}$.

The Berkeley Echo protocol uses distance bounding to verify location claims. This proposal models several important factors encountered in practical implementation. They considered the device processing time as well as transmission delays. An attacker can successfully cheat the distance bounding from the uncertainty provided by the above two factors. The use of ROA eliminates such attack. However the solution they provided is effective, but not very efficient. Similarly, Cambridge DBP protocol verifies the location of an RFID token. The protocol used a response matrix instead of XOR function. The attacker has probability of $(\frac{3}{4})^n$ to successfully break the protocol. The MAD protocol proposed in secure tracking of node encounters also uses distance bounding protocol. This protocol has a similar fashion as used in DH-DB protocol, and shares the same security property.

Essentially the distance bounding protocol will be used in this work for key establishment. This chapter illustrated several examples that successfully adopt the distance bounding protocol. In next chapter, A key establishment protocol with distance bounding will be discussed and implemented. The implementation procedure and issues will be addressed.

Chapter 4

Implementation of Key Establishment Protocol based on Ultrasound Distance Bounding

This chapter will focus on how secure key establishment protocol has been implemented, and implementation issues considered. The Cricket in door location system from MIT has been used in this project. The choice of key establishment protocol among different solutions is first described. The Cricket system is then briefly described. The fundamental concepts about the embedded operating system TinyOS that is running on Cricket system is explained. The implementation of the protocol is divided into several small modules that are the building blocks to the final implementation. Each building block is described, implementation issues and design decisions are discussed along the way.

4.1 Introduction

In Chapter 2, several different key establishment techniques have been described. A session key can be decided by one party and established with use of *public key cryptography*. Each party possesses of a key pair. Encryption using the sender's private key can be used to authenticate the message by the receiver. Encryption using the receiver's public key ensures only the true recipient can open the secret message. The use of public key cryptography is a secure measure, but the computational cost is too high. In this work, the Cricket system is used, it is a simple device with very limited resources. The use of public key cryptography is basically not feasible. The *Diffie-Hellman* method seems to be very appropriate. The direct implementation of DH method has been found flawed. Several other proposals are available. The *Pre-Authentication* is a good start. The use of location limited channel such as infrared ensures no other devices could intercept with the authentication. However, the Cricket system does not provide any location limited communication channel. *Password Authenticated Key Exchange* uses a shared weak password to derive a stronger secret key. It is extended from the 'EKE' method. This method requires the user to provide a secret password in the field, and typed into the device. Such requirement cannot be satisfied by the Cricket system. Similarly, the *Seeing is Believing(SiB)* requires a digital camera and a display to complete the key establishment. The *Loud and clear*(L&C) uses a specific library to generate text strings, and 'Text-to-Speech' library to provide audio signals. Those requirements cannot be fulfilled in this work. The *Short string comparison (MANA)* and *DH-SC* method also requires a keypad or an output for string comparison, but they moved one step further for automated method. The *DH-DB* is a protocol uses DH method with distance bounding. The DH-DB protocol extends the DH-SC with automated string comparison. The 'success/failure' can be indicated by two Leds. The protocol also involves distance bounding, which provides a device localization mechanism to prevent MITM attack. This protocol doesn't require any informational input from user. The output can be displayed on a PC or iPAQ via a serial cable. The last described technique *Shake them up*, is designed for device with very limited resources. The key point of this method is to use device indistinguishability instead of existing cryptofunctions. In this work, the Cricket system is capable of handling symmetric cryptography, therefore this method is not further considered.

Chapter 3 further discussed several variant use of distance bounding protocols. Some are used for key establishment, others are used for verifying location claims. The first four protocols use the distance bounding for key establishment. *Berkeley Echo* is a verification of location claims. The significant downside of this protocol is the fact that ROA can be reduced to very small, so the protocol may not be very efficient. The *Cambridge DBP* protocol is used to verify the location of an RFID tag. The attack success probability is higher than the one found in DH-DB. The *MAD* protocol is used for mutual authentication for node encounters. This protocol assumes the two nodes already have a symmetric MAC key before protocol execution.

The 'Distance bounding - bit exchange' protocol can be seen as a simplified version of DH-DB. One execution of the protocol provides a public key information from one party to another. By exchanging role and re-invoke the protocol, both key information can be exchanged. The 'Distance bounding - Word exchange protocol' and the 'Appetizer' are two variant.

For the sake of simplicity, 'Distance bounding - bit exchange' protocol is implemented to demonstrate the principle idea behind secure key establishment with distance bounding.

In the following sections, the hardware system and software environment are explained. Several modules have been designed and implemented. Those modules are the building blocks toward protocol implementation. The next two sections will describe the hardware Cricket system, and the operating system it runs – 'TinyOS'.

4.2 Cricket System

The Cricket in door location system from MIT has been used in this work. This system provides complete hardware solution for low cost battery powered distributed embedded system. The Cricket system was designed in MIT and first described by Nissanka Bodhi Priyantha *et al* in [20] [21]. It is now commercially available from Crossbow Technologies[1].

The Cricket system benefits from the high integration of current IC technology. The small footprint makes it very attractive in many distributed embedded applications. The system consists of an 8 bits microcontroller, a pair of ultrasound transducers, a radio unit and external I/O ports[23].

The microcontroller used in the system is ATMEGA128 from ATMEL. It is an 8 bits microcontroller in Atmel AVR family[24]. AVR[2] stands for 'Advanced Virtual RISC', is a RISC microcontroller family. This microcontroller has many advanced features. The microcontroller is currently clocked at 7.3728 MHz

[1]Crossbow Technologies, http://www.xbow.com/

[2] AVR was originally conceived by Alf-Egil Bogen and Vergard Wollan at Norwegian Institute of Technology and further developed at Atmel Norway.

frequency, it is able to operate at 16 MHz. It has 32 8-bit general purpose working registers. Unlike many other 8051 compatible microcontroller family, ATMEGA128 executes most of the instructions in single clock cycle and has an on-chip 2-cycle multiplier. There are 128K bytes In-System Programmable (ISP) Flash memory, 4K bytes Internal SRAM. This microcontroller also features from several on-chip peripheral components, i.e. two 8-bit timer/counters, two expanded 16-bit timer/counters; several PWM (Pulse Width Modulation) channels, 8 channels of 10-bit ADC (Analog-Digital Converter). Two-wire serial interface, dual USART, and SPI serial interface enhanced the controller with connectivity and extensibility. IEEE 1149.1 compliant JTAG boundary scan supports on-chip debug capability. Six sleep modes, software selectable clock frequency make this controller suitable for many power-efficient and power-aware applications.

The ultrasound transducer pair is from Kobitone audio company. On each Cricket system, there is a transmitter and a receiver. Both operate at 40 KHz center frequency, bandwidth of 2 KHz. The transmitter operates at 12V supply voltage, driven by MAX864 DC-DC converter [25]. The receiver detects ultrasound signal and converts into voltage, an interrupt will be generated when the voltage level is higher than a pre-defined threshold. Details about ultrasound transducer can be found from corresponding datasheet[3] in [26].

The Cricket system has a low power RF transceiver on the board. It uses an $0.35\mu m$ CMOS (Complementary Metal-Oxide Semiconductor) single chip radio solution CC1000 from Chipcon. It operates on UHF frequency band (examples of frequency allocation and radio spectrum, please refer to [28]). It is mainly intended for Industrial, Scientific and Medical (ISM) and Short Range Device (SRD) frequency bands at 315, 433, 868 and 915 MHz. It is also programmable to be used at any other frequencies in 300-1000MHz range[27]. This transceiver supports programmable output power and operates at low supply voltage. It also provides RSSI (Received Signal Strength Indication) output. Interfaced with microcontroller using SPI.

There are two external interface available on the Cricket system. One is 51-pin connector mainly used for programming the device, and the other is 9-pin RS232 serial port. There are several signals can be reached from 51-pin connector, such as external interrupt, A/D convertor input, SPI, two wire serial interface, etc. This connector is mainly used to attach the device to the programmer board to download firmware. RS232 serial port can be used for attach the device to a PC or any other devices with same port. It can be used for software debugging, or user interface with interactive functionalities.

[3]The part number of ultrasound transmitter is 255-400ST12 and receiver is 255-400SR12. They are produced and distributed by Mouser electronics.

Further details about Cricket v2 system hardware and precompiled firmware can be found in Cricket User manual in [22].

4.3 TinyOS

The scaling of technology has shown a great impact on small computing devices with low power, distributed, self-organised embedded systems. Wireless sensor network is a very good example. The sensor nodes are equipped with many different technologies such as microcontroller, A/D converter and communication transceiver, all in a chip. But it is constrained with very limited resources, such as RAM, computation power, and energy. An underlying operating system is designed to be used in such scenario, it is TinyOS. There are several paper address the need for such operating system, and described the details of TinyOS, such as [44][45] and [46]. This section will only give a very brief description and introduction to basic concepts, so to understand the implementation described in the coming sections.

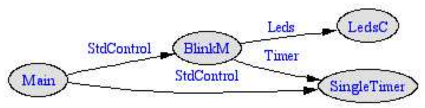


Figure 4.1: Mica2 Blink application

TinyOS communicates directly with hardware components. Each hardware component is managed and represented by a software *Module*. A *Module* can be understood as a driver to the specific hardware. Hardware components can interact and cooperate with each other, so software *Modules* can be brought together to perform some functions. In software, it is called a *Configuration*. The connection between software *Modules* (and hence hardware components) is called *wiring*. It has rather similar fashion to hardware description languages such as VHDL and Verilog. The modules and configuration that provide some functionalities is called a *Component*. The component can be a single module. It can also be a configuration that wires several modules. Each component also defines an *interface* that could communicate with another component. The interface 'possess' of two sets of functions. One set of functions is *commands* . The other set is *events*. A command is *called* by an upper layer component, and an event is *signaled* by a lower layer component due to hardware interrupt or event occurred. To *use* or *provide* an interface, the functionality of a component can be utilized.

An example is shown in figure 4.1. The 'Main' is the main component from TinyOS. This component is usually not visible from user application, but wired through 'StdControl' interface. The 'BlinkM' is a module that uses 'Leds' and 'Timer' interfaces, which in turn provided by 'LedsC' and 'SingleTimer' components. Since 'LedsC' and 'SingleTimer' are configurations, they are further wired together by several smaller components that are not shown. The 'SingleTimer' will signal a 'Timer.fired' event, which is handled by 'BlinkM' component. In this event handler, Leds will be toggled by calling a command 'Leds.redToggle'. Notice, every action performed is due to an event. The event is signaled and handled in a manner similar to 'call back' functions.

Commands and events are meant to be non-blocking. A command is a request to perform a service, e.g initiate an A/D sampling or send a radio message. The command should not wait until the message is completely sent. The event is used to signal the completion of such service. This is called a *split phase* operation. In some cases, a computational intensive operation might be needed in a command, e.g. compression or cryptofunctions. For such hardware independent algorithmic operations can be executed in a *task*. The *task* can be understood as a special type of thread. They are managed and scheduled by the operating system and executed at some later time. This allows the command and event to be more responsive. The standard TinyOS task schedular uses a non-preemptive, FIFO scheduling policy.

4.4 Data transmission over Ultrasound

In the protocol that is to be implemented, there is a need to transmit data between two parties via ultrasound. By now, the Cricket system does not provide any hardware nor software support for such functionality. This part has to be designed and implemented. This section first introduces the ultrasound primitive functions provided by Cricket. It shows the principles of data communication over ultrasound. Ultrasound transceiver is then designed and explained. A state machine is used to ensure the correct functioning.

4.4.1 Ultrasound primitives

In the Cricket system, there is a piece of program that provides ultrasound capability as primitive functions. The user program could use such primitives to send ultrasound pulses and detect the existence of the ultrasound signal. The primitive provides functions for both the transmitter and the receiver.

In the ultrasound transmitter primitives, a sequence of ultrasound pulses can be generated in each sending. The pulses consist of 8 cycles of ultrasound signal, each cycle is 25us (corresponds to $40kHz$). Therefore the duration of the pulses is 200us. The signal is captured by oscilloscope and shown in figure 4.2. It is measured on capacitor C54 right before the transducer. The tail after the 200us pulses are due to bouncing in the mechanical systems. There was several software bugs found in the original Cricket system. It exhibits slow discharging effect and timer errors due to software deficiency, which prevents the program from correct functioning. The details about the bugs found in this work and the fixes are explained in Appendix B. The circuit diagram of the Cricket system hardware construction with component annotation can be found in Appendix C[4].

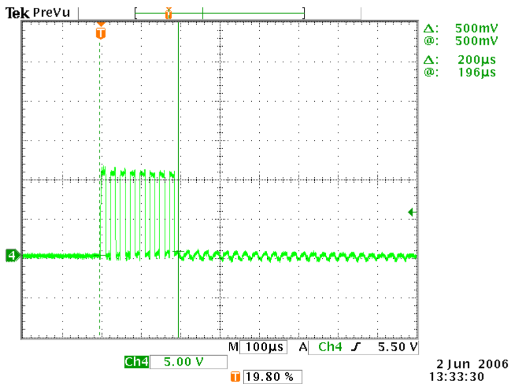


Figure 4.2: Sending Ultrasound Pulses

In the receiver, ultrasound pulses will be detected by the transducer. The signal pattern is shown in figure 4.3. It is measured just before the regulation diode D7. The additional capacitor C49 and resistor R49 together with D7 is similar to AM demodulation circuit. It is responsible for amplitude detection. The signal duration is about 1.3ms, it can be clearly seen that the signal is distorted heavily during transmission. The signal did not die out after this interval, there are small pulses following the main pulse. The reason for such occurrence is due to the echo come from any material it experienced. (Further details about US circuitry can be found in [30].)

By sending a limited number of ultrasound pulses, a signal with highest energy followed by several smaller ones can be detected. This indicates an informational relationship between the sender and the receiver. In the later section, a timing relationship is demonstrated. The situation shown in figure 4.2 and 4.3 can be

[4] or from original website: http://cricket.csail.mit.edu/

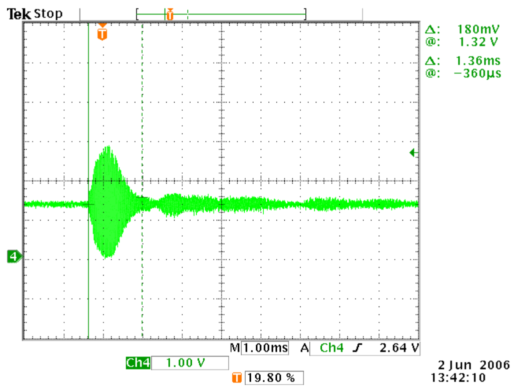


Figure 4.3: Ultrasound Pulses received

considered as an one-bit data transmission. A multi-bits data can be realized by repeating the above procedure.

4.4.2 Design ultrasound transceiver

Data rate

Based on the primitives, data transmission over ultrasound is now possible. The sender informs the receiver that an event has occurred by simply sends pulses. If no signal has been detected indicates that such event did not occur. By such method, one bit information can be represented. In data communication, an interval is usually defined for each bit, known as the bit width, has unit of *second per bit* . One bit is accommodated in such interval. Therefore, one bit can be sent to the channel after another, so that multiple bits can be transmitted. The closely related concept is *data rate*, the reciprocal of bit width. The data rate constrains the communication speed, and has unit of *bits per second* (bps).

Ideally the bit width can be determined by the signal length at the transmitter. However in this case, due to the nature of sound signal, it is more easy to be reflected and distorted, the determination of the bit width is slightly complicated. Apparently the received signal in figure 4.3 is much longer than the sending signal shown in figure 4.2. However, by extending the time in the oscilloscope, it gives a more surprising result. As shown in figure 4.4, not only one pulse is found in the received US signal, more pulses followed. Some pulses are smaller, few is even bigger. The reason for such effect is very simple, it is due to the reflection. The signal is sent to the open air, and propagates in a broadcast manner. The smaller echo is reflected from the material that absorbs the most energy, e.g. an open window, or the wall at the far end. The echo with high energy in this case, is likely due to the ceiling. The test environment took place, has a ceiling about 3 meters above the floor, the two Cricket devices under test are placed on the desk, which is about 1 meter above the floor. The transducers on the Cricket devices are all facing the ceiling. The maximum energy of the sound signal is therefore experienced by the ceiling. After traveling the round trip from sender to ceiling and back to the receiver, the distance is about 4 meters, which correspond to about $12ms$. The 2^{nd} pulse with high energy in the figure is located approximately with the same time. (Note that the time base is set to $10ms/div$)

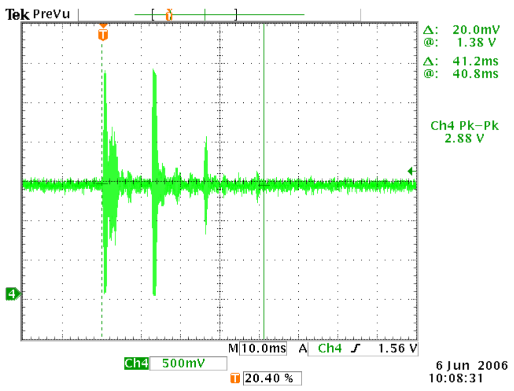


Figure 4.4: Ultrasound echoes

The echo greatly affects the signal reception. The transmitter sends one bit of information, multiple of signals are received. This phenomenon complicates the data transmission. In advanced data communication devices such as radio transceiver, similar effect such as multipath fading also exist. The common solution is to use dedicated hardware performs signal processing and use error correction code(ECC). The simplest ECC can perform Single Error Correction and Double Error Detection (SECDED). However for the current hardware construction, the Cricket system is so simple that a bandpass filtering is not even performed in ultrasound circuit (this fact can be observed in circuit diagram).

The only solution for the echoes with current hardware construction is to wait until all possible echo dies out, then sends another bit. Therefore the determination of the bit width becomes very important and affects the communication reliability.

From figure 4.4, it is observed that after sending one bit, the channel is free of echoes after about $40ms$. After the implementation and by experiment, the bit width of no less than $50ms$ is experienced. It gives an acceptable ultrasound data communication channel in the current test environment. For maximum reliability, bit width of 60 ms is used in the current setup. Longer bit width is not possible in the current Cricket system, due to the limitation of the hardware timer. It is also worth mention that in the real scenario, it is likely that the devices are in the open space where obstacles such as ceiling does not exist; and by correct mounting of ultrasound transducers and proper instruction, the transducers are facing each other. In such a way, the echoes are further eliminated providing a more reliable channel.

Communication protocol

After determine the communication data rate, a protocol is needed to specify the communication format. A protocol is a set of rules for the order in which messages of particular types are exchanged [31]. In this case, a simple protocol is needed to specify the data format, type and error handling. The basic function that are required for the protocol is to:

1. The receiver is capable of obtaining the data that has been sent by the transmitter.
2. The transceiver in both end should finish communication in finite time interval.
3. The data should be accommodated in a common unit, such as bit, byte or word.
4. An error detection mechanism should be adopted, e.g. CRC, Parity check.

The simplest protocol that can achieve such requirement is *Serial communication Protocol*, which is widely used in RS232 interface driven by a UART(Universal asynchronous receiver transmitter). It is a very economic protocol used in data communications between electronic devices and terminals. Data are transmitted in a serial manner. Usually the hardware UART is implemented in the devices. The parity check can be enabled to provide error detection. The channel is synchronised by a *Start bit*. In this protocol, the data transmission speed is defined by the *baud rate* agreed by the two terminals. Therefore to transmit a finite amount of data, the transmission time is bounded by a finite and deterministic time interval. The same property applies to the receiver. The data payload is usually one byte (i.e. 8 bits). One parity check bit follows the data payload providing error detection.

Since RS232 serial communication protocol satisfies the functional requirement, therefore it is used in this work to realize ultrasound data communication. The detail information about RS232 interface and serial communication can be found in chapter 16 in [32].

It is also worth mention that:

- Since the main focus of this work is *not* to develop and implement a reliable wireless communication protocol, further robust protocols are not investigated. However it can be left as a future work.
- The parity check has its widely known disadvantage. It can only detect single bit errors. CRC (Cyclic redundancy check) can be used instead. However it requires more computation power and implementation effort. For the sake of simplicity, the parity check is used as it was in the original RS232 standard.

4.4.3 Transceiver state machine

The hardware that realize the serial communication protocol is known as UART. There are many commercial products available. However in this work, there is no space for such expansion in hardware, it has to be implemented in the software. In hardware implementation, it is usually realized by a Finite State Machine (FSM). In software implementation, such methodology can also be used, known as Finite State Automaton (FSA). This method models the system behavior by states and transitions. Actions can be taken in state or during transition. In this work, UPPAAL is used as a tool to model the ultrasound transceiver state machine. It is capable of modeling, simulate and verify the system behavior. Further details about FSM can be found in Appendix B.6 of [33] and section 8.5 of [34]. A tutorial of UPPAAL can be found in [35].

The coarse representation of the FSA is shown in figure 4.5, and only the fundamental principle is described here. The detailed complete FSA diagram with description of every *guard,invariant,channel synchronization and etc.* is shown in Appendix D.

The left one in the figure models the ultrasound transmitter, and the one on the right models the receiver. Both start at the *idle* location. When a transmission begins, the transmitter will be signaled on *channel* ‘Send’ and taken the *transition* into the location *StartBit*. This location indicates that the transmitter sent one start bit to synchronize the channel. At the same time, it should also sent an ultrasound signal noted by *Signal=1*. By this signal, the receiver’s internal

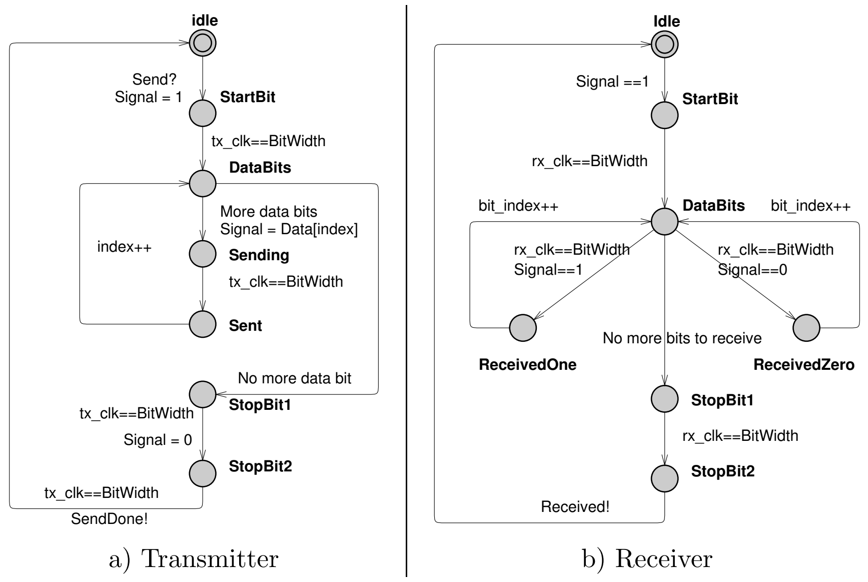


Figure 4.5: FSA of Ultrasound transceiver

clock is synchronized with the transmitter. After the synchronization signal is sent, the transmitter uses an internal clock (usually a hardware timer) to count the time elapsed. When the time is exactly equal to the bit-width pre-defined, the transition fired and bring the transmitter into location *DataBits*. From this location and here after, the transmitter will send the actual data bits to the channel. In this location, it will check if there is more data bit to be sent. If there is, it fetches the corresponding data bit and send to the channel. Again, the transmitter counts time to satisfy the signal bit-width requirement. When satisfied, it will fetch another bit to sent. If there is no more data bit, the transmitter will take the other transition from location *DataBits* and fire into location *StopBit1*. One of the purposes of *StopBit1* and *StepBit2* is to confirm with the receiver that there is no more data bits from the transmitter. Even though the number of bits in each transmission is pre-defined and known between transmitter and receiver, the stop bits give a further confirmation. The stop bits are provided by the transmitter sending logic '0' and bring the channel to idle. By using stop bits, the noise such as echo can be further reduced, so to provide a more reliable communication. The physical signal representing logic '1' and '0' is discussed in section 4.4.4.

The receiver will constantly listen to the channel. As soon as an ultrasound signal is detected, the receiver will fire the transition, go to *StartBit* location and be synchronized with the transmitter. It is now capable of receiving the data bits, as denoted by the *DataBits* location. In this location, the receiver listens to the channel for a time period defined by the bit width. If logic '1' is detected, the receiver will go into *ReceivedOne* location; if logic '0' is detected, it will go into *ReceivedZero* location. When every bits have been received, it will take the transition and go to *StopBit1* location.

The model has been extended with higher level 'Sender' and 'Reader'. The 'Sender' uses the Transmitter to send data, received by Receiver and provided to the 'Reader'. By simulation in UPPAAL, the transceiver works well. In order to prove the correctness of the model, verification in UPPAAL by simplified Computation tree logic (CTL) is performed. The verification is performed by queries expressed in CTL. Several queries has been stated and checked, they can be summarized as below:

1. There is no deadlock in transceiver.
2. The current communication taken place in the transceiver can terminate eventually. So that it does not suffer from livelock [5].
3. If no error taken place, e.g. due to noise and echo, the data received by the receiver is exactly the same as it was sent by the transmitter.
4. The communication time for a given amount of data is bounded by a deterministic value.

The above properties has been checked, and the model satisfies these properties. The formal expression of the queries can also be found in Appendix D.

4.4.4 Implementation

The protocol to be implemented follows from the RS232 standard. However, this standard was originally applied to wired communication between electronic devices, the signal logic level and physical signal representation does not apply to this work directly. Therefore, minor modification is required.

In RS232 standard, the channel is kept by constant logic '1' to represent an idle channel. In this work, sending ultrasound constantly to indicate idle channel requires huge amount of energy consumption, which is not realistic for battery

[5]Livelock is defined as a program is alive, but the processes are not going anywhere. [36] chapter 2 page 77

	RS232	Ultrasound
Logic ‘1’	-5 to -15V	200 us ultrasound pulses
Logic ‘0’	+5 to +15V	no ultrasound pulse
Channel idle	logic ‘1’	logic ‘0’
Start bit	logic ‘0’	logic ‘1’
Stop bit	logic ‘1’	logic ‘0’

Table 4.1: Modification of RS232 standard in ultrasound communication

powered device. Therefore, idle channel is represented by logic ‘0’. Similarly, the second modification is on the start bit. In RS232 standard, the start bit is represented by logic ’0’. In this work, the start bit is represented by logic ‘1’. Third, in RS232, the physical signal representation for logic ‘1’ is an electric signal between -5 and -15V, where logic ‘0’ is signal between +5 and +15V. In this work, the logic ‘1’ is by sending ultrasound pulses using Cricket primitive functions. Logic ‘0’ is by *not* sending ultrasound pulses. Notice that the Cricket primitive function sends ultrasound pulses with duration of 200us, whereas in previous section, the *bit width* is determined of about 60ms. The bit width is much longer than the actual *pulse width*. This interesting fact is due to the interference from the echo previously discovered in section 4.4.2. If the echoes can be eliminated, the *bit width* must be very close or equal to the *pulse width*. However, this is out of the scope in this work. Fourth, the stop bits in RS232 standard are represented by logic ‘1’, due to above reasons, they are modified to be logic ‘0’. The above four main changes is summarized in table 4.1. The ultrasound communication packet format is then shown in figure 4.6.

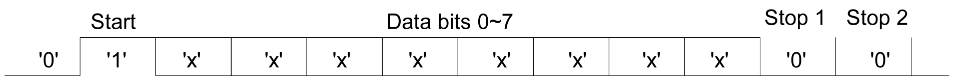


Figure 4.6: Ultrasound communication packet format

The ultrasound transceiver has been implemented using nesC language into Cricket system. The following paragraphs will explain some implementation issues, which is closely related to nesC language. For the readers that is not interested about the details, may proceed directly to section 4.5. The following paragraphs explains the software structure, program interface, how ultrasound sender and detector is implemented, how data rate are kept in communication and the arrival time of the packet.

The <u>software structure</u> of US transceiver is simple. The *interface* file is provided by ‘usTransceiver.nc’ and following the nesC naming convention, the *con-*

figuration file is 'usTransceiverC.nc' with *module* file 'usTransceiverM.nc'. The usTransceiverM module uses UltrasoundControl interface, which provides ultrasound primitive functions. The events that are *signaled* from UltrasoundControl are also handled by usTransceiverM.

The program interface allows the US transceiver to be used by higher level modules. There are two major interface provided, a command 'SendWord' and an event 'DataReceived'. The 'SendWord' can be invoked to start an US transmission with data in the parameter field. The data can be any bits (no longer than 15) defined in the header file as a constant. When an US packet is received, the data will be extracted and signaled as an event to the higher level module. If the parity check is needed, the 'SendWord' command also computes the parity value of the data, and append it at the end of the data word. Similarly the receiver also verifies the correctness of the data word by matching the parity value.

To send and detect ultrasound, primitive functions must be invoked. A sequence of ultrasound pulses are sent by calling 'SendPulse' command. Eight cycles of $40kHz$ in total $200us$ signal is generated. Such signal can be detected by the receiver and generates an hardware interrupt, which signals an event named'PulseDetected' that is handled by usTransceiver.

To keep both transceivers with the same pace, both end must have same data rate. *One way* is to use a software timer. By setting the timer interval equal to the bit width of $60ms$, a timer-fired event will occur when the time elapsed. The downside of this approach is that the software timer is not very accurate. Start and stoping a timer also adds overhead which makes the timing property even worse. *The second approach* is to play a trick; reuse ultrasound hardware timer. By using ultrasound primitive functions, the US detector can be enabled or disabled. Once enabled, the detector will listen to the channel. A dedicated hardware timer is also activated. When the timer expires, the detector will be turned off and an event will be signaled. By this approach, an event with an accurate timing property can be realized. In the event handler, the ultrasound detector can be restarted again, a time out event can be signaled periodically. The time out interval is hard coded by a constant defined in the header file. By such method, the same data rate in both transmitter and receiver can be kept according to the requirement.

An additional event is signaled in the usTransceiver to indicate Time of Arrival of the first bit in the packet. This event is named 'usMsgDetected'. It will give the handler a time reference of when the message actually reached the receiver. This can be used for distance measurement and ranging. Further details on this topic will be elaborated in section 4.6

The source code of Ultrasound transceiver is shown in Appendix G.1.

4.5 Radio transmission

The radio communication in Cricket system is well defined. It is based on Mica2 sensor node. There have been working programs developed for Cricket radio module. A RF transceiver module is developed in this work, it simply combines several radio communication module from Cricket system into one module. The transceiver uses interface 'BareSendMsg' and 'ReceiveMsg', which is responsible for sending and receiving. A command is supplied to change the role of the transceiver (as being transmitter or receiver). The radio packet is defined by 'TOS_Msg' structure, which defines the destination address, packet type, group ID, data payload, and length. A field for CRC code is also available. Transmitting a packet is done in split phase. The 'Send' command start sending the radio packet. When transmission is done, a 'sendDone' event will be signaled for notification. When a packet is received, 'receive' event will be signaled. In addition, there is an interface called 'RadioCoordinator' can be used during sending and receiving data packet. This interface provides an excellent mechanism to trace the progress of current transmission. This will be further explained in section 4.6 ranging precision.

For further details of radio message and radio protocol stack, please refer to [37] and [38]. The source code can be found in Appendix G.3.

4.6 Ranging over Ultrasound

The distance measurement using *Time of Arrival* [29] can be realized by ultrasound. The ranging can be demonstrated by following experiment. Two devices are placed about 31cm apart, a transmitter and a receiver. The transmitter sends an US pulse, and detected by the receiver. The signal is measured by oscilloscope and shown in figure 4.7.

For distance of $31cm$ at speed of $340m/s$, it takes ultrasound signal about $0.91ms$ to travel. In the measurement, time difference between sending and reception is found to be $1.04ms$, which is very close to the calculation. So by knowing the sending time and measure the arrival time, the distance between two parties can be determined. However, two separate devices are not synchronised. The receiver does not know when the ultrasound signal was sent. To synchronise the devices requires extra effort such as performing synchronisation protocol.

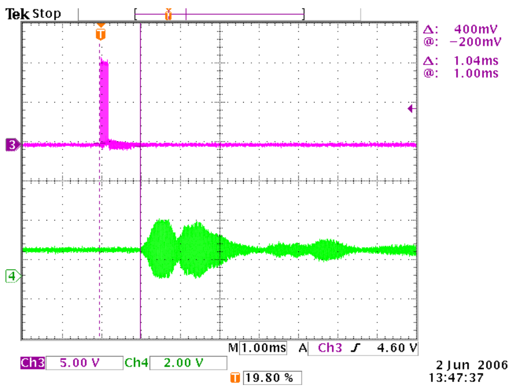


Figure 4.7: Ultrasound ranging

A simple way without using synchronisation protocol is to use radio link together with ultrasound. The ranging starts by sending a message from Alice over radio. Assuming radio travels at the speed of light. Bob will receive the message right away and send a message back to Alice via ultrasound. Alice can use a watch to record the time between sending of her message and reception of the response. Since ideally the radio wave travels at the speed of light, which is neglectable, therefore the time measured corresponds to the distance that ultrasound physically travelled (assuming no cheating take place).

Small programs are developed in nesC to realize ranging as described above. The following paragraphs will explain some implementation issues, the readers that are not interested in the detail may proceed to section 4.7.

'KeySender' and 'KeyReceiver' are developed in nesC. These two programs can be further extended for key exchange. By now, they are implemented with ranging functionality. The 'KeySender' (from here on noted as KS) sends a radio message to 'KeyReceiver' (noted as KR). Upon reception, KR immediately sends an ultrasound message back to KS. The software structure is simple and based on the module previously developed. It uses 'usTransceiver' and 'rfTransceiver' module for US and radio transmission. For debugging purpose, 'Leds' and 'RS232' are also used.

A timer module is used to regulate the pace of program execution. Once the timer is expired, an event is signaled. A radio packet is sent during the event handler. When the radio message is received, KR sends back an ultrasound message. The ultrasound message will be detected by KS and signal a 'DataReceived' event. The ultrasound detector will be enabled and counting time between the sending of radio message and the arrival of the ultrasound.

Ranging precision of more than $10cm$ was experienced. The problem is mainly due to the nondeterministic nature of 'task' execution. As mentioned in section 4.3, tasks are *posted* into a globally managed queue. The task queue is normally serviced with First-in-First-out (FIFO) policy. In addition, the operating system also posts tasks to the same queue. Therefore, the user program does not have any knowledge of task completion. In the radio transceiver software module, event 'DataReceived' is signaled by a task that is posted in a module at lower layer (module is named 'CRCPacket'). Therefore the time KR is informed about the reception of the radio packet is not exactly the time when it reaches KR. Hence, the ultrasound reply is delayed, the precision is lost. The solution to such problem is to use 'RadioCoordinator' interface. This interface will *wire* 'rfTransceiver' with module 'CC1000RadioIntM', which directly communicate with hardware transceiver. An event called 'byte' will be signaled each time one byte of the packet is being sent or received. By examining the source code, and trace down to the lowest level[6], this event is signaled directly from hardware. So KS can know exactly when a certain byte is being sent, and KR can know when it is being received. Since radio wave travels fast enough, so it can be understood that sending and receiving one byte occur at the same time. Both parties are now 'synchronised' at *byte* level. In KS, the ultrasound detector should be enabled during 'byte' event, and KR should correspondingly send ultrasound response back to KS. By such technique, the precision has been improved to less than $10cm$, but this is not the limit. Due to software overhead, (such as context protection for function call) and/or interference from preemptions (such as hardware/software interrupts), so that both transmitter and receiver are synchronised in the same byte but not the same bit. The Cricket system provides additional function to fine tune the synchronisation. By calling function 'GetRxBitOffset', the current bit index of that byte can be obtained. This bit index indicates how much radio receiver lags with transmitter[7]. Since it is a *bit offset*, the value is between 0 and 7. Value 0 indicates that it lags the most and 7 indicates no lag. The KR as radio receiver should obtain such information and send back to KS for correction. This information is carried in the ultrasound packet as the reply. By experiment, such technique has improved the precision to less than $2cm$.

It should be noticed that the ultrasound ranging program proceed in a way that is very similar to *Distance bounding* protocol. This piece of software makes one step further towards the final implementation of the secure protocol, and play a significant role along the development.

[6]No literature has been found describing the construction of 'RadioCoordinator', therefore the source code is examined here. This event is eventually signaled from module 'HPLSpiM'

[7]This technique is concluded by careful study of the Cricket system source code, supplied as open source

The source code for ultrasound ranging can be found in Appendix G.5.

4.7 Integrity and Authenticity

The essential goal of this work is to implement a secure key exchange protocol. Security is based on *Confidentiality, Integrity and Availability* (CIA) [39]. As known from Chapter 2, key establishment protocol exchanges public key information, instead of informing each other the final key. By public key information, the two parties involved in the key establishment will derive the final key through mathematical computation. In such a way, the exchanged key information need no longer to be private and secret. Confidentiality is not a necessity in this work. Since the key is established from public key information, this information must be kept unchanged during execution. *Integrity* is most important. Since key exchange protocol will not be executed very often, therefore availability will not be addressed.

Integrity prevents unauthorized modification. There are several method to detect such modification. As has been mentioned many times and used a lot in the communication society, *Parity check* and *CRC* are very popular. Error correction code also exist. This kind of error detection method is mainly used to detect nonmalicious changes during faulty transmission, introduced by noise or interference from nearby electronics. Malicious modification must be handled so that prevents the attacker from modifying data as well as error detection mechanism [39]. In security society, it is done by cryptographic checksum.

A cryptographic checksum is a function that uses cryptography to produce a checksum, also known as message digest. For error detection code, change made in the detection code can match the modified data. When using cryptographic checksum, such attack can be prevented. One bit modification alters the cipher text significantly, and vice versa. Some cryptographic function hardly have an inverse function[8], e.g. one-way hash function.

Security provided with TinyOS is through the use of TinySec. It is a link layer security architecture especially designed for wireless sensor networks. TinySec uses CBC-MAC (Cipher Block Chaining Message Authentication Code) to address message integrity and authenticity. The MAC implemented in the TinySec uses the same block cipher that is used for encryption, therefore the MAC is very easy to be used. TinySec offers two block ciphers, RC5 and Skipjack. Since RC5 is patented, Skipjack is the default cipher. Also from evaluation result, Skipjack is faster than RC5 in computation time [40].

[8]by hardly, we mean it is computational infeasible to compute the inverse

A small program is implemented with use of TinySec, *MACsender* and *MACreceiver*. The software structure is to wire the program to 'CBCMAC' module, and 'CBCMAC' is wired with 'SkipJack' module. When the program needs to compute MAC value, it essentially executes Skipjack block cipher. 'rf-Transceiver' module is also used for radio communication.

The program execution is: The MACsender computes the MAC on predefined clear text. Both clear text and MAC value are sent to the MACreceiver. Upon reception, MACreceiver recomputes the MAC value based on clear text, and compares with the one received. The received value and computed value are sent to PC through serial cable, hexadecimal value is displayed for visual inspection. Both end are provided with the same MAC key. It should be noticed that computing MAC is still time consuming. It should not be used in event handler. A 'task' should be posted into the global task queue to perform MAC computation. When the task is done, a flag should be set for notification.

The source code mentioned with use of TinySec can be found in Appendix G.6.

4.8 Debugging utility

The debugging capability of Cricket system is very limited. Unlike the common PC programming interface, e.g. Visual C++, MATLAB. There is no visual interface for debugging purpose. There are three LEDs on the Cricket hardware, which can be used to monitor program behavior, however very limited. Communication between Cricket and PC through a serial cable is provided. This gives a better chance for program analysis and trouble shooting.

A 'RS232' module is developed. It uses 'Serial' module, which wires with hardware UART module. To initiate serial communication, 'SetStdoutSerial' is called through 'Serial' interface. Function 'printf' can be use to send a string to PC. Upon receiving a string from PC, 'Receive' event will be signaled.

Debugging nesC code with AVR JTAG ICE is also available. It is an in-circuit debugging interface through JTAG technology[9]. With JTAG ICE, many debugging techniques can be used, such as breakpoint, memory and register content reading, etc. However, it requires additional hardware component to be purchased, e.g. JTAG ICE pod; and hardware modification is needed for certain device. Therefore, this method has not been experienced. Further details can be found in article 'Debugging nesC code with the AVR JTAG ICE' in TinyOS

[9]JTAG is a test interface used in boundary scan. It is widely used for circuit board testing and chip verification.

documentation[42].

Source code for 'RS232' module can be found in Appendix G.7.

4.9 Protocol Implementation

4.9.1 Distance bounding - bit exchange (Revisited)

There are several building blocks have been designed and implemented in the previous sections. This section will describe the implementation of the secure key establishment protocol. Before discussing the details of implementation issues, here is a review of the protocol. The protocol was shown in figure 3.5 on page 37. For convenience, shown again here in figure 4.8.

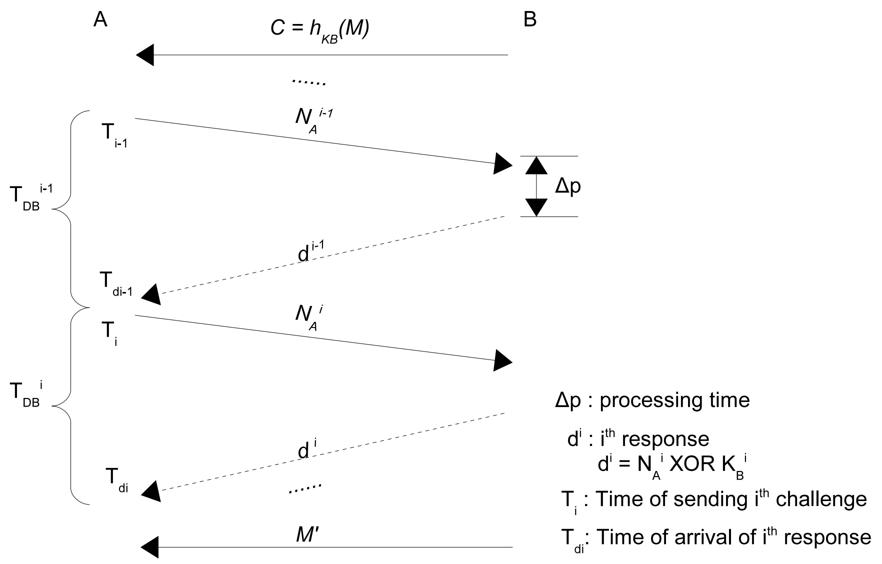


Figure 4.8: Distance bounding - Bit exchange with Expanded details(revisited)

Two parties Alice and Bob wish to establish a session key using DH method. Each should send their public key information to the other. This protocol starts from Bob. Bob picks up a number as public key information and makes up a message M containing the key information. He first commits to the message obtains C and sends to Alice via radio channel. Commitment can be implemented by using cryptographic hash functions, or any other cryptofunctions. As shown in the figure, a keyed hash function $h_{K_B}(\cdot)$ is used. K_B is the hash key. Alice generates an n-bit nonce N, and sends to Bob as a challenge. The challenge is sent in n rounds, one bit at a time. In each round, Bob should receive the challenge, computes the response d^i (by $N_A^i \oplus K_B^i$, where K_B^i is i^{th} bit in hash key.) and sends it back to Alice immediately using ultrasound. When Alice received the reply, she continues with the next round. Alice measures the time between sending of the challenge and arrival of the response, denoted as T_{DB}^i. When all n rounds are finished, distance bounding phase is completed. Bob reveals the message by sending it in clear text.

Alice obtains the hash key K_B from the response d^i and the nonce N^i. She recomputes the C by hash function $h_{K_B}(\cdot)$ over the message received in the last step M. She compares with the one received from the first step. If the two matches, the integrity property of the message M is satisfied; i.e. no one has modified the message. However, the authenticity of the message also need to be proved. This is achieved by distance bounding. Alice verifies the distance between her and Bob, and compares with the number found from distance bounding phase. She looks around, if no one else is within the 'Integrity Region' and the distance measured is true, then the message is authenticated, failed otherwise.

4.9.2 Implementation issues

Software structure

The protocol consists of basically three main phases: Commitment, Distance Bounding and Decommit. The commitment to the message indicates that the sender of the message had decided a key information, and will not change from here on. The MAC can be used as a commitment scheme, which is essentially a cryptographic function. The distance bounding phase consists of rapid bit exchange, in the form of challenge response pair. The challenge is sent in radio channel and response in ultrasound. Since radio travels at the speed of light, so the time depends on distance that sound travels. The decommit phase is to send key information committed in clear text. So the following functions are needed: Ultrasound transmission, Radio transmission, Cryptographic functionality and Ranging. These functions have already been realized by previous section from 4.4 to 4.7. To show the correct execution of the protocol and also ease the debugging, serial communication with PC is used. The following modules are used and wired together in the implementation: rfTransceiverC, usTransceiverC, CBCMAC, SkipJackM, RS232C, TimerC, LedsC.

The use of those modules makes the implementation issues addressed in previous sections also apply here in this implementation. Here is a short summary: The data rate of ultrasound transmission is 60 ms; the communication protocol uses RS232 standard, i.e. one start bit, 8 bits data payload, one parity check bit, and two stop bits; ranging precision is ensured by using 'RadioCoordinator' and replying the bit offset as computation overhead; the MAC computation should be placed in a 'task', since it is a intensive computation.

Program execution

Two programs are developed, 'PtlSender' and 'PtlReceiver'. The former (on behalf of Bob) initiates the key exchange protocol with commitment. The latter starts distance bounding with radio challenge. When the challenge is received, with the use of 'RadioCoordinator' for synchronization, ultrasound response is sent. As discussed in section 4.6, 'RadioCoordinator' ensures the ranging accuracy. The program execution is controlled by a state machine. Each state has an output that is the action to be taken, such as sending the commitment. When the action is performed, the state variable is updated. The action is taken when an event occur. For example, after 'PtlSender' sent the commitment, the state variable should be updated to the next value. Ultrasound response must be sent when a challenge is received. The state machine, unlike conventional FSM usually seen in the digital electronics such as hardware transceiver and processor unit, which is usually driven by a global clock[10]. The state machine developed in this work, progresses under events, it executes in an asynchronous fashion.

System State Machine

The state machines in 'PtlSender' and 'PtlReceiver' are very similar. State machine of 'PtlSender' is shown in table 4.2. The *Entry* indicates the event handler or tasks that perform actions and generates *Output*. It also determines the *Next State*. After the system is started, it resets the system into PHASE_START state. A timer is used to pace the system and brings it into PHASE_IDLE state. In the mean time, task 'compute_cipher' is computing the MAC and change the next state into PHASE1. When the timer is fired again, it will encounter PHASE1 state and send the commitment. During transmission, the system is in PHASE1_WAIT state. When 'sendDone' is signaled, next state is changed to PHASE2. In this state, challenges will be received and the response should be sent. The system keeps in PHASE2 state until enough challenges are received, then change the next state into PHASE3. Finally, the system is in PHASE4, the public key information is sent in clear text.

[10] Certainly asynchronous circuit also exist. Recently, researchers have paid more attention to asynchronous circuit, due to the fact that it brings a better power profile with less dynamic

PtlSender		
Entry	Output	Next State
StdControl.init	Reset System state	PHASE_START
Timer.fired	Computing MAC	PHASE_IDLE
compute_cipher	Compute MAC	PHASE1
Timer.fired	Send commitment	PHASE1_WAIT
rfTransceiver.sendDone	Commitment sent	PHASE2
RadioReceiveCoord.byte	Send response	PHASE2
rfTransceiver.receive	Accumulate challenge	PHASE3
Timer.fired	——	PHASE4
Timer.fired	Send message in clear text	——

Table 4.2: PtlSender state machine

In the 'PtlReceiver', the system is reset into PHASE_START state. When timer is fired, the system is ready to receive commitment and change the next state to PHASE1. When commitment is actually received, the system is ready to be PHASE2. It should send challenges and expect ultrasound response in PHASE3. When all responses are collected, it will set next state to PHASE4 and wait for decommitment. The system in PHASE4 should check the key information with the commitment and conclude the result. The state machine is shown in table 4.3.

PtlReceiver		
Entry	Output	Next State
StdControl.init	Reset System state	PHASE_START
Timer.fired	Wait for commitment	PHASE1
rfTransceiver.receive	——	PHASE2
Timer.fired	Send challenge	PHASE3
usTransceiver.msgDetected	Time US response	PHASE3
usTransceiver.DataReceived	Get US response	PHASE4
rfTransceiver.receive	Authenticate message	——

Table 4.3: PtlReceiver state machine

Ultrasound Ranging

The principle of precise ranging has been described in section 4.6. In this paragraph, the implementation details are explained. The ranging consists of a radio

energy consumption.

challenge sent by Alice, and Bob returns an ultrasound reply. The synchronization in ‘byte’ level is to use ‘RadioCoordinator’. The lag between Alice and Bob’s transceiver is indicated by bit offset, denoted as *comp*. In order to have a good ranging precision, the value *comp* should be informed to Alice by Bob. Alice will use *comp* to compensate the measured time-of-flight. Since it is known that radio transceiver has a bit rate of $20kpbs$, it takes about $48.8us$ to transmit or receive one bit. By multiply *comp* with $48.8us$, the lag time can be found. From Cricket user manual [22], it states that there are timer offset of $550us$ due to software overhead, and denoted as OF. The speed of sound is approximately $340m/s$, or equivalent to $34mm/100us$ in order to avoid numerical errors. The computation needed to obtain the distance between Alice and Bob is:

$$T = TOF - comp * 48 - OF \tag{4.1}$$
$$S = v \cdot T \tag{4.2}$$

where TOF is time-of-flight, *comp* is bit compensate, OF is timer offset, T is the corrected time, v is the speed of sound and S is the distance.
It has been found during test that offset OF of $550us$ is not precise, $900us$ is found to be more accurate.

Distance bounding

Distance bounding technique has a significant role in this protocol. The distance between two communicating parties is bounded by the result determined. To incorporate ranging with distance bounding in this implementation, some modification is required. The challenge sent over radio is only one bit. From original distance bounding protocol, only one bit response is needed. However as discussed in section 4.6, a correction is needed to obtain ranging precision. The correction is a integer number between 0 and 7, as a bit offset from radio transceiver. The ultrasound response packet should accommodate both information. The bit offset can be represented by a 3-bit binary number, together with one response bit, they can be accommodated in 8 bits data payload. The packet format is shown in figure 4.9.

ST| x^i |oft^0|oft^1|oft^2|NA|NA|NA|NA|PA|SP|SP

ST: start bit	NA: Empty bit
x^i : response bit	PA: Parity check bit
oft: offset	SP : stop bit

Figure 4.9: Ultrasound reply packet format

In the original distance bounding protocol, the arrival of the response is notified by the reception of x^i. x^i could be either ‘1’ or ‘0’. In this implementation, logic '0' is represented by absent of ultrasound signal in the channel. Without ultrasound signal, the ranging is impossible. If half of the responses are '0', half of the challenge response pair will have no distance measurement. This significantly weaken the security property of distance bounding protocol. In this implementation, such problem is solved by ultrasound transmission protocol described in section 4.4.4 figure 4.6. The arrival of the ultrasound packet is notified by the start bit. When the start bit is found by the ultrasound receiver, it should stop the watch and record the time. This method uses the advantage of the packet, makes the distance measurement independent of the data payload, and maintained the distance bounding protocol security property.

Commitment

The commitment is another important concept in this protocol. Bob cannot change the message M after she commits to the message and sent C, which is called *binding*. Alice cannot obtain any information from the commitment C about M, until Bob opens it, which is called *concealing* [43] [14]. The use of the commitment is to improve the security of the protocol, the security of the commitment scheme itself is not considered here in this work. The essential goal is to ensure the integrity of the message M. MAC is commonly used for such purpose as been used in section 4.7. One way hash is another example. TinySec provides MAC functionality. As in the previous section, MAC is implemented by a cryptofunction 'Skipjack'. The MAC size determines the security of the protocol. If the MAC is only one bit, then the attacker has $2^{(-1)}$ chance to succeed. The longer MAC size, more security the protocol has. Since this implementation is a demo of the protocol, default size of 64 bits is used. The chance for an attacker to succeed is therefore $2^{(-64)}$. Usually more than 128 bits is preferred.

Concurrency and race condition

It has been noted several times that TinyOS is an event-driven operating system. During normal program execution, software event (such as timer expired) and hardware interrupt (such as data is sampled from A/D converter) can occur. The current program execution can be preempted by other event. Access of the shared variables in one 'thread' can be *raced by* another 'thread'. The issues in concurrent/parallel programs also exist in this case. The solution provided by TinyOS is to use keyword *atomic*. The instruction guarded by *atomic* keyword will be protected from race conditions. Some implementation of the atomic action is through the use of hardware '*Test and Set*' instruction([36] pp.99). In TinyOS, the current implementation of *atomic* is by disabling and enabling interrupts [41]. This implementation has very low overhead. This approach is very effective and efficient. However the downside is also apparent: leaving interrupts disabled for a long period, makes the system less responsive, i.e. samples can be lost from A/D converter, and radio packet cannot be received. Therefore excessive use of *atomic* keyword should be avoided. Atomic statements are not allowed to call commands or signal events [41].

```
switch(state) {
   case s01:    call cmd1;
                state = s02;
                break;
   case s02:    ...
         ...
}
```

(a) race condition

```
atomic tempState = state;
   switch(tempState) {
      case s01:    call cmd1;
                   tempState = s02;
                   break;
      case s02:    ...
            ...
   }
atomic state = tempState;
```

(b) atomic guarded

Figure 4.10: State transition

By now, access shared variable can be protected with *atomic* keyword, however, there is a particular case that should be mentioned, *state variable*. The program execution is guided by a state machine. The state variable is also a shared variable and should be protected. The protection from race condition should be taken place during state transition. A very common implementation of state transition is to use *if* branch or *switch case* statement. An example is shown in figure 4.10, where (a) shows a piece of code that may suffer from race condition, there is no concurrency protection. The safest implementation is to guard the entire code segment by atomic keyword. Since it may contain command calls or event signaling, this method should be avoided. The recommended solution is shown in figure 4.10(b). The current state value is stored into a temporary variable as an atomic statement. the temporary variable should be a local variable that cannot be accessed by any other event handler. Any changes to the state value is made on this temporary variable. When the required actions are taken, the state value is then saved back into the global 'state' variable.

Reliability

This implementation of the protocol is mainly focused on the security aspect. However, since it is also a distributed system, it shares the problem that may occur in every distributed system. For example, a packet might be lost, data can be corrupted. The error occurred in data can be detected by error detection code, in this implementation, parity check is used in ultrasound communication and CRC in radio protocol stack. The cryptographic function used in MAC also provides such detection capability. The packet loss however, will create an impact to the program execution. The protocol progresses according to the state machine. The state value is updated after an event occur. The event is triggered by an arrival of a message. If a message is lost, the corresponding event won't occur and the state machine will not function correctly. This will essentially lead to a *deadlock* situation. Device A will not send message M_A^2 until device B sends a message M_B^1. Device B will not send message M_B^3 until M_A^2 is received. But if M_B^1 is lost at the first place, the sending of the following two messages from both A and B will not happen. Both devices cannot progress.

One solution to such problem can be found from TCP protocol, which provides a reliable connection. An acknowledgment 'ACK' is used as a response to inform a packet has been received. If 'ACK' is not replied within certain time interval, the packet is found to be lost and retransmission is required. In some cases, negative acknowledgment 'NACK' can also be found. This approach will add further complexity to the implementation, and the security property need to be re-evaluated. This approach is not used in this work. It can be left for further study.

The *second solution* is to use a 'Watch dog'. 'Watch dog' can be seen in many embedded systems. When the system is not responsive for a certain period of time, the 'Watch dog' will reset the system to a known state. Usually 'Watch dog' is a specific hardware module in the microcontroller. This solution is very simple. To be used in this implementation, only a software timer is needed. When a message is received, the event handler will reset the timer. If the message is not received due to packet loss, the watch dog timer will be fired after pre-defined interval. It will reset the system state machine, and restart the entire protocol. For current implementation, the sender of the key has a watch dog timer interval of 7 second, the receiver is set to 5 second. The reason for such inequality is that the sender initiates the protocol, the receiver has to be ready before that point of time.

The source code of 'PtlSender' and 'PtlReceiver' is shown in Appendix G.8.

4.10 Conclusion

The key establishment protocol has been implemented and described in previous sections. In this section, the protocol is reviewed, implementation details are summarized. The final protocol is shown in figure 4.11 on page 96.

Two parties A and B establish key by exchanging public key information. The protocol sends such information only from B to A. Before protocol starts, B should generate public key information and contained in message M. Both A and B should generate random nonce N_A and K_B respectively. The length of nonce is denoted as L, which is usually 64 bits or longer. The MAC is computed over M with nonce K_B, denoted as C and transmitted to A over radio link. The MAC is computed beforehand as a 'task'. In order to obtain B's nonce K_B to authenticate M, rapid bit exchange is invoked. Party A sends one bit of nonce N_A via radio link as challenge. Upon reception, B computes $N_A^i \oplus K_B^i$ and denoted as x^i. B should also obtain radio bit offset denoted as oft to keep synchronization as described in section 4.6. x^i is concatenated with oft denoted as d^i, and sent over ultrasound channel. The ultrasound packet format is also shown in figure 4.11. A receives the ultrasound response, and records the time between sending of challenge and reception of the response. This procedure repeats for L times, and forms the distance bounding. After this phase, the distance between A and B is determined. B will send M in clear text to A for integrity check. A will determine the nonce K_B from x. He computes $MAC_{K_B}(M')$ and compare with C. Key is established if the two matches, abort otherwise. A should also check if the distance between A and B matches with the one determined from distance bounding. If they don't match, the key should be discarded. By reversing the role, A's key information can be sent to B.

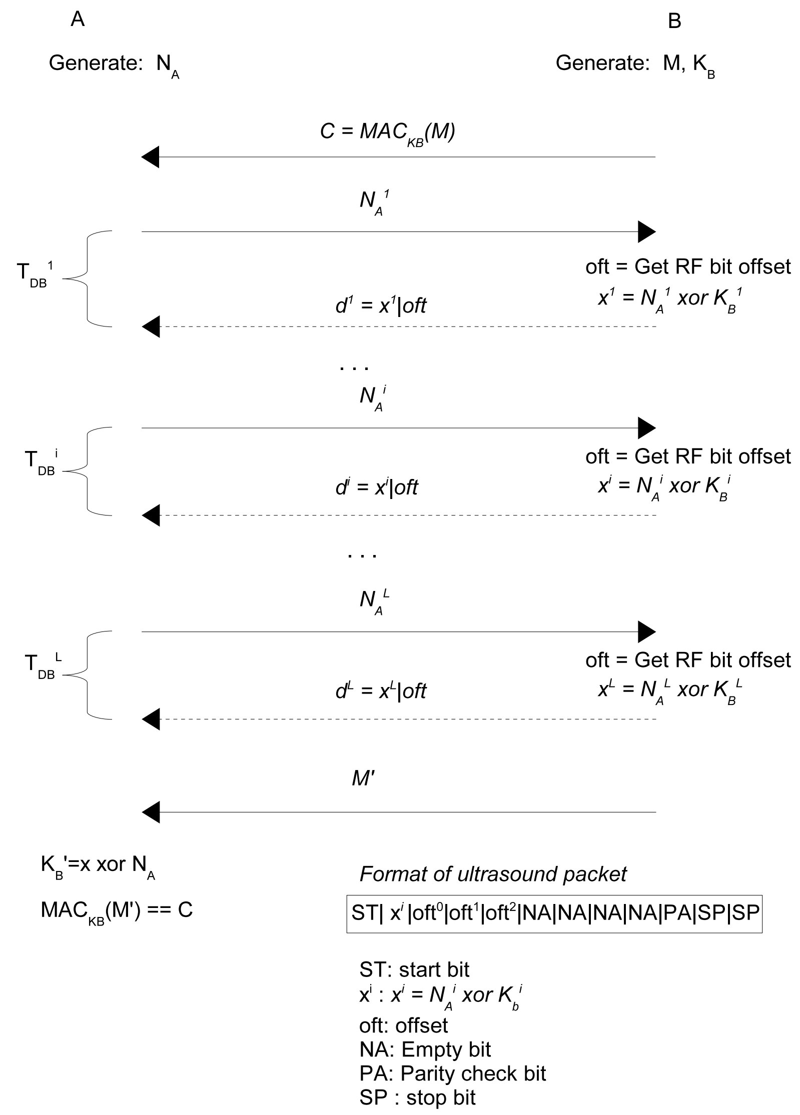


Figure 4.11: Protocol summary

Chapter 5

Tests

This chapter describes how each building blocks and the implementation of the secure key establishment protocol are tested. The test strategies are described, and test result is summarized.

5.1 Ultrasound transmission

The first building block to be tested is ultrasound transmission, ‘usTransceiver’ module. The test uses ‘usSender’ to send data to ‘usReceiver’ via ultrasound. The test is mainly focused on functional test, i.e. correct behavior in normal condition. The test evaluates whether transmission can be established, the correctness, and the coverage.

Test of transmission establishment

This is a rather simple functional test. The test is performed by sending regular data to the receiver. The test pattern is ‘0x30,0x31,0x32,0x33’ and circulating. Two devices are placed on desk with about 30cm apart. The test result shows the same data patterns have been received. The result is captured by ‘Hyper terminal’ and shown in Appendix E.1. Notice that the MSB (Most Significant Bit) is the parity bit.

Test of correctness

The correctness of the transmission is tested. The purpose is, 1st to evaluate that the correct transmission is independent of both the time and data. 2nd if noise is presented, error should be detected by error detection mechanism within its capability. To evaluate the first goal, the devices should be tested for a long time interval. The exhaustive test pattern should be applied. In digital circuit tests, pseudo random test patterns are often used. It can reduce the total number of test patterns applied to the system [47]. Both are used to test that the system does not suffer from certain data pattern. Exhaustive test is to use an Automatic Test Pattern Generator (ATPG) traverse all possible data in the Device Under Test (DUT). The pseudo random test pattern generator is usually implemented by a Linear Feedback Shift Registers (LFSRs). To evaluate the second goal, sound noise should be generated.

The 'usTransceiver' module is designed to carry 15 bits data payload. The bit width is $60ms$. To send a data packet with payload of 15 bits, including 4 bits overhead (start bits, parity bit and stop bits), in total 19 bits are sent, which takes at least $1.14s$ ($19 \times 0.06s = 1.14s$). 15 bits data will result of 32768 patterns. To exhaustively test all patterns requires 37355.52 second, which is about 10 hours. Such test is not realistic. Pseudo random test will reduce total number of test patterns, and the test result is a probability that the system functions correctly. However, an additional channel is needed to agree the current test pattern. Therefore, in this work, a so called *partial exhaustive test* is conducted. Since 8 bits data are commonly used, the test only evaluate the correctness of communication with 8 bits data. This method reduces the number of test patterns to 256. Each transmission only involves 12 bits, which is 0.72 second. To complete the test, about 3 minutes are required. This test interval is also enough to demonstrate time independency.

The test result found out that the parity bit has a 'stuck-at-0' fault[1]. It means the parity bit is always logic '0' independent of data pattern. This fault is due to the overflow of the data buffer in the transmitter. The fault is fixed, and the transceiver is working correctly. Test result is shown in AppendixE.1.

To evaluate second goal, sound noises are generated in the surroundings. The test location is picked as a nature environment with activities, i.e. a lab in DTU building 322 room 229. At the time test is conducted, there are about 25 students in the lab doing their exercises. The exercises are multi-people group work, hence increase the background noise. Intensive noises are also introduced, such as hitting the desk and drop a metal at the place very close to the DUT[2].

[1] A 'stuck at' fault is a most common fault model used in digital circuit. It indicates a node in the circuit being fixed at logic 0 or logic 1.

[2]by close to, we mean less than 5 cm

The test is conducted for 2 minutes. The first 1 minute is tested only with background noise, and second minute is tested with intensive noise. The test result shows that the DUT can work correctly under background noise. With intensive noise, several errors has be detected. However, it also shows that there are errors not detected, e.g. data pattern '0x1f3' received was neither generated, nor detected as error. The reason for such undetected fault is due to the inefficiency of the parity check mechanism. The pattern supports to be '0x133', which is equivalent to '100110011B'. The pattern received is '0x1f3', which is '111110011B'. The faulty pattern has two bit errors. The parity check is not capable of detecting such error. To improve the error detection rate, a better coding scheme might by used. The test result is shown in AppendixE.1.

Coverage

The coverage of the ultrasound transceiver is roughly evaluated. This test is just to give an idea of maximum communication distance between the transmitter and the receiver. This test is not meant to be precise. The coverage is determined at the distance when receiver fails to correctly receive data from the transmitter. The coverage is found to be about 3 meters. (The precise method is to calculate sound pressure at the receiver's position. It is known that distance doubles, the sound pressure reduces by half.)

5.2 Radio transmission

The radio transceiver module 'rfTransceiver' is tested by 'rfSender' and 'rfReceiver' module. Since 'rfTransceiver' used modules that are defined in TinyOS, intensive tests are not required. Only functional test is conducted. The test patterns are binary number sequence from 0x0 to 0x7. The patterns received are displayed on LEDs. The test result shows that the radio transmission functioning correctly.

5.3 Ultrasound ranging

An ranging application using ultrasound has been developed, and described in previous chapter. Two devices are used for ranging test, one is programmed with 'KeySender' and the other with 'KeyReceiver'. Two major aspects are concerned in the test: the correct functioning of the ranging system and the accuracy.

Functional test

Two devices are place 30 cm apart. The test lasts for one minute, the distance measured is very accurate. By input test result into MATLAB, accuracy is calculated by mean value and standard deviation. The mean is $\mu = 29.677$, and standard deviation is $\rho = 0.599$.

Accuracy test

The accuracy is evaluated by testing at different distances, record the result and tabulate the mean and standard deviation. The table is shown below:

Test condition (cm) (d)	Mean (μ)	Std Dev. (ρ)	Bias $\lvert d-\mu \rvert$
30	29.677	0.599	0.323
60	58.233	0.568	1.767
100	99.846	0.464	0.154
200	196.5	0.5099	3.5

Table 5.1: Ranging accuracy

The mean value is the average of the test result. Standard deviation indicates how much the test result differ from the mean. It can be seen that the test readings are rather stable. The bias indicates how much the measured value differs from the truth. At 200 cm distance, the measurement has the maximum error.

The test data can be found in Appendix E.2.

5.4 Commitment

This section is to test commitment scheme. The MAC is used for commitment. It is implemented by invoking Skipjack ciphers. Two programs 'MACSender' and 'MACReceiver' are developed to test the commitment. The former sends a radio packet containing 29 bytes of data. 4 bytes MAC code are computed over the data. The MAC is sent again over radio link. The latter receives two packets and computes the MAC of data packet. The clear text data, MAC received and the MAC computed are sent over serial cable to PC, displayed on HyperTerminal.

Two tests have been conducted, a functional test and a test under attack.

Functional test

The functional test is to test the correct behavior of the system. The 'MACReceiver' should compute the MAC of received data. The result of computed MAC should match the one received. The test data (test pattern) are 29 bytes long. To run exhaustive test over 29 bytes data is impossible. The test patterns are arranged such that there are correlation between each byte in one test, and correlation between the current test pattern and the next. In total 20 patterns are tested. The test result shows that without interference (and no error situation) the MAC computed matches with the MAC received. There is no obvious correlation between each byte in the MAC, nor correlation between the current MAC computed and the next MAC. The test result confirms with the expectation. The result is shown in Appendix E.3.

Attack situation

The situation where attacker is involved is also tested. A third device is programmed with 'rfSender' developed in previous section. The purpose of this device is to send arbitrary data over radio link so that collision will alter the packet content in the radio channel. Such modification should result of significant change between MAC received and the one computed. The same test patterns are used in this test. Some test results are selected and listed in table 5.2. The full listing is shown in Appendix E.3.

Test	Correct MAC	Received MAC	Computed MAC	Authenticity	Modification
1	2d 9 96 86	2d 9 96 86	2d 9 96 86	Yes	No
2	30 99 4a 79	31 0 0 0	30 99 4a 79	No	MAC
3	8a bf 23 11	33 0 0 0	66 ae c7 8d	No	Text + MAC
4	9c 5e ba 80	32 33 34 35	39 59 aa 2d	No	Text + MAC
5	47 52 42 9c	31 0 0 0	3c 3e d9 41	No	Text + MAC

Table 5.2: Test of MAC

The correct MAC values are listed. Only the data from first test was authenticated, the others have been modified. The second test, the computed MAC equals to the correct MAC, but the received MAC is different. The MAC has been changed. For other three shown in the table, neither MAC received nor computed matches with the correct MAC, modification is found in both clear text and MAC. The MAC has successfully detected unauthorized changes.

5.5 Key establishment protocol

The key establishment protocol has been implemented and described in section 4.9.2. Two Cricket devices are required, one is programmed with ‘PtlSender’ and the other is programmed with ‘PtlReceiver’. The former initiates the protocol and sends his key information to the latter. The protocol involves commitment and distance bounding. Two tests have been applied, functional test and intrusion test. Since the key exchange protocol is implemented by small modules that have been tested in previous sections, therefore, in this section, block tests will not be conducted.

Functional test

The functional test is to show the system can behave correctly. Two devices are placed about 60cm apart. The messages received by ‘PtlReceiver’ are sent through serial cable to HyperTerminal. The test result is shown in Appendix E.4. It first shows the commitment, followed by distance bounding. The information supplied during distance bounding has the following format:

bt:2559us, 0 @1, ofst:4, t:1803us, D:61cm

The meaning for each abbreviation is shown in table 5.3. The above line says: The response data is ‘0’, at bit index 1 (0 @1); the uncorrected time is found to be 2559us(bt:2559us), with bit offset 4(ofst:4); the corrected time is calculated as 1803us(t:1803us), and hence distance is 61cm(D:61cm).

bt:nnnn us,	n	@n,	ofst:n,	t:nnnn us,	D:nn cm
uncorrected time	bit value	bit index	bit offset	corrected time	distance

Table 5.3: Information format for Distance bounding

The data in clear text is also displayed. The ‘PtlReceiver’ derives the MAC key of ‘PtlSender’, computes the MAC over the clear text and compares with the MAC received.

The test result shows that the implementation is functioning correctly.

Distance shortening attack

Intrusion test has been performed. The test is focused on distance shortening attack. A dishonest key sender has been developed to perform such attack. The attacker is a modified version of the 'PtlSender'. There are two places to be modified, i.e. the byte level synchronization and bit level offset correction. In this test, the attacker is synchronized two byte ahead of time, this will give the attacker about $768us$ to save ($2 \cdot 8 \cdot 48us = 768us$), which correspond to $261mm$ shortened. For bit level offset correction, the minimum bit level offset is always replied to the key receiver, which is 7 (indicates no lag). The test condition is exactly the same as functional test: two devices placed $60cm$ apart. The test result is recorded by HyperTerminal, and shown in Appendix E.4. It shows that the distance has been shortened with about $20cm$. Using MATLAB, the mean value is $42.78cm$, the standard deviation is $4.045cm$, maximum distance measured is $49cm$ and minimum distance measured is $37cm$. The standard deviation is much larger than the test conducted in section 5.3 Ultrasound ranging.

Even though the distance has been shortened, it does not bring any advantage to the attacker. At the end of the protocol, key receiver performs message authentication by MAC matching. The result shows that two MAC don't match with each other. The authentication found that the key sender is not trusted, the key establishment failed! In order to show that this single test was not a coincidence, this test has been performed 3 more times. The test results all confirm with the first one, distance is shortened but key establishment failed due to authentication. The extra 3 tests are also shown in the Appendix. The statistical data of the distance bounding result over all four tests are shown in table 5.4.

Test condition (cm)	Mean	Std Dev.	Shortened	Max.	Min.
(d)	(μ)	(ρ)	$\lvert d-\mu \rvert$		
60	42.78	4.045	17.22	49	37
60	41.67	3.40	18.33	49	37
60	42.95	4.138	17.55	49	36
60	43.40	3.70	16.6	49	37

Table 5.4: Statistical data of distance bounding under attack

It should be emphasized again that when the attacker cheats on the distance, he has to send out the response before he could know the challenge. The response to be correct is purely by chance. The attacker has only $2^{(-n)}$ probability to be completely correct. In this test, the MAC size is 8 bytes that is $n = 64$ bits. The chance for attacker to succeed is extremely small.

By the above four tests under distance shortening attack, this implementation of the protocol is concluded to be correct and secure.

5.6 Conclusion

The implementation of secure key establishment protocol in chapter 4 is based on several building blocks developed. Each building block has been tested in this chapter for functionality and correctness. For ranging and commitment building blocks, the precision and security are also tested specifically. The key establishment protocol is evaluated with functional test. A modification has been made to perform an attack. The implementation correctly passed the functional test and successfully rejected the message from dishonest attacker.

Chapter 6

Conclusion

The technology is developing faster and faster, the expanding of the Internet makes people connected. There have been many forms of connections in cyberspace, e.g. wired connection, wireless network, structured network, Ad-hoc network, etc. The life will be rather different if without any form of communication. Certainly the security will be a great concern in such beneficial technology. The security measures to provide Confidentiality and Integrity has been taken into account in the design of such technology. Cryptographic is a magic that plays a significant role in security. Messages are encrypted with a key and check sum can be computed. However, the design failure and implementation fault makes the protection easy to be broken.

To protect the information exchanged between two participants is to use encryption. The encryption key is required to be established. The key established will be used for security, but what can secure the key establishment. During exchanging of the key, anyone can eavesdrop the communication since it is an open network (no secrecy). The information transmitted in the network can be modified without being detected (no integrity). The attacker can even replace the entire message of a packet false claim to be someone else (no authenticity). The three problems above are the major issues in key establishment. Especially in the ad-hoc network, there is no infrastructure available in the network in any form.

Several proposals have been studied in this work. The proposals all focus on the above mentioned three issues, and provide reasonable solutions. Some proposal uses public key cryptography. The users only exchange the public key. Since such a key is intended to be public, the secrecy is no more a concern. Similarly, public key exponent can be exchanged (DH key exchange), which has the same property. The attacker can eavesdrop the information exchanged, but he cannot derive the session key. The concept about Integrity and Authenticity is different, but somehow related. In public key cryptography, the session key is encrypted by sender's private key. The receiver can authenticate the message by decryption with sender's public key. Since the message is also encrypted by receiver's public key, no one can modify the message so to ensure the Integrity. Without public key cryptosystem, other proposals use other ways to protect the integrity and authenticity. To ensure integrity, message digest or cryptographic hash is used. Upon reception of the message, the receiver computes the digest of the message and compares with the digest received. If the two matches, the message is free from modification. The authenticity is the last issue to be address, but not the least. Many attacks actually succeed from the failure of the authenticity check. Some proposal converts the digest into image and displayed on the screen. The other participant captures the image to authenticate the message sender. This is called an image authenticator. Some proposal uses audio. Others use string authenticator. A solution uses the distance (physical truth) to authenticate the sender of the message. It is known as distance bounding authenticator.

The distance bounding authenticator uses the distance bounding protocol. The distance bounding protocol discovers the physical distance between the message sender and receiver. The user should visually inspect the distance between them. If no third party is within the bound, the information exchanged is considered to be secure. However, the implementation of the distance bounding authenticator can be flawed. The attacker can perform distance shortening attack. The Distance Bounding - Bit exchange (DB-BE) maximizes the security insurance. The challenge-response pair is proceeded in a sequential manner, which makes the attacker very difficult to succeed. The DH-DB protocol puts the key exchange into a symmetric form, the key information is exchanged and key established in one pass of the protocol. It shares the same property with DB-BE. There are also several implementations of the distance bounding protocol used for many different purposes, still closely related with localization.

The key establishment protocol using distance bounding with bit exchange is implemented. The implementation targets on small embedded device with limited resources. Cricket system is used. It runs TinyOS, an embedded operating system. Since it has limited resources, DH method is suitable for such scenario. The DB-BE has been implemented. Several functional blocks have been developed. They are responsible for ultrasound communication, radio communication, ultrasound ranging, cryptography and debugging. The detailed implementation issues have been discussed. The key establishment protocol is implemented based on those building blocks. One device initiates the protocol by sending commitment of the public key information. It is followed by distance bounding. The receiver of the message reveals the information and authenticates with the distance bounding authenticator. Since the public key information is intended to be public, confidentiality is not an issue. The commitment provides hiding and binding property, which ensures the integrity.

The implementation of the protocol has been tested. The building blocks are tested before the final system test. Functionality and correctness is tested. Some block is even tested with attack scenario. The test result shows the building blocks are all working properly. There is no security flaw found. The implementation of the protocol is tested of functionality. The implementation works well. Distance shortening attack is also performed to test the security of the protocol. This implementation has successfully detected and rejected the attacker.

Bibliography

[1] N.Borisov, I.Goldberg, D.Wagner, Intercepting mobile communications: The insecurity of 802.11.
Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM 2001

[2] M.Jakobsson, S.Wetzel, Security weaknesses in Bluetooth.
The Cryptographers' Track at RSA Conference 2001 2001

[3] Whitfield Diffie, Martin E. Hellman, New Directions in Cryptography.
IEEE Transcations on Information Theory Nov 1976

[4] D.Balfanz, D.K.Smetters, P.Stewart, H.C.Wong, Talking to strangers: authentication in ad-hoc wireless networks.
Ninth Annual Symposium on Network and Distributed System Security 2002

[5] N. Asokan and Philip Ginzboorg, Key-Agreement in Ad-hoc Networks.
Computer Communications,volume 23 2000

[6] J.M.McCune, A.Perrig, M.K. Reiter, Seeing-is-believing: using camera phones for human-verifiable authentication.
Electronics and the Environment, 2005 IEEE Symposium on Security and Privacy, 2005

[7] P.Juola, Whole-word phonetic distances and the PGPfone alphabet.
Proceeding of Fourth International Conference on Spoken Language Processing, IEEE 1996

[8] Michael T. Goodrich, Michael Sirivianos, John Solis, Gene Tsudik, and Ersin Uzun Loud and Clear: Human-Verifiable Authentication Based on

Audio.
citeseer.ist.psu.edu/743772.html, Department of Computer Science, University of California, Irvine

[9] D.Maher, Secure communication method and apparatus.
U.S. Patent 5,450,493, Dec 29, 1993

[10] Christian Gehrmann, Chris J. Mitchell, and Kaisa Nyberg, Manual authentication for wireless devices.
RSA Cryptobytes, 7(1):29¨C37, January 2004.

[11] Serge Vaudenay, Sylvain Pasini, An Optimal Non-interactive Message Authentication Protocol.
Lecture Notes in Computer Science, Vol.3860, Springer-Verlag 2005

[12] C.Castelluccia, P.Mutaf, Shake them up! a movement-based pairing protocol for CPU-constrained devices.
Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (MobiSys 2005) USENIX Association 2005

[13] Mario Cagalj, Jean-Pierre Hubaux. Key agreement over a radio link. *http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200416.pdf*

[14] Mario Cagalj, Capkun and Jean-Pierre Hubaux. Key agreement in peer-to-peer wireless networks. *Proceedings of the IEEE (Special Issue on Cryptography and Security)*, 2006. Preliminary version.

[15] Stefan Brands, David Chaum. Distance-Bounding Protocols. *Workshop on the Theory and Application of Cryptographic Techniques Proceedings*, Advances in Cryptology - EUROCRYPT '93.

[16] Naveen Sastry, Umesh Shankar and David Wagner. Secure Verification of Location Claims. *Proceedings of the Workshop on Wireless Security*, Proceedings of the 2003 ACM Workshop on Wireless Security.

[17] G.P. Hancke, M.G. Kuhn. An RFID Distance Bounding Protocol. *Security and Privacy for Emerging Areas in Communications Networks*, SecureComm 2005.

[18] Srdjan Capkun, Levente Buttyan, Jean-Pierre Hubaux. SECTOR: Secure Tracking of Node Encounters in Multi-hop Wireless Networks. *ACM Workshop on Security of Ad Hoc and Sensor Networks*, SASN03.

[19] Yih-Chun Hu, Adrian Perrig, David B. Johnson. Packet Leashes: A Defense against Wormhole Attacks in Wireless Ad Hoc Networks. *Technical Report TR01-384, December 17, 2001 Revised: September 25, 2002,*

[20] N.B.Priyantha, A.Chakraborty, H.Balakrishnan. The Cricket location-support system. *MobiCom 2000. Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, ACM; IEEE Commun. Soc.; USENIX Assoc

[21] Nissanka Bodhi Priyantha. The Cricket Indoor Location System, PhD Thesis *Computer Science and Engineering at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY June 2005*, http://nms.csail.mit.edu/papers/index.php?detail=132

[22] Cricket v2 User Manual, Cricket Project. *MIT Computer Science and Artificial Intelligence Lab Cambridge, MA 02139, January 2005*, http://cricket.csail.mit.edu/

[23] Cricket system datasheet, Cricket wireless location system datasheet, *Crossbow*, http://www.xbow.com/products/productsdetails.aspx?sid=116

[24] ATmega128 datasheet, ATmega128, ATmega128L 8-bit Microcontroller datasheet,
Atmel

[25] MAX864 datasheet, MAX864 Dual-Output Charge Pump with Shutdown *MAXIM-IC* http://pdfserv.maxim-ic.com/en/ds/MAX864.pdf

[26] 255-400ST12 and 255-400SR12 datasheet Ultrasound transmitter 255-400ST12 and ultrasound receiver 255-400ST12 datasheet
Mouser Electronics - Electronic Component Distributor http://www.mouser.com/

[27] cc1000 datasheet Chipcon CC1000 Single Chip Very Low Power RF Transceiver
Chipcon http://www.chipcon.com/

[28] Ultra high frequency
Wikipedia, The Free Encyclopedia
http://en.wikipedia.org/w/index.php?title=Ultra_high_frequency
&oldid=54518574

[29] C.Savarese, J.M.Rabaey, J.Beutel, Locationing in Distributed Ad-Hoc Wireless Sensor Networks, *Acoustics, Speech, and Signal Processing, 2001. Proceedings.*
2001 IEEE

[30] Hari Balakrishnan, Roshan Baliga, Dorothy Curtis, Michel Goraczko, Allen Miu, Bodhi Priyantha, Adam Smith, Ken Steele, Seth Teller, Kevin Wang, Lessons from Developing and Deploying the Cricket Indoor Location System, *MIT Computer Science and Artificial Intelligence Laboratory*
Nov 7, 2003

[31] Robin Sharp, Principles of protocol design, 2ed edition, *Technical University of Denmark, Informatics and Mathematical Modelling*
ISBN 0131821555, Lyngby. 2002.

[32] Daniel L. Metzger, Microcomputer and Electronics, A Practical Approach to Hardware, Software, Troubleshooting, and Interfacing, *Prentice Hall*
ISBN 013579871X, 1989.

[33] David A. Patterson, John L. Hennessy, Computer Organization and Design Second Edition : The Hardware/Software Interface, 2nd edition *Morgan Kaufmann*
ISBN 1558604286, August 1, 1997.

[34] Victor P. Nelson, H. Troy Nagle, Bill D. Carroll, David Irwin, Digital Logic Circuit Analysis and Design, 1st edition *Prentice Hall*
ISBN 0134638948, March 8, 1995.

[35] Kim G. Larsen, Alexandre David, Gerd Behrmann, A Tutorial on Uppaal *Lecture Notes in Computer Science Vol.3185*
Springer-Verlag

[36] Gregory R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, 1st edition *Addison Wesley*
ISBN 0201357526, October 22, 1999

[37] TinyOS Tutorial, *http://www.tinyos.net/tinyos-1.x/doc/tutorial/*
23 September 2003

[38] Nelson Lee, Philip Levis, Jason Hill, Mica High Speed Radio Stack
September 11, 2002

[39] Charles P. Pfleeger, Shari Lawrence Pfleeger, Security in Computing, Third Edition *Prentice Hall*
ISBN 0130355488, December 2, 2002

[40] Chris Karlof, Naveen Sastry, David Wagner, TinySec: A link layer security architecture for wireless sensor networks *SenSys'04 - Proceedings of the Second International Conference on Embedded Networked Sensor Systems*
2004

[41] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, David Culler, The nesC language: A holistic approach to networked embedded systems *Proceedings of the ACM Sigplan 2003 Conference on Programming Language Design and Implementation*
ACM 2003

[42] TinyOS Documentation, Debugging nesC code with the AVR JTAG ICE, *http://www.tinyos.net/tinyos-1.x/doc/nesc/debugging.html*
23 September 2003

[43] Gilles Brassard, Claude Crepeau, Dominic Mayers, Louis Salvail, The Security of Quantum Bit Commitment Schemes, *citeseer.ist.psu.edu/222264.html*

[44] D E Culler, J Hill, P Buonadonna, R Szewczyk, A Woo, A Network-Centric Approach to Embedded Software for Tiny Devices, *Lecture Notes in Computer Science Vol.2211*
ISSN 03029743, 2001

[45] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer and David Culler, TinyOS: An Operating System for Sensor Networks, TinyOS team

[46] Ruben Pinilla, Nacho Navarro, Marisa Gil, TinyOS: A Case of Study for Adaptable Embedded Applications Based on Sensor Networks, *Technical report*
UPC-DAC-2004-6, February, 2004

[47] Stanley L. Hurst, VLSI Testing: Digital and Mixed Analogue/Digital Techniques, *Institution of Electrical Engineers*
ISBN 0852969015, February 1999

[48] DATA ENCRYPTION STANDARD (DES).
http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf
U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology
FIPS PUB 46-3, FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION
1999 October 25

[49] ADVANCED ENCRYPTION STANDARD (AES).
http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf
Federal Information Processing Standards Publication 197
November 26, 2001

[50] R.L. Rivest The RC5 Encryption Algorithm.
In the Proceedings of the Second International Workshop on Fast Software Encryption (FSE)
1994

[51] Skipjack and KEA Algorithm Specifications.
http://csrc.nist.gov/CryptoToolkit/skipjack/skipjack.pdf
May 1998

[52] R. Rivest, A. Shamir, L. Adleman A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM, Vol. 21 (2), pp.120¨C126*
1978

[53] MD5, rfc1321.

[54] SHA-1, rfc3174.

Appendix A

Basics about Security

A.1 Encryption

The three important aspects of computer security are *Confidentiality, Integrity* and *Availiablity* [39]. The confidentiality ensures that the information is not accessed by any unauthorized parties. In security society, confidentiality is ensured by encryptions. A very simple encryption, for example for a text sentence, is to scramble the sequence. The reader has to know the correct sequence, as a key, to unscramble the text. The algorithm that encrypts the clear text into cipher text is the encryption algorithm. The encryption algorithms are all public and well known. For example, the information about the well known encryption algorithm 'Data Encryption Standard (DES)' can be found in [48] and can be freely downloaded from Internet. Therefore, the secret is protected by technology (more precisely by mathematics) not by another secret.

To encrypt a message M, a secret key K_E is needed. The resulting cipher text C is:

$$C = Enc(M, K_E)$$

To reveal the message M from C is called decryption, is a reverse function of the encryption. Decryption requires the party possess of decryption key K_D:

$$M = Enc(C, K_D)$$

If the key K'_D possessed by a party that is different from K_D, the resulting message M' should be significantly different from M.

There are two kinds of encryption algorithms, symmetric and asymmetric. Symmetric algorithm has the same key used for both encryption and decryption, whereas in asymmetric algorithm K_E is different from K_D. The well known symmetric algorithms are: DES, Advanced Encryption Standard (AES) (can be obtained from [49]), RC5 [50] and Skipjack [51]. The example of asymmetric key algorithm (also known as *public key* cryptography) is RSA [52].

The two basic techniques for encryption are *Diffusion* and *Confusion*. The more systematic study about encryption can be found in [39].

A.2 Integrity

Even though the secrecy provided by encryption makes the intruder to be confused enough and have no way to figure out what the secret is, that does not always satisfy everyone's need. Encryption does not prevent the intruder from modifying the information. The mechanism to detect such unauthorized modification, so to ensure information integrity should be provided.

The fundamental idea to detect message modification is by message digest or sometimes called a digital fingerprint. It is for a message M, computes a unique output via a function $h()$. Error detection code found in communication society is an example of such digital figureprint. *Parity check* code is the most basic form of error detection code. It only checks the parity of the information during transmission. Most severe problem of parity code is that it only detects one bit error. *CRC* (Cyclic redundancy check) is a much improved error detection code. However, those error detection code is mainly focused on error introduced by noise and Electromagnetic Interference(EMI).

The cryptographic hash function is a hash function with additional security properties. A good cryptographic hash funciton is a one-way function, that is to compute a hash of the message is easy, whereas to compute the original message from hash value is impossible. This is also known as *preimage resistant*. For example, the hash of a message 'I am at school' might be '0xABCDE'. To obtain hash value '0xABCDE' from plain text message is a simple computation. However, the hash value does not reveal any information about plain text message. Another property of cryptographic hash is *second preimage resistant*. For a given message M, to find another message M' that is different from M but hash(M)=hash(M) is very hard. For example, the message 'I'm at school' is similar to the first example, but the hash might be '0x12345', which is quite different[1]. By using exhaustive search to go through all combinations hence hash values to match with the one under attack be computational infeasible. The most commonly known cryptographic hash functions are MD5 (MD for message digest) and SHA (Secure Hash Algorithm). Further information about MD5 can be found in rfc1321 [53], and SHA-1 can be found in rfc3174 [54]. Unfortunately security flaws have been found in both algorithms.

A concept that is very similar to hash function is MAC (Message Authentication Code). Unlike hash functions, MAC function take two inputs to produce an output. It has the form $MAC(k, M)$, where M is the message, and k is the key to compute the message authentication code. The MAC is used to authenticate the message. It protects both a message's integrity as well as its authenticity. Sometime, MAC is called a keyed hash function. Hash function can also be used to implement MAC. Consider the following scenario, two party Alice and Bob communicate with each other via a wireless link. Alice wishes to send a message containing the most recent data sampled for scientific research. The reliability of the information is very important, so that no false data from adversary should be reported. To authenticate the message, the MAC of each message is computed, tagged at the end of the message and sent over the radio link. In order to check the authenticity at the receiver side, the receiver Bob must also hold Alice's MAC key beforehand. Upon reception of the message, Bob recomputes the MAC value over the message body received, and matches with the MAC tagged at the message body. If the two match, the message is authenticated to be sent from Alice. A key-chain is some time used. After the current message is sent and authenticated at the receiver, both Alice and Bob advance their key by one, so new MAC key is generated and replaces the key used for last communication. Such method is used to avoid replay attack.

[1]The hash values shown here in this section are arbitrary numbers for demonstration purpose only.

Appendix B

Cricket system bugs and fixes

There are several bugs found in the Cricket software system. This Appendix will describes the bugs and corresponding fixes.

B.1 Ultrasound transmitter power shut down

In the ultrasound primitive functions provided in the file 'UltrasoundControlm.nc', the ultrasound transmitter sends pulses by command 'UltrasoundControl.SendPulse', and stopped by interrupt handler
'TOSH_INTERRUPT(SIG_OUTPUT_COMPARE2)'. However the two functions start and stop the transmitter by enable and disable the FET (Field Effect Transistor) transistor Q2, which in turn shuts down the device U18 (MAX864). The U18 is a DC-DC voltage converter supply power to OP-AMP (U17) that drives the ultrasound transmitter (US2). When U18 is shut down, the U17 will be shut down. When the program call the command to send a sequence of ultrasound pulses, the transmitter will be powered for a while, and shut down soon. The fault effect can be seen in figure B.1.

The oscilloscope is adjusted to time base of $200us/div$. The first $200us$ shows the ultrasound pulses. After the end of the pulses, a slow discharge effect can be

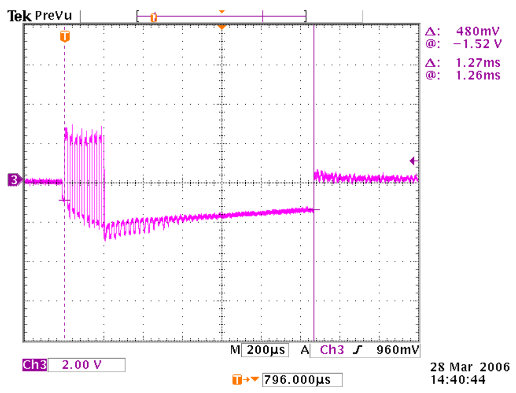


Figure B.1: Ultrasound transmitter power shut down

seen. This effect lasts for more than 1ms. This fault may influence the accuracy of signal duration. The solution is to disable the the following instruction in the interrupt handle 'TOSH_INTERRUPT(SIG_OUTPUT_COMPARE2)':
sbi(PORTG,2);

The correct signal is shown in figure B.2.

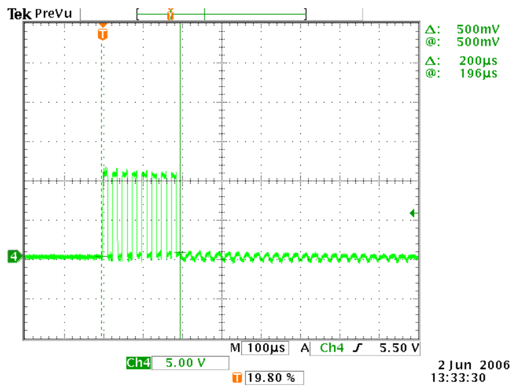


Figure B.2: Correct ultrasound signal

B.2 Ultrasound receiver power shut down

Similar problem also found in ultrasound receiver software. In ultrasound primitive functions, command ‘UltrasoundControl.StartDetector’ and ‘UltrasoundControl.StopDetector’ are used to start and stop the detector. It not only manipulates the timers/counters, it also enables and disables the FET Q3, which directly control the power source of U13A and U14. The U13A and U14 are two OP-AMPs that used to detect signal level and generate interrupts to microcontroller. The fault effect is when detector is stopped and restarted again, an interrupt will be generated even without ultrasound in the channel. This fault will generate false detections, which influence the correct program execution.

The solution to this fault is to disable the following instrcution in command ‘UltrasoundControl.StartDetector’:
cbi(PORTB, 4);
and the following in ‘UltrasoundControl.StopDetector’:
sbi(PORTB,4);
To correct enable and disable the power of ultrasound receiver, TinyOS API can be used:
‘TOSH_SET_US_IN_EN_PIN()’
and
‘TOSH_CLR_US_IN_EN_PIN()’.

B.3 Ultrasound receiver timer out failure

The ultrasound receiver, the detector has a time-out interval defined in ‘UltrasoundControl.StartDetector’ command. The time-out is set by calling ‘__outw(OCR1AL, timeout)’ instruction to set timeout value to register ‘OCR1AL’. The prototype of this instruction is defined in ‘avrhardware.h’ as:

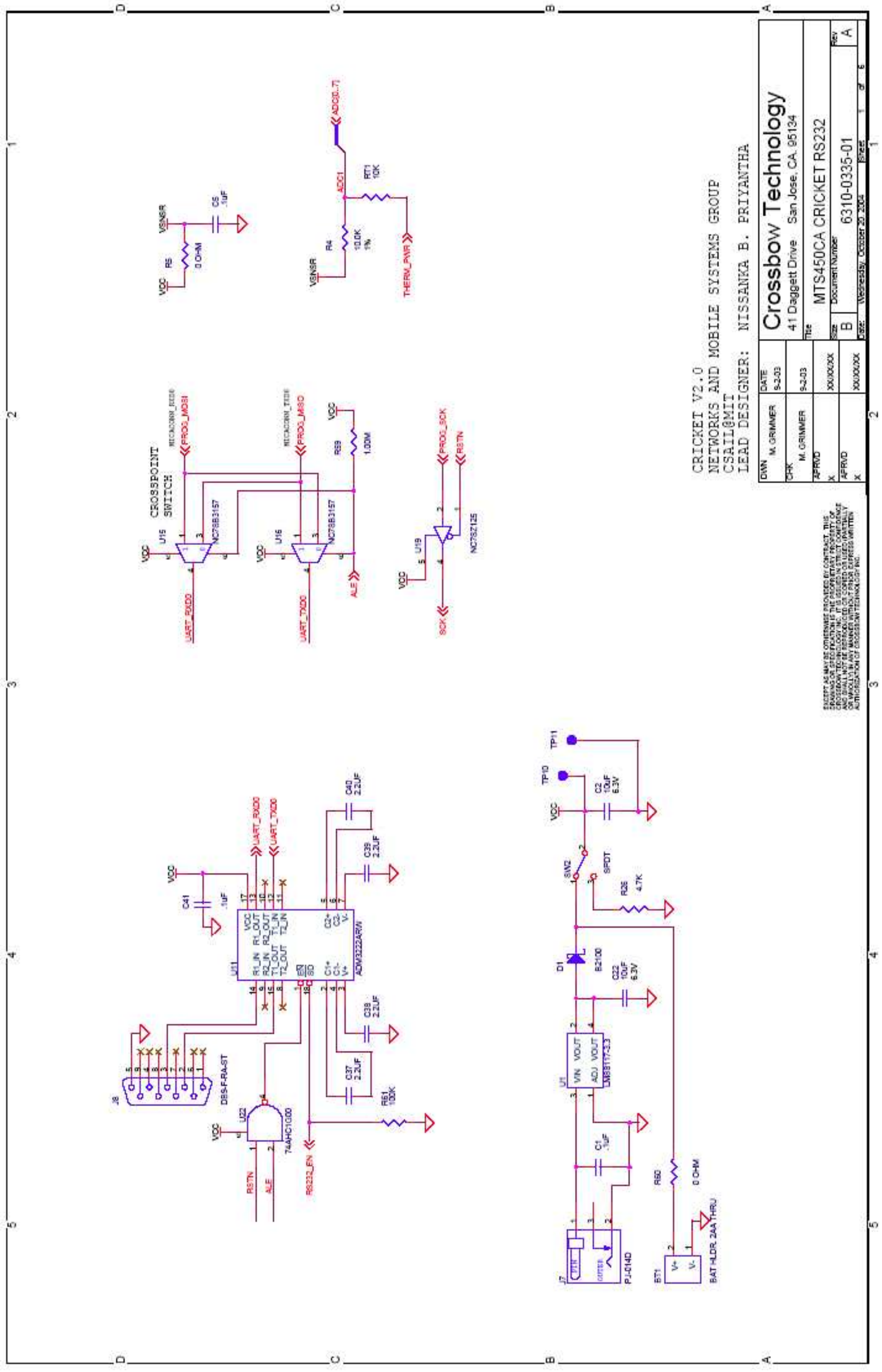
```
#define __outw(val, port) outw(port, val);
```

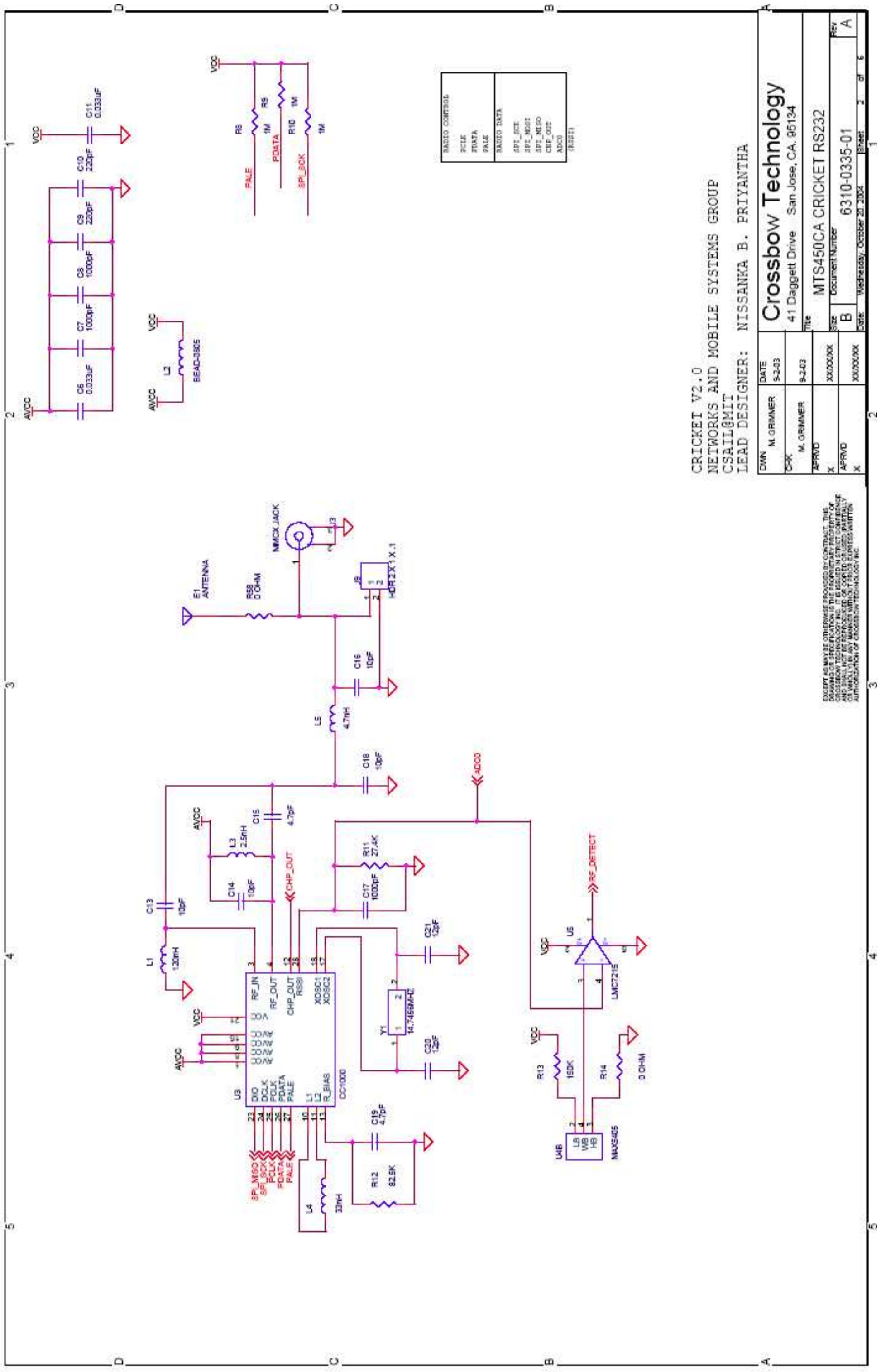
Notice, in the prototype, the time-out value first, port number after. The Cricket source code has swapped the two, which means that an illegal value (OCR1AL) is written to port that is specified by the time-out. The correct time-out value has not been written to the correct register. The solution to this bug is to replace original instruction with the following:

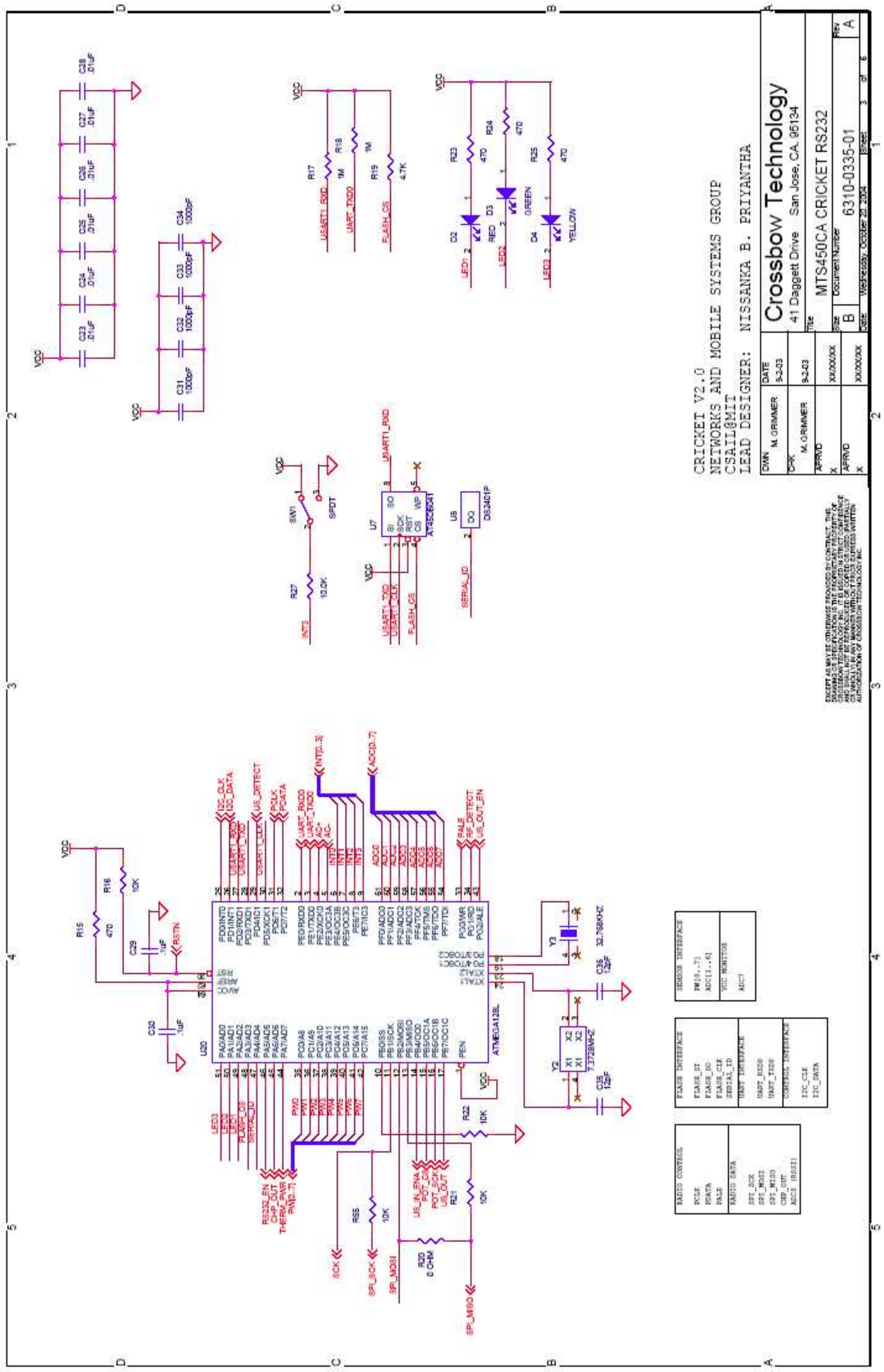
```
__outw(timeout , OCR1AL);
```

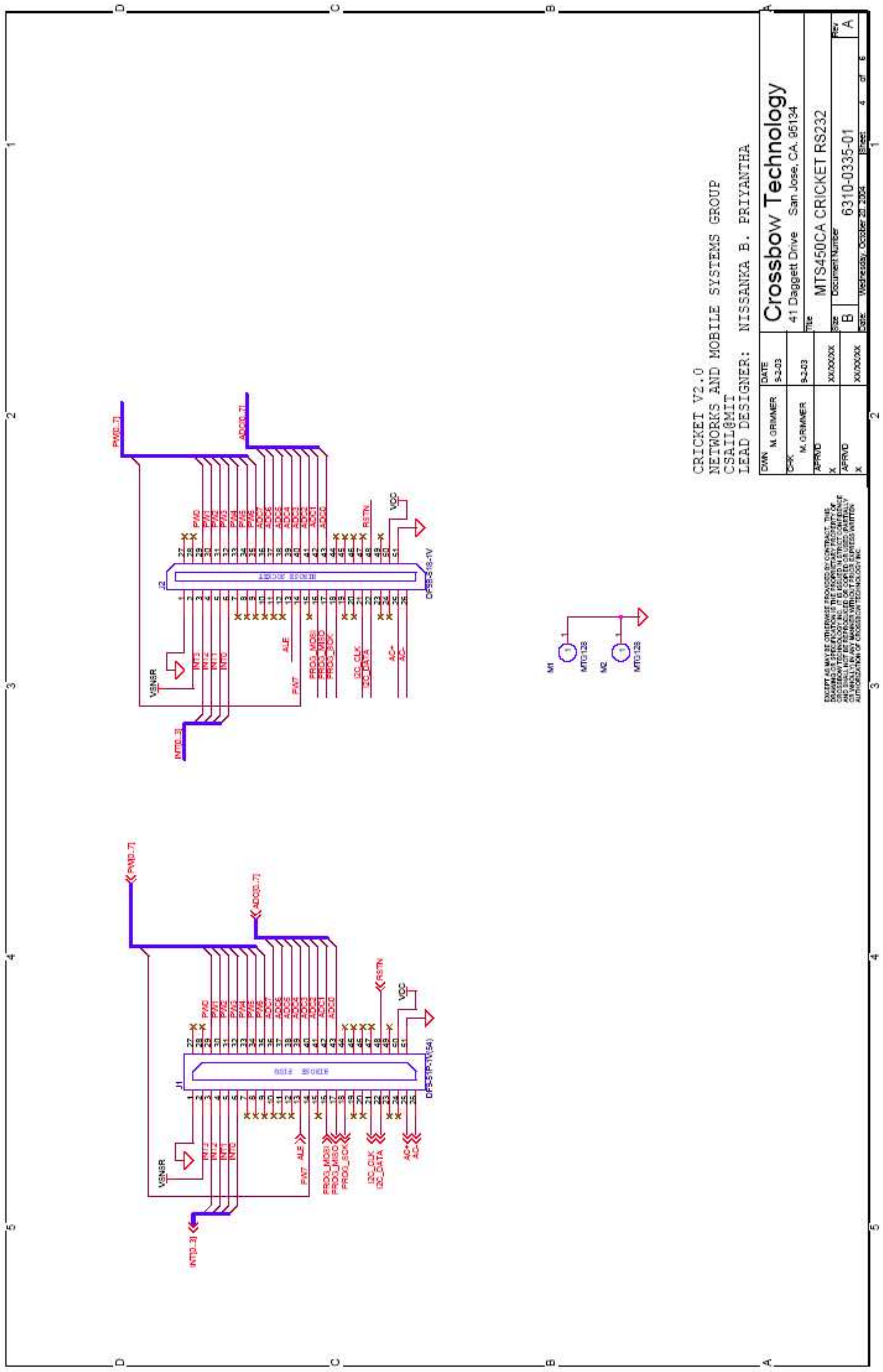
Appendix C

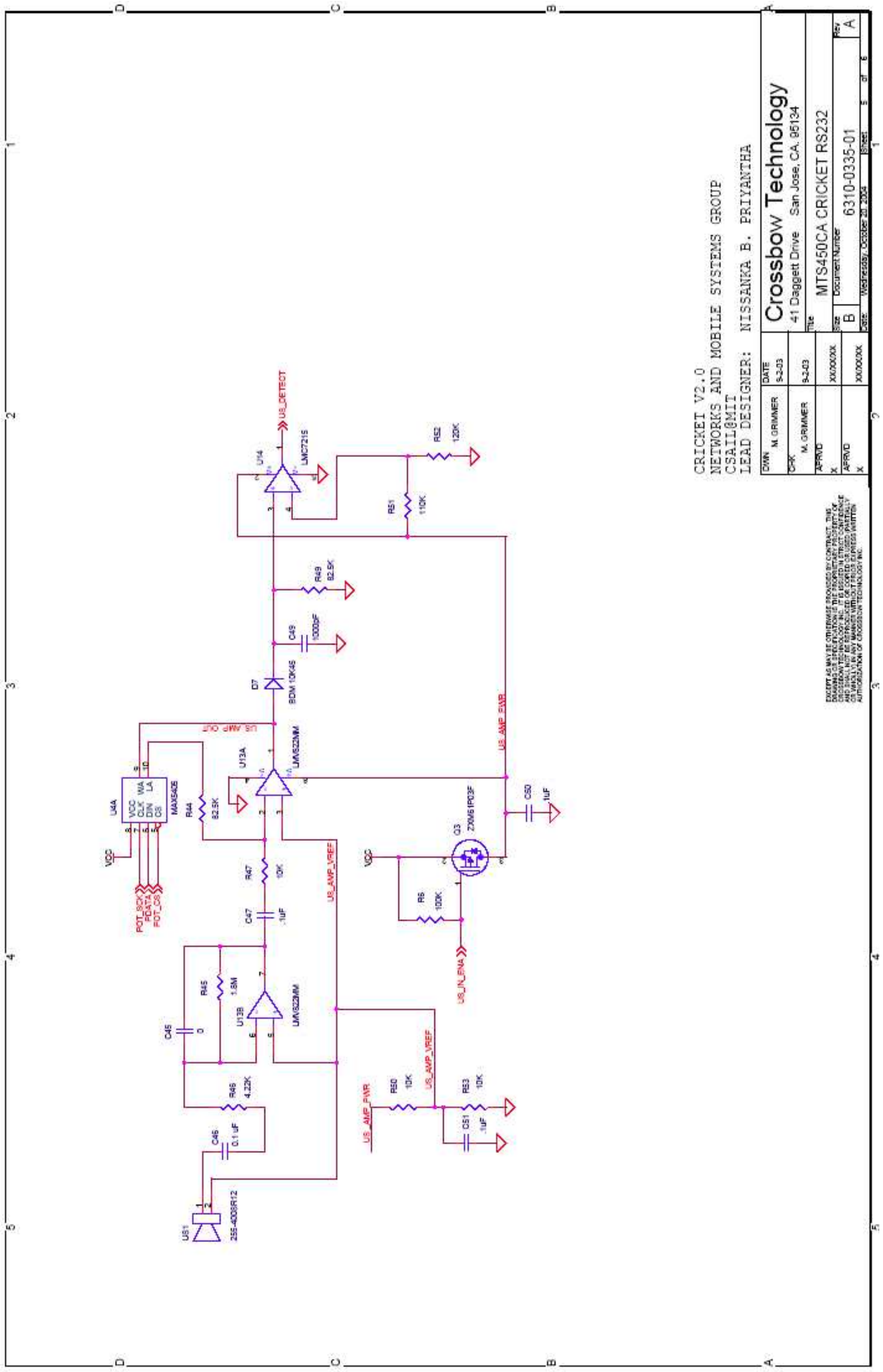
Cricket Circuit Diagrams

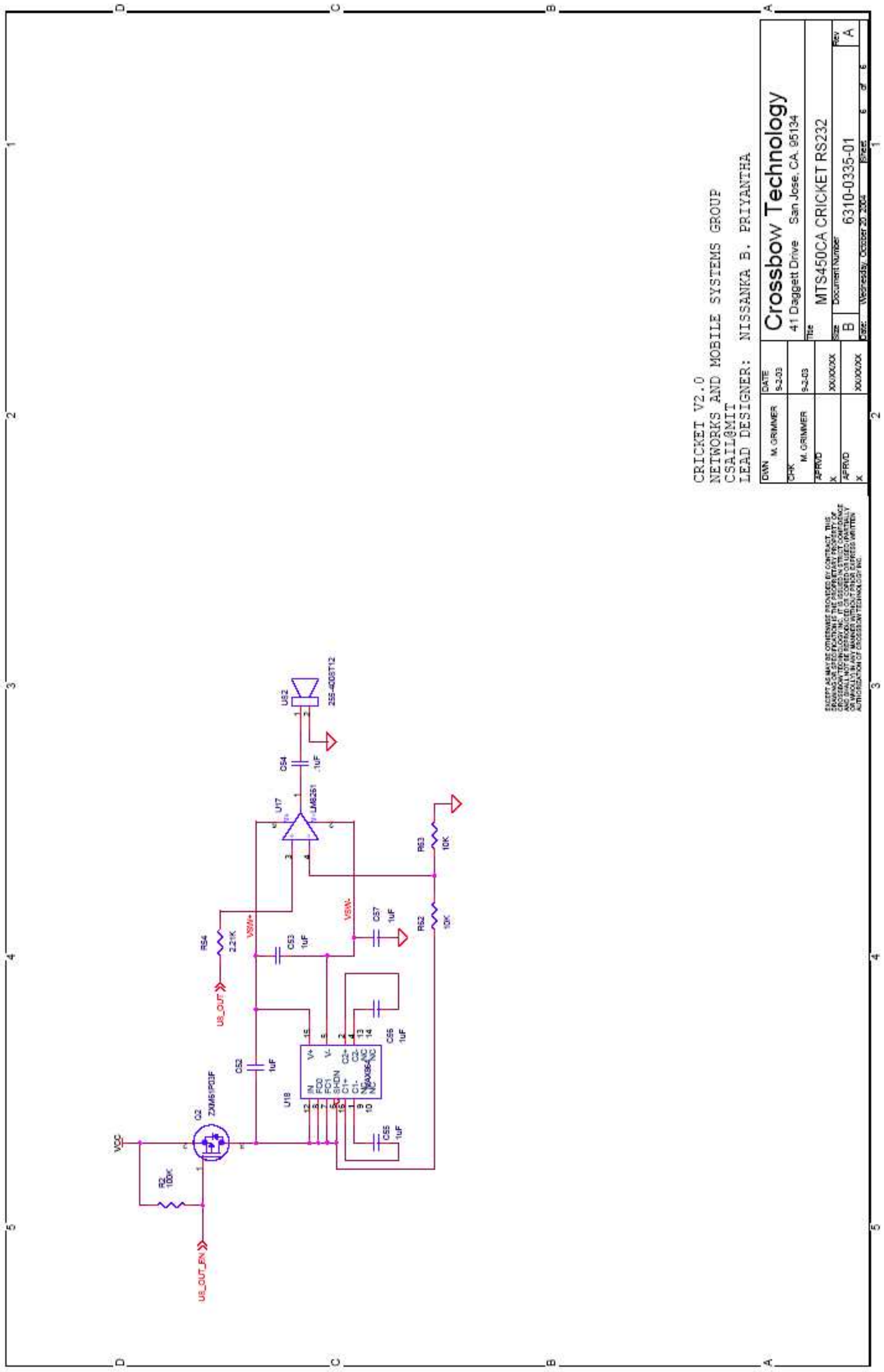

CRICKET V2.0
NETWORKS AND MOBILE SYSTEMS GROUP
CSAIL@MIT
LEAD DESIGNER: NISSANKA B. PRIYANTHA
Crossbow Technology
41 Daggett Drive San Jose, CA 95134
MTS450CA CRICKET RS232
6310-0335-01
Wednesday, October 20, 2004
CROSSPOINT SWITCH
ADM3222ARW
LM1117-3.3
NC7SB3157
NC7SZ125
74AHC1G00
DB9-F-RA-ST
BAT HLDR, 2AA THRU
VCC
UART_RXD0
UART_TXD0
PROG_MOSI
PROG_MISO
PROG_SCK
RSTN
ALE
SCK
RS232_EN
VSNSR
THERM_PWR
ADC[0..7]


CRICKET V2.0
NETWORKS AND MOBILE SYSTEMS GROUP
CSAIL@MIT
LEAD DESIGNER: NISSANKA B. PRIYANTHA
Crossbow Technology
41 Daggett Drive San Jose, CA. 95134
MTS450CA CRICKET RS232
6310-0335-01
Wednesday, October 20, 2004
Sheet 2 of 6
Rev A
CC1000
LMC7215
MAX6405
MMCX JACK
ANTENNA
14.7456MHZ
BEAD-0805
RADIO CONTROL
RADIO DATA
SPI_MISO
SPI_SCK
PCLK
PDATA
PALE
CHP_OUT
ADC0
RF_DETECT


CRICKET V2.0
NETWORKS AND MOBILE SYSTEMS GROUP
CSAIL@MIT
LEAD DESIGNER: NISSANKA B. PRIYANTHA
Crossbow Technology
41 Daggett Drive San Jose, CA. 95134
MTS450CA CRICKET RS232
6310-0335-01
Wednesday, October 20, 2004
Sheet 3 of 6
ATMEGA128L
AT45DB041
DS2401P
SPDT
7.3728MHZ
32.768KHZ
RED
GREEN
YELLOW


CRICKET V2.0
NETWORKS AND MOBILE SYSTEMS GROUP
CSAIL@MIT
LEAD DESIGNER: NISSANKA B. PRIYANTHA
Crossbow Technology
41 Daggett Drive San Jose, CA. 95134
MTS450CA CRICKET RS232
Document Number
6310-0335-01
Size B
Rev A
Date: Wednesday, October 20, 2004
Sheet 4 of 6
DWN M. GRIMMER 9-2-03
CHK M. GRIMMER 9-2-03
APRVD XXXXXXX
J1
J2
M1 MTG128
M2 MTG128
PW[0..7]
ADC[0..7]
INT[0..3]
VSNSR
ALE
PW7
PROG_MOSI
PROG_MISO
PROG_SCK
I2C_CLK
I2C_DATA
AC+
AC-
RSTN
VCC
DF9-51P-1V(54)
DF9B-51S-1V


CRICKET V2.0
NETWORKS AND MOBILE SYSTEMS GROUP
CSAIL@MIT
LEAD DESIGNER: NISSANKA B. PRIYANTHA
Crossbow Technology
41 Daggett Drive San Jose, CA. 95134
MTS450CA CRICKET RS232
6310-0335-01
Wednesday, October 20, 2004
DWN M. GRIMMER 9-2-03
CHK M. GRIMMER 9-2-03
APPVD XXXXXXX
Rev A
Size B
US_DETECT
U14 LMC7215
R52 120K
R51 110K
R49 82.5K
C49 1000pF
D7 BDM 10K45
US_AMP_OUT
US_AMP_PWR
U13A LMV822MM
U4A MAX5405
R44 82.5K
VCC
POT_SCK
PDATA
POT_CS
R47 10K
C47 .1uF
US_AMP_VREF
Q3 ZXM61P03F
C50 .1uF
R8 100K
US_IN_ENA
R45 1.8M
U13B LMV822MM
C45 0
R46 4.22K
C46 0.1 uF
US1 255-400SR12
US_AMP_PWR
R50 10K
R53 10K
C51 .1uF


CRICKET V2.0
NETWORKS AND MOBILE SYSTEMS GROUP
CSAIL@MIT
LEAD DESIGNER: NISSANKA B. PRIYANTHA
Crossbow Technology
41 Daggett Drive San Jose, CA 95134
MTS450CA CRICKET RS232
6310-0335-01
DWN M. GRIMMER 9-2-03
CHK M. GRIMMER 9-2-03
APRVD
Size B
Rev A
Date: Wednesday, October 20, 2004
Sheet 6 of 6
VCC
Q2 ZXM61P03F
R2 100K
U8_OUT_EN
U8_OUT
R54 2.21K
VSW+
VSW-
C52 1uF
C53 1uF
C55 1uF
C56 1uF
C57 1uF
U18 MAX865
U17 LM8261
C54 .1uF
R62 10K
R63 10K
U82 255-400ST12

Appendix D

Modelling of ultrasound transceiver FSM

This Appendix describes the UPPAAL model of ultrasound transceiver. The model uses formal prove with CTL logic. The coarse repesentation was explained in section 4.4.3. The model include a receiver, a transmitter. To test the transceiver, a sender and a reader are also created.

D.1 Ultrasound receiver FSM

The receiver model start from 'Idle' location, indicate channel idle. There are two edges out from this location; one is leading to 'StartBit', the other leads back to 'Idle'. The invariant 'rx_clk¡=1' indicates that one of the edges has to be taken when the clock increment to '1'. One edge can be taken when there is no ultrasound signal detected, that is guarded by 'Signal==0'. If this edge is taken, the system reset the receiver bit index ('bit_index') and the clock ('rx_clk'). If there is an ultrasound signal detected (by guard 'Signal==1'), that means the transmitter has send a pulse to channel to synchronize the receiver. The system will be taken to 'StartBit' location. The system will stay in this location for time duration specified by the guard and the invariant. The time duration must satisfy the transceiver bit rate requirement ('rx_clk¡=DATA_RATE').

When both inveriant and guard are satisfied, the edge will be taken and bring the system to 'DataBits' location to receive data. During the transition, receiver clock and bit index are reset. There are 5 possible outgoing edges to 3 locations. Two edges leads to location 'ReceivedOne', two edges leads to locaiton 'ReceivedZero' and one edge leads to 'StopBit1'. Location 'ReceivedOne' indicate that a logic '1' is received by the receiver, where 'ReceivedZero' indicate that a logic '0' is received. The different between the two is whether an ultrasound signal is detected. If ultrasound signal is found (noted by guard 'signal==1'), a '1' is concluded; otherwise a '0' is concluded. An addition guard is used to ensure the data rate ('rx_clk==DATA_RATE'). However error may occur during transmission and falsely conclude a '1' or '0'. This is modelled by the guard 'FAULT==1' in the second transition. After receive one bit, the system increment the bit counter ('bit_index++') and reset receiver clock ('rx_clk=0'), bring the system back to 'DataBits' location. When all the data bits are received ('bit_index==DATA_WIDTH'), the last edge from this location will be taken and bring the system to 'StopBit1' location. This location and the one follows it ('StopBit2') are used to receive stop bits. When two stops are received, the last edge of the system will be taken, bring the system back to idle and signal the 'Reader' process that data has been received. Notice that location 'ReceivedOne', 'ReceivedZero' and 'Stopbit2' are marked as urgent, so this location will not spend any time. It will take the outgoing transition as soon as possible.

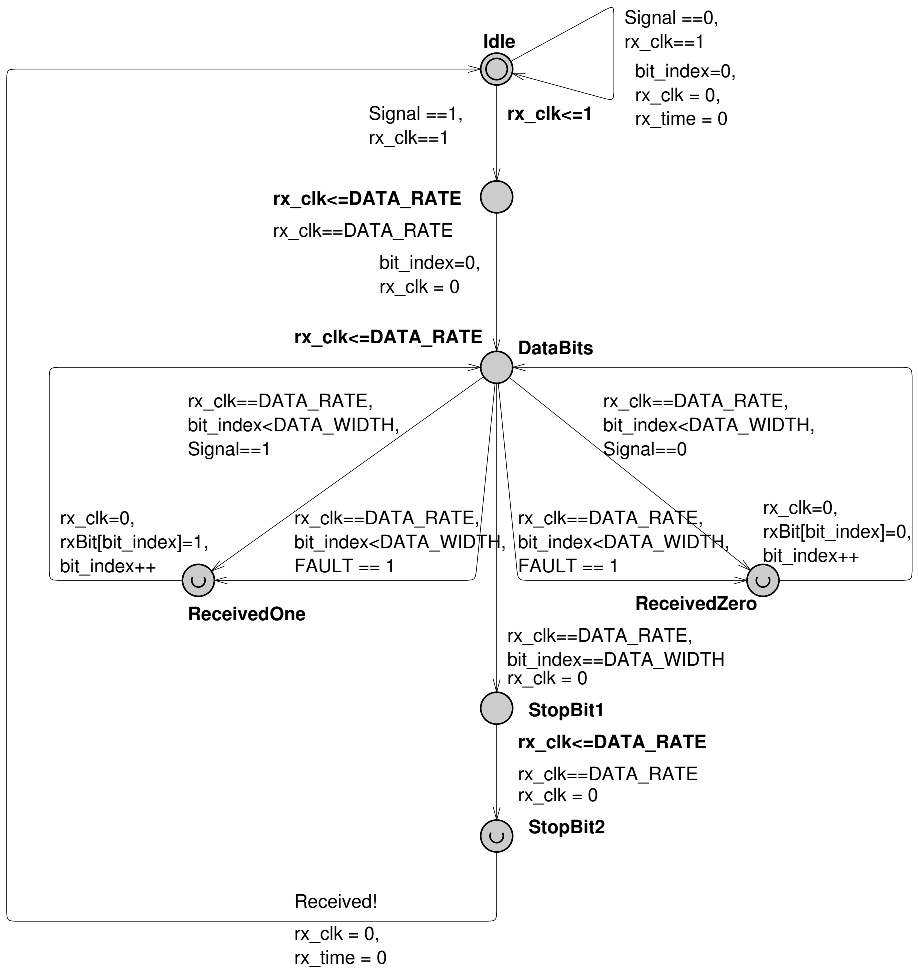


Figure D.1: US receiver FSM

D.2 Ultrasound transmitter FSM

Similarly, the transmitter start from 'Idle' location. When there is a data packet to be sent in ultrasound, the sender will signal the transmitter by 'Send!', correspondingly the transmitter is signaled by 'Send?'. This will take the edge and bring the system to 'StartBit' location. At the same time set send ultrasound pulses ('Signal=1') to synchronize the channel. After the amount of time specified by the invariant and guard ('tx_clk¡=DATA_RATE', 'tX_CLK==DATA_RATE'), the system take the edge and fired to location 'DataBits'. From this location, the transmitter start sending actual data bits. The transmission is divided into two phases, a signal generation and signal stablisation (noted by location 'Sending' and 'Sent' respectively). The signal generation is to determine what to send (by assignment 'Signal=(txbit[bit_index]==1)?1 0'). and generate corresonding signal to the channel. The signal stablisation is to satisfy the bit rate (timing) requirement. After one bit is actually sent, the system will increment the bit counter ('bit_index++') and fired back to 'DataBits' location. When every data bits are sent ('bit_index¿=DATA_WIDTH'), the system will take edge from 'DataBits' to 'StopBit1' location. Similar to receiver, two stop bits are defined. The transmitter signal the sender by 'SendDone!' and back to idle.

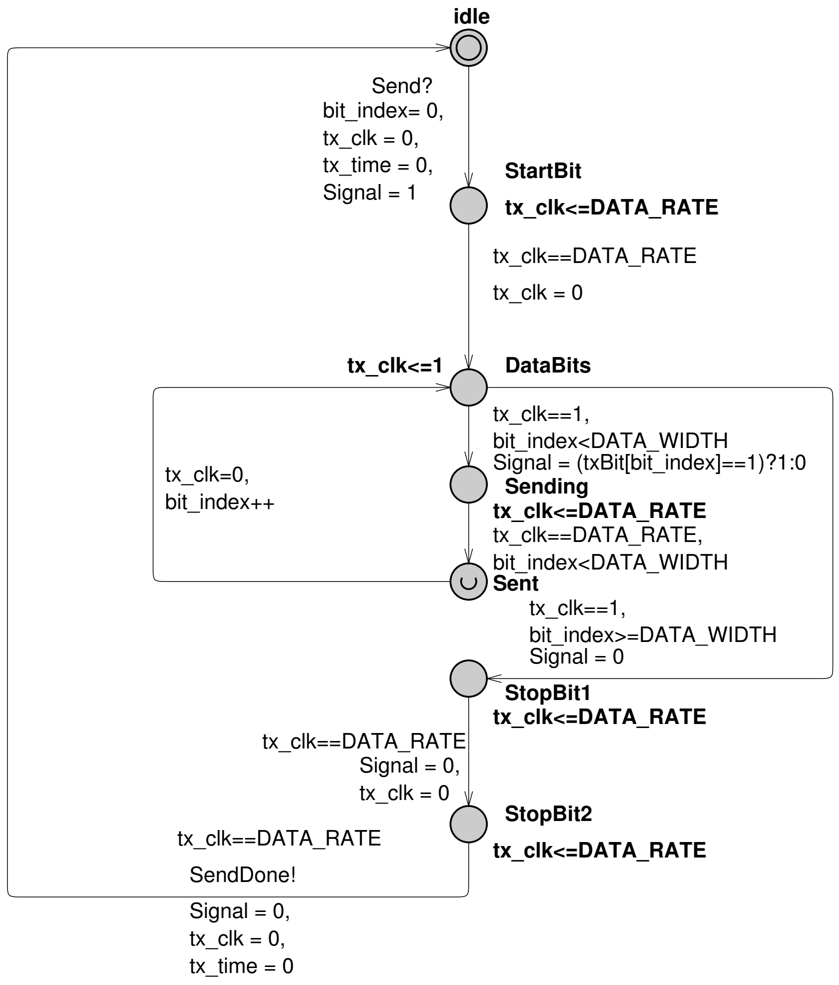


Figure D.2: US transmitter FSM

D.3 Sender & Reader

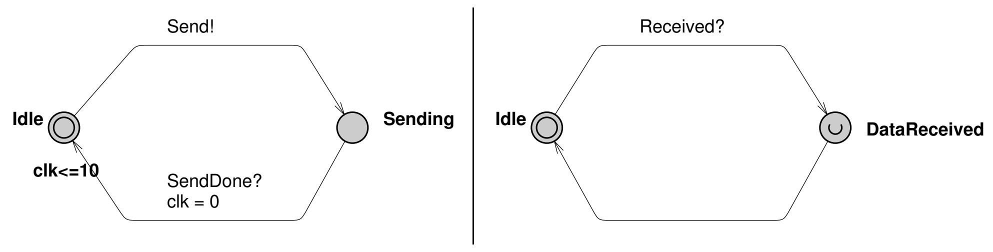


Figure D.3: US transmitter FSM

D.4 CTL verification

//Verification.q //This file was generated from UPPAAL 3.4.11, Jun 2005

```
/*
The receiver can obtain the data within limited time
For all path, always satisfy the following requirement:
( One start bit + Two stop bits + Data Bits ) * Data rate = Communication time
*/
A[] (Receiver.rx_time<=(1+DATA_WIDTH+2)*DATA_RATE)


/*
The data is sent within limited time
For all path, always satisfy the following requirement:
( One start bit + Two stop bits + Data Bits ) * Data rate = Communication time
*/
A[] (Transmitter.tx_time<=(1+DATA_WIDTH+2)*DATA_RATE)


/*
What has been received is what was transmitted
For all path, the received value eventually is the same as sent value
rxBit[i] = txBit[i]
*/
A<>(Receiver.StopBit2 and (Receiver.rxBit[7] == Transmitter.txBit[7])
and (Receiver.rxBit[6] == Transmitter.txBit[6]) and (Receiver.rxBit[5]
== Transmitter.txBit[5]) and (Receiver.rxBit[4] == Transmitter.txBit[4])
and (Receiver.rxBit[3] == Transmitter.txBit[3]) and (Receiver.rxBit[2]
== Transmitter.txBit[2]) and (Receiver.rxBit[1] == Transmitter.txBit[1])
and (Receiver.rxBit[0] == Transmitter.txBit[0]))


/*
The data can be sent
For all path, eventually in StopBit2 location
*/
A<>(Transmitter.StopBit2)


/*
The receiver can stop
For all path, eventually in StopBit2 location
*/
A<>(Receiver.StopBit2)


/*
System without deadlock
*/
A[](!deadlock)
```

Appendix E

Test result

E.1 Test of US transmission

Test of transmission establishment

```
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
```

```
 data received: 130
 data received: 31
 data received: 32
 data received: 133
```

<u>Test of correctness with exhaustive test</u>

```
data received: 100             data received: 121
data received: 1               data received: 122
data received: 2               data received: 23
data received: 103             data received: 124
data received: 4               data received: 25
data received: 105             data received: 26
data received: 106             data received: 127
data received: 7               data received: 128
data received: 8               data received: 29
data received: 109             data received: 2a
data received: 10a             data received: 12b
data received: b               data received: 2c
data received: 10c             data received: 12d
data received: d               data received: 12e
data received: e               data received: 2f
data received: 10f             data received: 130
data received: 10              data received: 31
data received: 111             data received: 32
data received: 112             data received: 133
data received: 13              data received: 34
data received: 114             data received: 135
data received: 15              data received: 136
data received: 16              data received: 37
data received: 117             data received: 38
data received: 118             data received: 139
data received: 19              data received: 13a
data received: 1a              data received: 3b
data received: 11b             data received: 13c
data received: 1c              data received: 3d
data received: 11d             data received: 3e
data received: 11e             data received: 13f
data received: 1f              data received: 40
data received: 20              data received: 141
data received: 142             data received: 6e
data received: 43              data received: 16f
data received: 144             data received: 70
data received: 45              data received: 171
data received: 46              data received: 172
data received: 147             data received: 73
data received: 148             data received: 174
data received: 49              data received: 75
data received: 4a              data received: 76
data received: 14b             data received: 177
data received: 4c              data received: 178
data received: 14d             data received: 79
data received: 14e             data received: 7a
data received: 4f              data received: 17b
data received: 150             data received: 7c
data received: 51              data received: 17d
data received: 52              data received: 17e
data received: 153             data received: 7f
data received: 54              data received: 80
data received: 155             data received: 181
data received: 156             data received: 182
data received: 57              data received: 83
data received: 58              data received: 184
data received: 159             data received: 85
data received: 15a             data received: 86
data received: 5b              data received: 187
data received: 15c             data received: 188
data received: 5d              data received: 89
data received: 5e              data received: 8a
data received: 15f             data received: 18b
data received: 160             data received: 8c
data received: 61              data received: 18d
data received: 62              data received: 18e
data received: 163             data received: 8f
data received: 64              data received: 190
data received: 165             data received: 91
data received: 166             data received: 92
data received: 67              data received: 193
data received: 68              data received: 94
data received: 169             data received: 195
data received: 16a             data received: 196
data received: 6b              data received: 97
data received: 16c             data received: 98
data received: 6d              data received: 199
data received: 19a
data received: 9b
data received: 19c
data received: 9d
data received: 9e
data received: 19f
data received: 1a0
data received: a1
data received: a2
data received: 1a3
data received: a4
data received: 1a5
data received: 1a6
data received: a7
data received: a8
data received: 1a9
data received: 1aa
data received: ab
data received: 1ac
data received: ad
data received: ae
data received: 1af
data received: b0
data received: 1b1
data received: 1b2
data received: b3
data received: 1b4
data received: b5
data received: b6
data received: 1b7
data received: 1b8
data received: b9
data received: ba
data received: 1bb
data received: bc
data received: 1bd
data received: 1be
data received: bf
data received: 1c0
data received: c1
data received: c2
data received: 1c3
data received: c4
data received: 1c5
data received: 1c6
data received: c7
data received: c8
data received: 1c9
data received: 1ca
data received: cb
data received: 1cc
data received: cd
data received: ce
data received: 1cf
data received: d0
data received: 1d1
data received: 1d2
data received: d3
data received: 1d4
data received: d5
data received: d6
data received: 1d7
data received: 1d8
data received: d9
data received: da
data received: 1db
data received: dc
data received: 1dd
data received: 1de
data received: df
data received: e0
data received: 1e1
data received: 1e2
data received: e3
data received: 1e4
data received: e5
data received: e6
data received: 1e7
data received: 1e8
data received: e9
data received: ea
data received: 1eb
data received: ec
data received: 1ed
data received: 1ee
data received: ef
data received: 1f0
data received: f1
data received: f2
data received: 1f3
data received: f4
data received: 1f5
data received: 1f6
data received: f7
data received: f8
data received: 1f9
data received: 1fa
data received: fb
data received: 1fc
data received: fd
data received: fe
data received: 1ff
data received: 100
data received: 1
data received: 2
```

Test of error detection mechanism

```
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 1b0
Rcv Wrong: 1b0
data received: 31
data received: 32
data received: 133
data received: 13c
data received: 31
data received: 32
data received: 133
data received: 130
data received: 131
Rcv Wrong: 131
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 1f3
data received: 130
data received: 31
data received: 32
data received: 173
Rcv Wrong: 173
data received: 130
data received: 31
data received: 3f
Rcv Wrong: 3f
data received: 133
data received: 130
data received: 3d
data received: 32
data received: 133
data received: 131
Rcv Wrong: 131
data received: 31
data received: 32
data received: 1f3
data received: 130
data received: 31
data received: 32
data received: 133
data received: 137
Rcv Wrong: 137
data received: 31
data received: 32
data received: 173
Rcv Wrong: 173
data received: 130
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
data received: 63
Rcv Wrong: 63
data received: 31
data received: 32
data received: 133
data received: 130
data received: 31
data received: 32
data received: 133
```

E.2 Test of ultrasound ranging

```
// Test condition
// Distance ~= 30cm
// Result = ok
// Duration ~= 60 sec.

Sent Time: 0 us
backtime: 1653 us
offset: 2
uncorrected time: 1653 us,integrity time: 893 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1596 us
offset: 1
uncorrected time: 1596 us,integrity time: 884 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1546 us
offset: 0
uncorrected time: 1546 us,integrity time: 882 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1790 us
```

```
offset: 5
uncorrected time: 1790 us,integrity time: 886 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1642 us
offset: 2
uncorrected time: 1642 us,integrity time: 882 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1594 us
offset: 1
uncorrected time: 1594 us,integrity time: 882 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1548 us
offset: 0
uncorrected time: 1548 us,integrity time: 884 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1645 us
offset: 2
uncorrected time: 1645 us,integrity time: 885 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1656 us
offset: 2
uncorrected time: 1656 us,integrity time: 896 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1656 us
offset: 2
uncorrected time: 1656 us,integrity time: 896 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1798 us
offset: 5
uncorrected time: 1798 us,integrity time: 894 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1920 us
offset: 7
uncorrected time: 1920 us,integrity time: 920 us, Distance: 31 cm
Sent Time: 0 us
backtime: 1753 us
offset: 4
uncorrected time: 1753 us,integrity time: 897 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1752 us
offset: 4
uncorrected time: 1752 us,integrity time: 896 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1606 us
```

```
offset: 1
uncorrected time: 1606 us,integrity time: 894 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1558 us
offset: 0
uncorrected time: 1558 us,integrity time: 894 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1751 us
offset: 4
uncorrected time: 1751 us,integrity time: 895 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1920 us
offset: 7
uncorrected time: 1920 us,integrity time: 920 us, Distance: 31 cm
Sent Time: 0 us
backtime: 1751 us
offset: 4
uncorrected time: 1751 us,integrity time: 895 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1609 us
offset: 1
uncorrected time: 1609 us,integrity time: 897 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1750 us
offset: 4
uncorrected time: 1750 us,integrity time: 894 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1705 us
offset: 3
uncorrected time: 1705 us,integrity time: 897 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1559 us
offset: 0
uncorrected time: 1559 us,integrity time: 895 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1706 us
offset: 3
uncorrected time: 1706 us,integrity time: 898 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1558 us
offset: 0
uncorrected time: 1558 us,integrity time: 894 us, Distance: 30 cm
Sent Time: 0 us
backtime: 1839 us
```

```
offset: 6
uncorrected time: 1839 us,integrity time: 887 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1644 us
offset: 2
uncorrected time: 1644 us,integrity time: 884 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1838 us
offset: 6
uncorrected time: 1838 us,integrity time: 886 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1695 us
offset: 3
uncorrected time: 1695 us,integrity time: 887 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1789 us
offset: 5
uncorrected time: 1789 us,integrity time: 885 us, Distance: 29 cm
Sent Time: 0 us
backtime: 1610 us
offset: 1
uncorrected time: 1610 us,integrity time: 898 us, Distance: 30 cm


// Test condition
// Distance ~= 60cm
// Result = ok
// Duration ~= 60 sec.

Sent Time: 0 us
backtime: 2381 us
offset: 0
uncorrected time: 2381 us,integrity time: 1717 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2493 us
offset: 2
uncorrected time: 2493 us,integrity time: 1733 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2406 us
offset: 0
uncorrected time: 2406 us,integrity time: 1742 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2442 us
offset: 1
```

```
uncorrected time: 2442 us,integrity time: 1730 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2528 us
offset: 3
uncorrected time: 2528 us,integrity time: 1720 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2452 us
offset: 1
uncorrected time: 2452 us,integrity time: 1740 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2444 us
offset: 1
uncorrected time: 2444 us,integrity time: 1732 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2635 us
offset: 5
uncorrected time: 2635 us,integrity time: 1731 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2615 us
offset: 5
uncorrected time: 2615 us,integrity time: 1711 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2745 us
offset: 7
uncorrected time: 2745 us,integrity time: 1745 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2516 us
offset: 3
uncorrected time: 2516 us,integrity time: 1708 us, Distance: 57 cm
Sent Time: 0 us
backtime: 2403 us
offset: 0
uncorrected time: 2403 us,integrity time: 1739 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2635 us
offset: 5
uncorrected time: 2635 us,integrity time: 1731 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2686 us
offset: 6
uncorrected time: 2686 us,integrity time: 1734 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2454 us
offset: 1
```

```
uncorrected time: 2454 us,integrity time: 1742 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2539 us
offset: 3
uncorrected time: 2539 us,integrity time: 1731 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2624 us
offset: 5
uncorrected time: 2624 us,integrity time: 1720 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2492 us
offset: 2
uncorrected time: 2492 us,integrity time: 1732 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2373 us
offset: 0
uncorrected time: 2373 us,integrity time: 1709 us, Distance: 57 cm
Sent Time: 0 us
backtime: 2693 us
offset: 6
uncorrected time: 2693 us,integrity time: 1741 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2586 us
offset: 4
uncorrected time: 2586 us,integrity time: 1730 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2646 us
offset: 5
uncorrected time: 2646 us,integrity time: 1742 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2443 us
offset: 1
uncorrected time: 2443 us,integrity time: 1731 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2697 us
offset: 6
uncorrected time: 2697 us,integrity time: 1745 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2615 us
offset: 5
uncorrected time: 2615 us,integrity time: 1711 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2405 us
offset: 0
```

```
uncorrected time: 2405 us,integrity time: 1741 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2441 us
offset: 1
uncorrected time: 2441 us,integrity time: 1729 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2694 us
offset: 6
uncorrected time: 2694 us,integrity time: 1742 us, Distance: 59 cm
Sent Time: 0 us
backtime: 2530 us
offset: 3
uncorrected time: 2530 us,integrity time: 1722 us, Distance: 58 cm
Sent Time: 0 us
backtime: 2615 us
offset: 5
uncorrected time: 2615 us,integrity time: 1711 us, Distance: 58 cm



// Test condition
// Distance ~= 100cm
// Result = ok
// Duration ~= 60 sec.

uncorrected time: 3644 us,integrity time: 2932 us, Distance: 99 cm
Sent Time: 0 us
backtime: 3629 us
offset: 0
uncorrected time: 3629 us,integrity time: 2965 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3910 us
offset: 6
uncorrected time: 3910 us,integrity time: 2958 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3629 us
offset: 0
uncorrected time: 3629 us,integrity time: 2965 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3863 us
offset: 5
uncorrected time: 3863 us,integrity time: 2959 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3907 us
```

```
offset: 6
uncorrected time: 3907 us,integrity time: 2955 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3668 us
offset: 1
uncorrected time: 3668 us,integrity time: 2956 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3667 us
offset: 1
uncorrected time: 3667 us,integrity time: 2955 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3775 us
offset: 3
uncorrected time: 3775 us,integrity time: 2967 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3860 us
offset: 5
uncorrected time: 3860 us,integrity time: 2956 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3658 us
offset: 1
uncorrected time: 3658 us,integrity time: 2946 us, Distance: 99 cm
Sent Time: 0 us
backtime: 3616 us
offset: 0
uncorrected time: 3616 us,integrity time: 2952 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3921 us
offset: 6
uncorrected time: 3921 us,integrity time: 2969 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3836 us
offset: 4
uncorrected time: 3836 us,integrity time: 2980 us, Distance: 101 cm
Sent Time: 0 us
backtime: 3775 us
offset: 3
uncorrected time: 3775 us,integrity time: 2967 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3667 us
offset: 1
uncorrected time: 3667 us,integrity time: 2955 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3827 us
```

```
offset: 4
uncorrected time: 3827 us,integrity time: 2971 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3766 us
offset: 3
uncorrected time: 3766 us,integrity time: 2958 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3725 us
offset: 2
uncorrected time: 3725 us,integrity time: 2965 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3645 us
offset: 1
uncorrected time: 3645 us,integrity time: 2933 us, Distance: 99 cm
Sent Time: 0 us
backtime: 3900 us
offset: 6
uncorrected time: 3900 us,integrity time: 2948 us, Distance: 99 cm
Sent Time: 0 us
backtime: 3645 us
offset: 1
uncorrected time: 3645 us,integrity time: 2933 us, Distance: 99 cm
Sent Time: 0 us
backtime: 3825 us
offset: 4
uncorrected time: 3825 us,integrity time: 2969 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3618 us
offset: 0
uncorrected time: 3618 us,integrity time: 2954 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3919 us
offset: 6
uncorrected time: 3919 us,integrity time: 2967 us, Distance: 100 cm
Sent Time: 0 us
backtime: 3859 us
offset: 5
uncorrected time: 3859 us,integrity time: 2955 us, Distance: 100 cm



// Test condition
// Distance ~= 200cm
// Result = ok
// Duration ~= 60 sec.

Sent Time: 0 us
backtime: 6674 us
offset: 6
uncorrected time: 6674 us,integrity time: 5822 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6449 us
offset: 2
uncorrected time: 6449 us,integrity time: 5789 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6364 us
offset: 0
uncorrected time: 6364 us,integrity time: 5800 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6546 us
offset: 4
uncorrected time: 6546 us,integrity time: 5790 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6676 us
offset: 6
uncorrected time: 6676 us,integrity time: 5824 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6362 us
offset: 0
uncorrected time: 6362 us,integrity time: 5798 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6713 us
offset: 7
uncorrected time: 6713 us,integrity time: 5813 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6556 us
offset: 4
uncorrected time: 6556 us,integrity time: 5800 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6518 us
offset: 3
uncorrected time: 6518 us,integrity time: 5810 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6447 us
offset: 2
uncorrected time: 6447 us,integrity time: 5787 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6410 us
```

```
offset: 1
uncorrected time: 6410 us,integrity time: 5798 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6517 us
offset: 3
uncorrected time: 6517 us,integrity time: 5809 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6700 us
offset: 7
uncorrected time: 6700 us,integrity time: 5800 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6547 us
offset: 4
uncorrected time: 6547 us,integrity time: 5791 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6703 us
offset: 7
uncorrected time: 6703 us,integrity time: 5803 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6546 us
offset: 4
uncorrected time: 6546 us,integrity time: 5790 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6410 us
offset: 1
uncorrected time: 6410 us,integrity time: 5798 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6690 us
offset: 7
uncorrected time: 6690 us,integrity time: 5790 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6529 us
offset: 3
uncorrected time: 6529 us,integrity time: 5821 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6548 us
offset: 4
uncorrected time: 6548 us,integrity time: 5792 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6655 us
offset: 6
uncorrected time: 6655 us,integrity time: 5803 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6445 us
```

```
offset: 2
uncorrected time: 6445 us,integrity time: 5785 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6557 us
offset: 4
uncorrected time: 6557 us,integrity time: 5801 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6567 us
offset: 4
uncorrected time: 6567 us,integrity time: 5811 us, Distance: 197 cm
Sent Time: 0 us
backtime: 6411 us
offset: 1
uncorrected time: 6411 us,integrity time: 5799 us, Distance: 196 cm
Sent Time: 0 us
backtime: 6595 us
offset: 5
uncorrected time: 6595 us,integrity time: 5791 us, Distance: 196 cm
```

E.3 Test of commitment

Functional test

```
Clear text: 303132333435363738393a3b3c3d3e3f404142434445464748494a4b4c
MAC received: 2d 9 96 86
MAC computed: 2d,9,96,86
Clear text: 3132333435363738393a3b3c3d3e3f404142434445464748494a4b4c4d
MAC received: 30 99 4a 79
MAC computed: 30,99,4a,79
Clear text: 32333435363738393a3b3c3d3e3f404142434445464748494a4b4c4d4e
MAC received: 8a bf 23 11
MAC computed: 8a,bf,23,11
Clear text: 333435363738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f
MAC received: 9c 5e ba 80
MAC computed: 9c,5e,ba,80
Clear text: 3435363738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f50
MAC received: 47 52 42 9c
MAC computed: 47,52,42,9c
Clear text: 35363738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f5051
MAC received: 2f a4 8c 41
MAC computed: 2f,a4,8c,41
Clear text: 363738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152
MAC received: c 7f a8 e6
MAC computed: c,7f,a8,e6
Clear text: 3738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f50515253
MAC received: b0 3e a7 9d
MAC computed: b0,3e,a7,9d
Clear text: 38393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f5051525354
MAC received: 18 5e b6 2
MAC computed: 18,5e,b6,2
Clear text: 393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455
MAC received: f6 8d 7a e2
MAC computed: f6,8d,7a,e2
Clear text: 3a3b3c3d3e3f404142434445464748494a4b4c4d4e4f50515253545556
MAC received: b6 fe df 72
MAC computed: b6,fe,df,72
Clear text: 3b3c3d3e3f404142434445464748494a4b4c4d4e4f5051525354555657
MAC received: a5 3d 2d 77
MAC computed: a5,3d,2d,77
Clear text: 3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455565758
MAC received: 73 27 f3 77
MAC computed: 73,27,f3,77
Clear text: 3d3e3f404142434445464748494a4b4c4d4e4f50515253545556575859
MAC received: 5c 36 3e 6e
MAC computed: 5c,36,3e,6e
Clear text: 3e3f404142434445464748494a4b4c4d4e4f505152535455565758595a
MAC received: 2b 8e 18 25
MAC computed: 2b,8e,18,25
Clear text: 3f404142434445464748494a4b4c4d4e4f505152535455565758595a5b
MAC received: f2 4f a2 7b
MAC computed: f2,4f,a2,7b
Clear text: 404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c
MAC received: 41 da 74 8a
MAC computed: 41,da,74,8a
Clear text: 4142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d
MAC received: 39 2d 88 1a
MAC computed: 39,2d,88,1a
Clear text: 42434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e
MAC received: 41 c0 f1 2d
MAC computed: 41,c0,f1,2d
Clear text: 434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f
MAC received: 4b e9 fa 5a
MAC computed: 4b,e9,fa,5a
```

Test with attack

```
Clear text: 303132333435363738393a3b3c3d3e3f404142434445464748494a4b4c
MAC received: 2d 9 96 86
MAC computed: 2d,9,96,86
Clear text: 3132333435363738393a3b3c3d3e3f404142434445464748494a4b4c4d
MAC received: 31 0 0 0
MAC computed: 30,99,4a,79
Clear text: 32000000000000000000000048494a4b4c4d
MAC received: 33 0 0 0
MAC computed: 66,ae,c7,8d
Clear text: 30ff994a7935363738393a3b3c3d3e3f404142434445464748494a4b4c4d
MAC received: 32 33 34 35
MAC computed: 39,59,aa,2d
Clear text: ff8affbf2311363738393a3b3c3d3e3f404142434445464748494a4b4c4d4e
MAC received: 31 0 0 0
MAC computed: 3c,3e,d9,41
Clear text: 333435363738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f
MAC received: 9c 5e ba 80
MAC computed: 9c,5e,ba,80
Clear text: 3100000000000000000000004a4b4c4d4e4f
MAC received: 34 35 36 37
MAC computed: 8f,f4,cd,57
Clear text: 475242ff9c38393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f50
MAC received: 31 0 0 0
MAC computed: d1,c7,99,8d
Clear text: 3200000000000000000000004b4c4d4e4f50
MAC received: 35 36 37 38
MAC computed: d6,ab,10,31
Clear text: 2fffa4ff8c41393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f5051
MAC received: 31 0 0 0
MAC computed: 3c,12,8f,a7
Clear text: 3100000000000000000000004c4d4e4f5051
MAC received: 36 37 38 39
MAC computed: 19,4a,c8,cf
Clear text: 3200000000000000000000004d4e4f505152
MAC received: c 7f a8 e6
MAC computed: 3,69,aa,f4
Clear text: 3738393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f50515253
MAC received: b0 3e a7 9d
MAC computed: b0,3e,a7,9d
Clear text: 310000000000000000000004e4f50515253
MAC received: 38 39 3a 3b
MAC computed: d4,1f,43,8
Clear text: 320000000000000000000004f5051525354
MAC received: 18 5e b6 2
MAC computed: 70,c,c3,bc
Clear text: 393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455
MAC received: 3a 3b 3c 3d
MAC computed: f6,8d,7a,e2
```

E.4 Test of key exchange protocol

Functional test

Notation:
bt: uncorrected time
@: the bit number
ofst: bit offset
t: corrected time
D: Distance

```
========Ready for commitment=========
Commitment received:52_e4_91_e_25_7c_15_66_
bt:2498us, 0 @0, ofst:3, t:1790us, D:60cm
bt:2559us, 0 @1, ofst:4, t:1803us, D:61cm
bt:2366us, 0 @2, ofst:0, t:1802us, D:61cm
bt:2515us, 0 @3, ofst:3, t:1807us, D:61cm
bt:2623us, 0 @4, ofst:5, t:1819us, D:61cm
bt:2693us, 1 @5, ofst:6, t:1841us, D:62cm
bt:2383us, 1 @6, ofst:0, t:1819us, D:61cm
bt:2528us, 0 @7, ofst:3, t:1820us, D:61cm
bt:2462us, 1 @8, ofst:2, t:1802us, D:61cm
bt:2414us, 1 @9, ofst:1, t:1802us, D:61cm
bt:2692us, 0 @10, ofst:6, t:1840us, D:62cm
bt:2709us, 1 @11, ofst:7, t:1809us, D:61cm
bt:2613us, 1 @12, ofst:5, t:1809us, D:61cm
bt:2419us, 0 @13, ofst:1, t:1807us, D:61cm
bt:2574us, 1 @14, ofst:4, t:1818us, D:61cm
```

```
bt:2466us, 0 @15, ofst:2, t:1806us, D:61cm
bt:2559us, 0 @16, ofst:4, t:1803us, D:61cm
bt:2609us, 0 @17, ofst:5, t:1805us, D:61cm
bt:2721us, 0 @18, ofst:7, t:1821us, D:61cm
bt:2515us, 1 @19, ofst:3, t:1807us, D:61cm
bt:2416us, 0 @20, ofst:1, t:1804us, D:61cm
bt:2612us, 1 @21, ofst:5, t:1808us, D:61cm
bt:2623us, 1 @22, ofst:5, t:1819us, D:61cm
bt:2574us, 1 @23, ofst:4, t:1818us, D:61cm
bt:2681us, 0 @24, ofst:6, t:1829us, D:61cm
bt:2572us, 0 @25, ofst:4, t:1816us, D:61cm
bt:2717us, 1 @26, ofst:7, t:1817us, D:61cm
bt:2612us, 1 @27, ofst:5, t:1808us, D:61cm
bt:2467us, 0 @28, ofst:2, t:1807us, D:61cm
bt:2708us, 1 @29, ofst:7, t:1808us, D:61cm
bt:2380us, 0 @30, ofst:0, t:1816us, D:61cm
bt:2623us, 1 @31, ofst:5, t:1819us, D:61cm
bt:2621us, 1 @32, ofst:5, t:1817us, D:61cm
bt:2716us, 1 @33, ofst:7, t:1816us, D:61cm
bt:2709us, 1 @34, ofst:7, t:1809us, D:61cm
bt:2416us, 0 @35, ofst:1, t:1804us, D:61cm
bt:2612us, 1 @36, ofst:5, t:1808us, D:61cm
bt:2524us, 1 @37, ofst:3, t:1816us, D:61cm
bt:2529us, 0 @38, ofst:3, t:1821us, D:61cm
bt:2370us, 0 @39, ofst:0, t:1806us, D:61cm
bt:2678us, 0 @40, ofst:6, t:1826us, D:61cm
bt:2692us, 0 @41, ofst:6, t:1840us, D:62cm
bt:2417us, 1 @42, ofst:1, t:1805us, D:61cm
bt:2563us, 0 @43, ofst:4, t:1807us, D:61cm
bt:2513us, 1 @44, ofst:3, t:1805us, D:61cm
bt:2428us, 1 @45, ofst:1, t:1816us, D:61cm
bt:2368us, 0 @46, ofst:0, t:1804us, D:61cm
bt:2712us, 1 @47, ofst:7, t:1812us, D:61cm
bt:2609us, 1 @48, ofst:5, t:1805us, D:61cm
bt:2425us, 1 @49, ofst:1, t:1813us, D:61cm
bt:2513us, 1 @50, ofst:3, t:1805us, D:61cm
bt:2415us, 1 @51, ofst:1, t:1803us, D:61cm
bt:2466us, 0 @52, ofst:2, t:1806us, D:61cm
bt:2467us, 1 @53, ofst:2, t:1807us, D:61cm
bt:2528us, 1 @54, ofst:3, t:1820us, D:61cm
bt:2722us, 1 @55, ofst:7, t:1822us, D:61cm
bt:2415us, 1 @56, ofst:1, t:1803us, D:61cm
bt:2523us, 1 @57, ofst:3, t:1815us, D:61cm
bt:2719us, 1 @58, ofst:7, t:1819us, D:61cm
```

```
bt:2710us, 0 @59, ofst:7, t:1810us, D:61cm
bt:2683us, 1 @60, ofst:6, t:1831us, D:62cm
bt:2562us, 0 @61, ofst:4, t:1806us, D:61cm
bt:2614us, 0 @62, ofst:5, t:1810us, D:61cm
bt:2371us, 0 @63, ofst:0, t:1807us, D:61cm
NA xor NB: 60_5b_e8_ac_37_b4_ef_17_
Message in clear text: This is test string 1
MAC key of B is: 16_8d_76_76_fc_e6_15_59_
Received MAC: 52_e4_91_e_25_7c_15_66_
Computed MAC: 52_e4_91_e_25_7c_15_66_
Key established!
```

Distance shortening attack

```
========Ready for commitment=========
Commitment received:52_e4_91_e_25_7c_15_66_
bt:2178us, 0 @0, ofst:7, t:1278us, D:43cm
bt:2181us, 1 @1, ofst:7, t:1281us, D:43cm
bt:2009us, 0 @2, ofst:7, t:1109us, D:37cm
bt:2203us, 1 @3, ofst:7, t:1303us, D:44cm
bt:2254us, 1 @4, ofst:7, t:1354us, D:45cm
bt:2011us, 1 @5, ofst:7, t:1111us, D:37cm
bt:2352us, 1 @6, ofst:7, t:1452us, D:49cm
bt:2351us, 1 @7, ofst:7, t:1451us, D:49cm
bt:2009us, 1 @8, ofst:7, t:1109us, D:37cm
bt:1995us, 0 @9, ofst:7, t:1095us, D:37cm
bt:2349us, 0 @10, ofst:7, t:1449us, D:48cm
bt:2305us, 0 @11, ofst:7, t:1405us, D:47cm
bt:2136us, 0 @12, ofst:7, t:1236us, D:41cm
bt:2353us, 1 @13, ofst:7, t:1453us, D:49cm
bt:2174us, 0 @14, ofst:7, t:1274us, D:43cm
bt:2352us, 0 @15, ofst:7, t:1452us, D:49cm
bt:2204us, 1 @16, ofst:7, t:1304us, D:44cm
bt:2172us, 1 @17, ofst:7, t:1272us, D:43cm
bt:2350us, 0 @18, ofst:7, t:1450us, D:49cm
bt:2303us, 1 @19, ofst:7, t:1403us, D:47cm
bt:2352us, 0 @20, ofst:7, t:1452us, D:49cm
bt:2193us, 0 @21, ofst:7, t:1293us, D:43cm
bt:2254us, 1 @22, ofst:7, t:1354us, D:45cm
bt:2195us, 0 @23, ofst:7, t:1295us, D:43cm
bt:2190us, 1 @24, ofst:7, t:1290us, D:43cm
```

```
bt:2351us, 1 @25, ofst:7, t:1451us, D:49cm
bt:2109us, 0 @26, ofst:7, t:1209us, D:40cm
bt:2195us, 0 @27, ofst:7, t:1295us, D:43cm
bt:2194us, 0 @28, ofst:7, t:1294us, D:43cm
bt:1997us, 0 @29, ofst:7, t:1097us, D:37cm
bt:2340us, 1 @30, ofst:7, t:1440us, D:48cm
bt:2048us, 1 @31, ofst:7, t:1148us, D:38cm
bt:1997us, 1 @32, ofst:7, t:1097us, D:37cm
bt:2193us, 1 @33, ofst:7, t:1293us, D:43cm
bt:2058us, 0 @34, ofst:7, t:1158us, D:39cm
bt:2240us, 1 @35, ofst:7, t:1340us, D:45cm
bt:2096us, 0 @36, ofst:7, t:1196us, D:40cm
bt:2146us, 1 @37, ofst:7, t:1246us, D:42cm
bt:2058us, 1 @38, ofst:7, t:1158us, D:39cm
bt:2063us, 0 @39, ofst:7, t:1163us, D:39cm
bt:2057us, 1 @40, ofst:7, t:1157us, D:39cm
bt:2311us, 1 @41, ofst:7, t:1411us, D:47cm
bt:2097us, 0 @42, ofst:7, t:1197us, D:40cm
bt:2304us, 0 @43, ofst:7, t:1404us, D:47cm
bt:2303us, 0 @44, ofst:7, t:1403us, D:47cm
bt:2048us, 0 @45, ofst:7, t:1148us, D:38cm
bt:2109us, 1 @46, ofst:7, t:1209us, D:40cm
bt:2196us, 0 @47, ofst:7, t:1296us, D:43cm
bt:2152us, 1 @48, ofst:7, t:1252us, D:42cm
bt:2009us, 0 @49, ofst:7, t:1109us, D:37cm
bt:2094us, 0 @50, ofst:7, t:1194us, D:40cm
bt:2253us, 0 @51, ofst:7, t:1353us, D:45cm
bt:2011us, 0 @52, ofst:7, t:1111us, D:37cm
bt:2010us, 1 @53, ofst:7, t:1110us, D:37cm
bt:2207us, 1 @54, ofst:7, t:1307us, D:44cm
bt:2245us, 1 @55, ofst:7, t:1345us, D:45cm
bt:2253us, 0 @56, ofst:7, t:1353us, D:45cm
bt:2008us, 0 @57, ofst:7, t:1108us, D:37cm
bt:2046us, 1 @58, ofst:7, t:1146us, D:38cm
bt:2353us, 0 @59, ofst:7, t:1453us, D:49cm
bt:2339us, 0 @60, ofst:7, t:1439us, D:48cm
bt:2147us, 0 @61, ofst:7, t:1247us, D:42cm
bt:2108us, 1 @62, ofst:7, t:1208us, D:40cm
bt:2256us, 1 @63, ofst:7, t:1356us, D:45cm
NA xor NB: fa_21_4b_c3_6b_43_e1_c4_
Message in clear text: This is test string 1
MAC key of B is: 8c_f7_d5_19_a0_11_1b_8a_
Received MAC: 52_e4_91_e_25_7c_15_66_
Computed MAC: 7a_fd_e4_f0_61_b1_dc_6d_
```

```
Key establishment failed!!
```

Distance shortening attack – 3 additional test

3 additional test to confirm with the previous test.

```
========Ready for commitment=========
Commitment received:52_e4_91_e_25_7c_15_66_
bt:2063us, 0 @0, ofst:7, t:1163us, D:39cm
bt:2015us, 1 @1, ofst:7, t:1115us, D:37cm
bt:2259us, 0 @2, ofst:7, t:1359us, D:45cm
bt:2104us, 1 @3, ofst:7, t:1204us, D:40cm
bt:2356us, 1 @4, ofst:7, t:1456us, D:49cm
bt:2066us, 1 @5, ofst:7, t:1166us, D:39cm
bt:2069us, 1 @6, ofst:7, t:1169us, D:39cm
bt:2018us, 1 @7, ofst:7, t:1118us, D:37cm
bt:2294us, 1 @8, ofst:7, t:1394us, D:47cm
bt:2065us, 0 @9, ofst:7, t:1165us, D:39cm
bt:2150us, 0 @10, ofst:7, t:1250us, D:42cm
bt:2262us, 0 @11, ofst:7, t:1362us, D:46cm
bt:2164us, 0 @12, ofst:7, t:1264us, D:42cm
bt:2203us, 1 @13, ofst:7, t:1303us, D:44cm
bt:2263us, 0 @14, ofst:7, t:1363us, D:46cm
bt:2069us, 0 @15, ofst:7, t:1169us, D:39cm
bt:2200us, 1 @16, ofst:7, t:1300us, D:44cm
bt:2113us, 1 @17, ofst:7, t:1213us, D:41cm
bt:2004us, 0 @18, ofst:7, t:1104us, D:37cm
bt:2112us, 1 @19, ofst:7, t:1212us, D:41cm
bt:2199us, 0 @20, ofst:7, t:1299us, D:43cm
bt:2116us, 0 @21, ofst:7, t:1216us, D:41cm
bt:2044us, 1 @22, ofst:7, t:1144us, D:38cm
bt:2340us, 0 @23, ofst:7, t:1440us, D:48cm
bt:2295us, 1 @24, ofst:7, t:1395us, D:47cm
bt:2152us, 1 @25, ofst:7, t:1252us, D:42cm
bt:2065us, 0 @26, ofst:7, t:1165us, D:39cm
bt:2053us, 0 @27, ofst:7, t:1153us, D:39cm
bt:2357us, 0 @28, ofst:7, t:1457us, D:49cm
bt:2103us, 0 @29, ofst:7, t:1203us, D:40cm
bt:2132us, 1 @30, ofst:7, t:1232us, D:41cm
bt:2264us, 1 @31, ofst:7, t:1364us, D:46cm
bt:2100us, 1 @32, ofst:7, t:1200us, D:40cm
bt:2066us, 1 @33, ofst:7, t:1166us, D:39cm
```

```
bt:2052us, 0 @34, ofst:7, t:1152us, D:39cm
bt:2307us, 1 @35, ofst:7, t:1407us, D:47cm
bt:2250us, 0 @36, ofst:7, t:1350us, D:45cm
bt:2115us, 1 @37, ofst:7, t:1215us, D:41cm
bt:2008us, 1 @38, ofst:7, t:1108us, D:37cm
bt:2056us, 0 @39, ofst:7, t:1156us, D:39cm
bt:2152us, 1 @40, ofst:7, t:1252us, D:42cm
bt:2248us, 1 @41, ofst:7, t:1348us, D:45cm
bt:2163us, 0 @42, ofst:7, t:1263us, D:42cm
bt:2152us, 0 @43, ofst:7, t:1252us, D:42cm
bt:2102us, 0 @44, ofst:7, t:1202us, D:40cm
bt:2005us, 0 @45, ofst:7, t:1105us, D:37cm
bt:2213us, 1 @46, ofst:7, t:1313us, D:44cm
bt:2058us, 0 @47, ofst:7, t:1158us, D:39cm
bt:2001us, 1 @48, ofst:7, t:1101us, D:37cm
bt:2149us, 0 @49, ofst:7, t:1249us, D:42cm
bt:2298us, 0 @50, ofst:7, t:1398us, D:47cm
bt:2161us, 0 @51, ofst:7, t:1261us, D:42cm
bt:2163us, 0 @52, ofst:7, t:1263us, D:42cm
bt:2069us, 1 @53, ofst:7, t:1169us, D:39cm
bt:2214us, 1 @54, ofst:7, t:1314us, D:44cm
bt:2118us, 1 @55, ofst:7, t:1218us, D:41cm
bt:2101us, 0 @56, ofst:7, t:1201us, D:40cm
bt:2051us, 0 @57, ofst:7, t:1151us, D:39cm
bt:2016us, 1 @58, ofst:7, t:1116us, D:37cm
bt:2212us, 0 @59, ofst:7, t:1312us, D:44cm
bt:2263us, 0 @60, ofst:7, t:1363us, D:46cm
bt:2298us, 0 @61, ofst:7, t:1398us, D:47cm
bt:2017us, 1 @62, ofst:7, t:1117us, D:37cm
bt:2058us, 1 @63, ofst:7, t:1158us, D:39cm
NA xor NB: fa_21_4b_c3_6b_43_e1_c4_
Message in clear text: This is test string 1
MAC key of B is: 8c_f7_d5_19_a0_11_1b_8a_
Received MAC: 52_e4_91_e_25_7c_15_66_
Computed MAC: 7a_fd_e4_f0_61_b1_dc_6d_
Key establishment failed!!

========Ready for commitment=========
Commitment received:52_e4_91_e_25_7c_15_66_
bt:2065us, 1 @0, ofst:7, t:1165us, D:39cm
bt:2003us, 1 @1, ofst:7, t:1103us, D:37cm
bt:2161us, 0 @2, ofst:7, t:1261us, D:42cm
bt:2102us, 1 @3, ofst:7, t:1202us, D:40cm
bt:2296us, 1 @4, ofst:7, t:1396us, D:47cm
```

```
bt:2248us, 1 @5, ofst:7, t:1348us, D:45cm
bt:2252us, 1 @6, ofst:7, t:1352us, D:45cm
bt:2020us, 1 @7, ofst:7, t:1120us, D:38cm
bt:2050us, 1 @8, ofst:7, t:1150us, D:39cm
bt:2014us, 0 @9, ofst:7, t:1114us, D:37cm
bt:2151us, 0 @10, ofst:7, t:1251us, D:42cm
bt:2165us, 0 @11, ofst:7, t:1265us, D:42cm
bt:2068us, 0 @12, ofst:7, t:1168us, D:39cm
bt:2358us, 1 @13, ofst:7, t:1458us, D:49cm
bt:2019us, 0 @14, ofst:7, t:1119us, D:37cm
bt:2300us, 0 @15, ofst:7, t:1400us, D:47cm
bt:2003us, 1 @16, ofst:7, t:1103us, D:37cm
bt:2355us, 1 @17, ofst:7, t:1455us, D:49cm
bt:2297us, 0 @18, ofst:7, t:1397us, D:47cm
bt:2113us, 1 @19, ofst:7, t:1213us, D:41cm
bt:2151us, 0 @20, ofst:7, t:1251us, D:42cm
bt:2056us, 0 @21, ofst:7, t:1156us, D:39cm
bt:2056us, 1 @22, ofst:7, t:1156us, D:39cm
bt:2153us, 0 @23, ofst:7, t:1253us, D:42cm
bt:2150us, 1 @24, ofst:7, t:1250us, D:42cm
bt:2001us, 1 @25, ofst:7, t:1101us, D:37cm
bt:2356us, 0 @26, ofst:7, t:1456us, D:49cm
bt:2162us, 0 @27, ofst:7, t:1262us, D:42cm
bt:2017us, 0 @28, ofst:7, t:1117us, D:37cm
bt:2310us, 0 @29, ofst:7, t:1410us, D:47cm
bt:2116us, 1 @30, ofst:7, t:1216us, D:41cm
bt:2006us, 1 @31, ofst:7, t:1106us, D:37cm
bt:2196us, 1 @32, ofst:7, t:1296us, D:43cm
bt:2102us, 1 @33, ofst:7, t:1202us, D:40cm
bt:2298us, 0 @34, ofst:7, t:1398us, D:47cm
bt:2102us, 1 @35, ofst:7, t:1202us, D:40cm
bt:2105us, 0 @36, ofst:7, t:1205us, D:40cm
bt:1983us, 1 @37, ofst:7, t:1083us, D:36cm
bt:2350us, 1 @38, ofst:7, t:1450us, D:49cm
bt:2327us, 0 @39, ofst:7, t:1427us, D:48cm
bt:2357us, 1 @40, ofst:7, t:1457us, D:49cm
bt:2344us, 1 @41, ofst:7, t:1444us, D:48cm
bt:2164us, 0 @42, ofst:7, t:1264us, D:42cm
bt:2299us, 0 @43, ofst:7, t:1399us, D:47cm
bt:2056us, 0 @44, ofst:7, t:1156us, D:39cm
bt:2361us, 0 @45, ofst:7, t:1461us, D:49cm
bt:2361us, 1 @46, ofst:7, t:1461us, D:49cm
bt:2116us, 0 @47, ofst:7, t:1216us, D:41cm
bt:2357us, 1 @48, ofst:7, t:1457us, D:49cm
```

```
bt:2002us, 0 @49, ofst:7, t:1102us, D:37cm
bt:2248us, 0 @50, ofst:7, t:1348us, D:45cm
bt:2151us, 0 @51, ofst:7, t:1251us, D:42cm
bt:2300us, 0 @52, ofst:7, t:1400us, D:47cm
bt:2213us, 1 @53, ofst:7, t:1313us, D:44cm
bt:2298us, 1 @54, ofst:7, t:1398us, D:47cm
bt:2136us, 1 @55, ofst:7, t:1236us, D:41cm
bt:2113us, 0 @56, ofst:7, t:1213us, D:41cm
bt:2152us, 0 @57, ofst:7, t:1252us, D:42cm
bt:2250us, 1 @58, ofst:7, t:1350us, D:45cm
bt:2357us, 0 @59, ofst:7, t:1457us, D:49cm
bt:2163us, 0 @60, ofst:7, t:1263us, D:42cm
bt:2348us, 0 @61, ofst:7, t:1448us, D:48cm
bt:2262us, 1 @62, ofst:7, t:1362us, D:46cm
bt:2214us, 1 @63, ofst:7, t:1314us, D:44cm
NA xor NB: fb_21_4b_c3_6b_43_e1_c4_
Message in clear text: This is test string 1
MAC key of B is: 8d_f7_d5_19_a0_11_1b_8a_
Received MAC: 52_e4_91_e_25_7c_15_66_
Computed MAC: 41_d1_45_71_88_cb_93_d3_
Key establishment failed!!

========Ready for commitment=========
Commitment received:52_e4_91_e_25_7c_15_66_
bt:2343us, 1 @0, ofst:7, t:1443us, D:48cm
bt:2212us, 1 @1, ofst:7, t:1312us, D:44cm
bt:2348us, 0 @2, ofst:7, t:1448us, D:48cm
bt:2297us, 1 @3, ofst:7, t:1397us, D:47cm
bt:2210us, 1 @4, ofst:7, t:1310us, D:44cm
bt:2300us, 1 @5, ofst:7, t:1400us, D:47cm
bt:2166us, 1 @6, ofst:7, t:1266us, D:42cm
bt:2118us, 1 @7, ofst:7, t:1218us, D:41cm
bt:2150us, 1 @8, ofst:7, t:1250us, D:42cm
bt:2343us, 0 @9, ofst:7, t:1443us, D:48cm
bt:2016us, 0 @10, ofst:7, t:1116us, D:37cm
bt:2162us, 0 @11, ofst:7, t:1262us, D:42cm
bt:2248us, 0 @12, ofst:7, t:1348us, D:45cm
bt:2008us, 1 @13, ofst:7, t:1108us, D:37cm
bt:2348us, 0 @14, ofst:7, t:1448us, D:48cm
bt:2261us, 0 @15, ofst:7, t:1361us, D:46cm
bt:2247us, 1 @16, ofst:7, t:1347us, D:45cm
bt:2101us, 1 @17, ofst:7, t:1201us, D:40cm
bt:2101us, 0 @18, ofst:7, t:1201us, D:40cm
bt:2015us, 1 @19, ofst:7, t:1115us, D:37cm
```

```
bt:2200us, 0 @20, ofst:7, t:1300us, D:44cm
bt:2104us, 0 @21, ofst:7, t:1204us, D:40cm
bt:2347us, 1 @22, ofst:7, t:1447us, D:48cm
bt:2104us, 0 @23, ofst:7, t:1204us, D:40cm
bt:2103us, 1 @24, ofst:7, t:1203us, D:40cm
bt:2346us, 1 @25, ofst:7, t:1446us, D:48cm
bt:2112us, 0 @26, ofst:7, t:1212us, D:41cm
bt:2153us, 0 @27, ofst:7, t:1253us, D:42cm
bt:2065us, 0 @28, ofst:7, t:1165us, D:39cm
bt:2151us, 0 @29, ofst:7, t:1251us, D:42cm
bt:2019us, 1 @30, ofst:7, t:1119us, D:37cm
bt:2250us, 1 @31, ofst:7, t:1350us, D:45cm
bt:2355us, 1 @32, ofst:7, t:1455us, D:49cm
bt:2259us, 1 @33, ofst:7, t:1359us, D:45cm
bt:2260us, 0 @34, ofst:7, t:1360us, D:46cm
bt:2003us, 1 @35, ofst:7, t:1103us, D:37cm
bt:2250us, 0 @36, ofst:7, t:1350us, D:45cm
bt:2102us, 1 @37, ofst:7, t:1202us, D:40cm
bt:2116us, 1 @38, ofst:7, t:1216us, D:41cm
bt:2350us, 0 @39, ofst:7, t:1450us, D:49cm
bt:2346us, 1 @40, ofst:7, t:1446us, D:48cm
bt:2258us, 1 @41, ofst:7, t:1358us, D:45cm
bt:2115us, 0 @42, ofst:7, t:1215us, D:41cm
bt:2310us, 0 @43, ofst:7, t:1410us, D:47cm
bt:2102us, 0 @44, ofst:7, t:1202us, D:40cm
bt:2260us, 0 @45, ofst:7, t:1360us, D:46cm
bt:2162us, 1 @46, ofst:7, t:1262us, D:42cm
bt:2216us, 0 @47, ofst:7, t:1316us, D:44cm
bt:2307us, 1 @48, ofst:7, t:1407us, D:47cm
bt:2161us, 0 @49, ofst:7, t:1261us, D:42cm
bt:2359us, 0 @50, ofst:7, t:1459us, D:49cm
bt:2260us, 0 @51, ofst:7, t:1360us, D:46cm
bt:2359us, 0 @52, ofst:7, t:1459us, D:49cm
bt:2298us, 1 @53, ofst:7, t:1398us, D:47cm
bt:2105us, 1 @54, ofst:7, t:1205us, D:40cm
bt:2067us, 1 @55, ofst:7, t:1167us, D:39cm
bt:2354us, 0 @56, ofst:7, t:1454us, D:49cm
bt:2101us, 0 @57, ofst:7, t:1201us, D:40cm
bt:2250us, 1 @58, ofst:7, t:1350us, D:45cm
bt:2200us, 0 @59, ofst:7, t:1300us, D:44cm
bt:2015us, 0 @60, ofst:7, t:1115us, D:37cm
bt:2202us, 0 @61, ofst:7, t:1302us, D:44cm
bt:2164us, 1 @62, ofst:7, t:1264us, D:42cm
bt:2054us, 1 @63, ofst:7, t:1154us, D:39cm
```

```
NA xor NB: fb_21_4b_c3_6b_43_e1_c4_
Message in clear text: This is test string 1
MAC key of B is: 8d_f7_d5_19_a0_11_1b_8a_
Received MAC: 52_e4_91_e_25_7c_15_66_
Computed MAC: 41_d1_45_71_88_cb_93_d3_
Key establishment failed!!
```

Appendix F

MATLAB program for Berkeley Echo

```
% LocationVeri.m
% This MATLAB program is to simulate behavior of
% location verification using Radio and Ultrasound
% communication.
% The protocol is based on Berkeley Echo protocol
%
% Ref:
%   Naveen Sastry, Umesh Shankar and David Wagner.
%   Secure Verification of Location Claims.
%   Proceedings of the Workshop on Wireless Security,
%   Proceedings of the 2003 ACM Workshop on Wireless Security.
%
%@version: ver1.0
%@Author: Feng Kai
%@Date:   May 3th, 2006
%@Place:  Building 322,
%     Informatics and Mathematical Modelling (IMM)
%         Technical Unversity of Denmark (DTU)


clear;
close all;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parameters for situation to be simulated
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%///// Please change the following to simulate\\\\\\
%------------------ Role parameters-----------------
% Location of Prover (meters)
dp = 501;
% Location of Prover claim to be (meters)
dl = 176;
```

```
% Processing time delay Prover claimed
tpp = 0.01;
% Verifier range (meters)
R = 500;
% -------------- Protocol parameters --------------
% Nonce size, bits
N = 16;
% Radio communication bandwidth (bit rate), bits/s
b_r = 48000;
% Ultrasound communication bandwidth (bit rate), bits/s
b_s = 17;
% Radio bit rate, sec/bit
v_r = 1/b_r;
% Ultrasound bit rate, sec/bit
v_s = 1/b_s;
% Time to transmit a radio package, second
t_r = N / b_r;
% Time to transmit a Ultrasound package, second
t_s = N / b_s;

%////////////////// Constants \\\\\\\\\\\\\\\\\\\\\\
% ---------------- Natual constant ----------------
% Speed of light (radio) m/s
C = 3e8;
% Speed of sound m/s
S = 340;
% ---------------- Appearance ---------------------
%  Verifier
% color of verifier radius
VERIFIER_COLOR = 'r';
% color of verifier circle
VERIFIER_RANGE = 'r';
% color of ROA circle with processing time
VERIFIER_ROA_P = 'g-.';
% color of ROA radius
VERIFIER_R_P = 'g';
% color of ROA circle with processing time + transmission time
VERIFIER_ROA_PT = 'b-.';
% color of ROA radius
VERIFIER_R_PT = 'b';
% verifier, green small circle
VERIFIER = 'go';
% Prover
% prover, red pentagram
PROVER = 'rp';
% prover's claim , diamond
PROVER_CLAIM = 'd';

% EPS figure position
epsFigurePos = [0.25 0.25 12 12];

%///////////// Start of simulation \\\\\\\\\\\\\\\
%//////// Please do not change the following \\\\\\
% ---------- Calculate current ROA ----------------
% Location of verifier (meters)
dv = 0;
% Location to be verified
dvl = dl - dv;
% True Distance between prover and verifier
Dpv = dp - dv;
% Region of Acceptance (ROA) due to processing time
ROA_P = R - S*tpp;
% ROA due to processing time + transmission time
ROA_ALL = ROA_P - (t_r + t_s)*S;
% ---------- Evaluate ROA 01 -----------------
% verifier'r radius under evaluation
% 0 ~ 2km
eva_R01 = 0:0.01:2000;
eva_ROA_ALL01 = eva_R01 - S*tpp - (t_r + t_s)*S;
% ---------- Evaluate ROA 02 -----------------
% ROA as a function of (R,n)
% the number of bits for Key
% N = 1 ~ 64 bits
eva_N02 = 1:64;
% R = 0 ~ 4km
eva_r = 1:4000;
eva_r2D = ones(length(eva_N02),1)*eva_r;
eva_ps = S*tpp + (eva_N02/b_r + eva_N02/b_s)*S;
eva_ps = (eva_ps') * ones(1,length(eva_r));
eva_ROA_ALL02(:,eva_r) = eva_r2D(:,eva_r) - eva_ps(:,eva_r);

for i=1:b_s
    eva_r2D_03 = ones(length(eva_N02),1)*eva_r;
    eva_ps_03 = S*tpp + (eva_N02/b_r + eva_N02/i)*S;
    eva_ps_03 = (eva_ps_03') * ones(1,length(eva_r));
    eva_ROA_ALL03(:,eva_r) = eva_r2D_03(:,eva_r) - eva_ps_03(:,eva_r);
    eva_ROA_ALL(:,:,i) = eva_ROA_ALL03(:,:);
    clear eva_ROA_ALL03;
%x01(:,:,2) = eva_ROA_ALL03(:,:);
end
% ---------- Evaluate time  -----------------
% evaluate time parameters in all communications
% Radio:        verifer ---->  prover
% time for radio to travel from verifier to prover
tvp_r = Dpv / C;
% time to send a entire radio package
t_r;
% total radio sending time
tvp_pack_r = tvp_r + t_r;
% Radio:        verifer ---->  l
% time for radio to travel from verifier to prover
tvl_r = dvl / C;
% time to send a entire radio package
t_r;
% total radio sending time
tvl_pack_r = tvl_r + t_r;
% US:          l -------> verifier
% time for US to travel from Distance-claimed to verifier
tvl_s = dvl / S;
% time to send the entire US package
t_s;
% total US sending time
tvl_pack_s = tvl_s + t_s;
% US:        prover ------> verifier
% time for US to travel from prover to verifier
tvp_s = Dpv / S;
% time to send the entire US package
t_s;
% total US sending time
tvp_pack_s = tvp_s + t_s;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot and visualization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%--------- Calculation of current setup ------------
figure(1)
title('Situation visualization');
hold on;
%///////////// Verifier's location \\\\\\\\\\\\\\\
% plot verifier location
plot(dv,0,VERIFIER);
text(dv,5,'Verifier');
% plot verifier range
ang = 0:pi/100:2*pi;
plot(R*sin(ang),R*cos(ang),VERIFIER_RANGE);
% plot verifier radius
show_ang = 3/4*pi;
x = R*cos(show_ang);
y = R*sin(show_ang);
text(x/2,y/2,'R');
x = 0:0.1*x/abs(x):x;
y = 0:0.1*y/abs(y):y;
plot(x,y,VERIFIER_COLOR);

% plot verifier's ROA due to processing time
if ROA_P > 0
      ang = 0:pi/100:2*pi;
      plot(ROA_P*sin(ang),ROA_P*cos(ang),VERIFIER_ROA_P);
      % plot verifier radius
      show_ang = 1/4*pi;
      x = ROA_P*cos(show_ang);
      y = ROA_P*sin(show_ang);
      text(x/2,y/2,'ROA(P)');
      x = 0:0.1*x/abs(x):x;
      y = 0:0.1*y/abs(y):y;
      plot(x,y,VERIFIER_R_P);
  end
% plot verifier's ROA due to processing time + transmission time
if ROA_ALL>0
      ang = 0:pi/100:2*pi;
      plot(ROA_ALL*sin(ang),ROA_ALL*cos(ang),VERIFIER_ROA_PT);
      % plot verifier radius
      show_ang = 7/4*pi;
      x = ROA_ALL*cos(show_ang);
      y = ROA_ALL*sin(show_ang);
      text(x/2,y/2,'ROA(PT)');
      x = 0:0.1*x/abs(x):x;
      y = 0:0.1*y/abs(y):y;
      plot(x,y,VERIFIER_R_PT);
  end
%///////////// Prover's location \\\\\\\\\\\\\\\
% plot prover true location
plot(dp,0,PROVER);
text(dp,5,'Prover');
% plot prover's claimed location
plot(dl,0,PROVER_CLAIM);
text(dl,5,'claimed');
grid on
axis equal
hold off;
set(gcf, 'PaperPositionMode', 'manual');
set(gcf, 'PaperUnits', 'centimeters');
set(gcf, 'PaperPosition', epsFigurePos);
print -depsc2 'figure01.eps'
% ---------- Evaluate ROA 01 -----------------
% plot verifier's radius R vs. ROA
figure(2);
hold on;
grid on
axis equal
% plot R vs. ROA
plot(eva_R01,eva_ROA_ALL01);
title(sprintf('Verifier R vs ROA, N = %d',N));
xlabel('Verifier Radius');
```

```
ylabel('Verifiable ROA');
% markers
for i = 0:100:max(eva_ROA_ALL01)
    % the R axis
    temp = round(i + S*tpp + (t_r + t_s)*S);
    % the offset to shift marker up
    temp_offsetY = abs(min(eva_ROA_ALL01));
    plot(temp,i,'*');
    text(temp+20, i, sprintf('(%d,%d)',temp,i));
end;
hold off;
set(gcf, 'PaperPositionMode', 'manual');
set(gcf, 'PaperUnits', 'centimeters');
set(gcf, 'PaperPosition', epsFigurePos);
print -depsc2 'figure02.eps'
% ---------- Evaluate ROA 02 -----------------
% plot ROA vs. (R,N)
figure(3);
% viewing window of R
viewR_wnd = 801:1800;
viewN_wnd = eva_N02;
surf(viewR_wnd,viewN_wnd,eva_ROA_ALL02(viewN_wnd,viewR_wnd));
%text(min(viewR_wnd)+20,max(viewN_wnd),min(min(eva_ROA_ALL02(...
%viewN_wnd,viewR_wnd,i)))+200,sprintf('BitRate=%d bps',17));
for i=5:4:b_s
surf(viewR_wnd,viewN_wnd,eva_ROA_ALL(viewN_wnd,viewR_wnd,i));
text(min(viewR_wnd)+20,max(viewN_wnd)+10,min(min(eva_ROA_ALL(...
viewN_wnd,viewR_wnd,i)))+200,sprintf('BitRate=%d bps',i));
hold on
end
title('ROA as a function of R and N');
xlabel('R verifier radius(meter)');
ylabel('N nonce length(bits)');
hold off;
set(gcf, 'PaperPositionMode', 'manual');
set(gcf, 'PaperUnits', 'centimeters');
set(gcf, 'PaperPosition', epsFigurePos);
print -depsc2 'figure03.eps'
% ---------- Evaluate time parameter ------------
% plot time sequence diagram of communications
figure;
title('Time sequence diagram');
set(gca,'YDir','reverse');
xlabel('Distance (meters)');
ylabel('Time (1000 x microseconds)');
DispUnitScaler = 1000;
hold on;
grid on;
%Plot Verifier and her time axis
X_POS = 1; Y_POS = 2;
VerifierPos = [0,0];
TimeLine_Length = round((tvp_pack_r + tvl_pack_s)*DispUnitScaler);
plot([VerifierPos(X_POS) VerifierPos(X_POS)],[VerifierPos(X_POS)...
        TimeLine_Length]);
text(VerifierPos(X_POS)-10,VerifierPos(Y_POS)-50,'A');
%Plot Prover and his time axis
ProverPos = [Dpv,0];
plot([ProverPos(X_POS),ProverPos(X_POS)] ,[0 TimeLine_Length]);
text(ProverPos(X_POS)+10,-50,'B');
%Plot Claimed position and his time axis
ClaimedPos = [dvl,0];
plot([ClaimedPos(X_POS),ClaimedPos(X_POS)] ,[0 TimeLine_Length],'-.');
text(ClaimedPos(X_POS)+10,-50,'L');

% Plot radio message sent from verifier to prover
% Plot only first bit and last bit
% origin
ori = [0,0];
src = ori;
tar = [ProverPos(X_POS),tvp_r*DispUnitScaler];
plot([src(X_POS) tar(X_POS)], [src(Y_POS) tar(Y_POS)],'g>-');
for idx = 1:(N-1)
     src = src + [0,v_r*DispUnitScaler];
     tar = tar + [0,v_r*DispUnitScaler];
     plot([src(X_POS) tar(X_POS)], [src(Y_POS) tar(Y_POS)],'g>-');
end
%text(tar(X_POS),tar(Y_POS),sprintf('(%d,%d)',tar(X_POS),round(tar(Y_POS))));
% Plot ultrasound message from prover
tar = [0,tvl_s*DispUnitScaler];
src = [ProverPos(X_POS), (tvl_s+tvl_pack_r-tvp_s)*DispUnitScaler];
TimeLine_Cheater = src(Y_POS)-100;
plot([src(X_POS) tar(X_POS)], [src(Y_POS) tar(Y_POS)],'r<-');
text(src(X_POS),src(Y_POS),sprintf('(%d)',round(src(Y_POS))));
text(tar(X_POS),tar(Y_POS),sprintf('(%d)',round(tar(Y_POS))));
for idx = 1:(N-1),
    src = src + [0,v_s*DispUnitScaler];
    tar = tar + [0,v_s*DispUnitScaler];
    plot([src(X_POS) tar(X_POS)], [src(Y_POS) tar(Y_POS)],'r<-');
    text(src(X_POS),src(Y_POS),sprintf('(%d)',round(src(Y_POS))));
    text(tar(X_POS),tar(Y_POS),sprintf('(%d)',round(tar(Y_POS))));
end
% Axis
axis([VerifierPos(X_POS)-20 ProverPos(X_POS)+20 TimeLine_Cheater...
        TimeLine_Length+100]);
hold off;
set(gcf, 'PaperPositionMode', 'manual');
set(gcf, 'PaperUnits', 'centimeters');
set(gcf, 'PaperPosition', epsFigurePos);
print -depsc2 'figure04.eps'

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Show result
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
disp (' ');
disp('%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%');
disp('%%%%%             The result summary             %%%%%');
disp('%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%');
disp('--------------  General result ----------------------');
disp('True Distance betwwen Prover and verifier: (meters)');
disp(Dpv);
disp('Prover claimed distance: (meters)');
disp(dvl);
disp('Verifier range: (meters)');
disp(R);
disp('Verifier ROA due to processing time: (meters)');
disp(ROA_P);
if ROA_P < 0
   disp('Protocol not engaged');
end
disp('Verifier ROA due to processing time + transmission time: (meters)');
disp(ROA_ALL);
if ROA_ALL < 0
    disp('Protocol not engaged');
end

disp('----------  Timing result for current setup -------------');
disp('Time for radio to travel from verifier to prover: (milisecond)');
disp(tvp_r * 1000);
disp('Time for entire package to arrive at prover: (milisecond)');
disp(t_r * 1000);
disp('Total time from sending radio package till received: (milisecond)');
disp(tvp_pack_r * 1000);

disp('Time for US to travel from distance-claimed to verifier: (milisecond)');
disp(tvl_s * 1000);
disp('Time for entire US package to arrive at verifier: (milisecond)');
disp(t_s * 1000);
disp('Total time from sending US package till received: (milisecond)');
disp(tvl_pack_s * 1000);

disp('Time for US to travel from prover to verifier: (milisecond)');
disp(tvp_s * 1000);
disp('Time for entire US package to arrive at verifier: (milisecond)');
disp(t_s * 1000);
disp('Total time from sending US package till received: (milisecond)');
disp(tvp_pack_s * 1000);
```

Appendix G

NesC source code

G.1 US transceiver

usTransceiverM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     usTransceiverM.nc, 2006/02/27 22:00:00
 */

module usTransceiverM {
  provides {
    interface usTransceiver;
  }
  uses {
    interface Leds;
    interface UltrasoundControl;

  }
}
implementation {
```

```
/*
 *  variables for transmitter
 */
//Data to be transmitted
uint32_t Tx_Data;
// Data has been sent
bool Tx_dataSent;
// New data to be sent is ready
bool Tx_dataReady;
// Transmitter current state
uint8_t Tx_currentState;


/*
 *  variables for receiver
 */
// The whole data byte received
uint32_t Rx_data;
// New data byte has been received
bool Rx_dataReceived;
// Current bit received
bool Rx_CurrentBit;
// Receiver current state
uint8_t Rx_currentState;
// us receiver enable, to avoid multiple INT within the same bit
bool Rx_enable;



/*
 *  variables for application
 */
// Data received
uint16_t Rx_buffer;
// Parity check of data received
bool Data_Healthy;
// First data bit arrival time
uint16_t arrivalTime;

/*
 *  Configuration variables
 */
bool Sender;
uint8_t us_Gain;


  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t usTransceiver.init() {

    atomic {
     // initialize Tx variable
             Tx_Data = 0;
             Tx_dataReady = 0;
             Tx_dataSent = 1;
             Tx_currentState = US_OFFLINE;

     // initialize Rx variable
             Rx_enable = 1;
             Rx_data = 0;
             Rx_CurrentBit = 0;
```

```
            Rx_dataReceived = 0;
            Rx_currentState = US_OFFLINE;
    }

    //Default as receiver
    call usTransceiver.setRole(US_RECEIVER);
    //Default gain
    call usTransceiver.setGain(127);
    //Enable ultrasound receiver
    TOSH_CLR_US_IN_EN_PIN();
// Init LEDs
    call Leds.init();
    return  SUCCESS;
  }


  /**
   * Start things up. Start US detector if configured as receiver.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t usTransceiver.start() {

    if (!Sender)
    call UltrasoundControl.StartDetector(US_DETECT_TIMEOUT);

    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   * Stop US detector
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t usTransceiver.stop() {
    call UltrasoundControl.StopDetector();
    return SUCCESS;
  }


  /**
   * Set role of transceiver
   * @param tx options are RF_SENDER or RF_RECEIVER
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t usTransceiver.setRole(bool tx) {
     atomic Sender = tx;
     // Stop detector if it is a sender
     if (Sender)
        call UltrasoundControl.StopDetector();

     return SUCCESS;
  }

  /**
   * Set gain
   * @param g 0~255, typically 127
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t usTransceiver.setGain(uint8_t g) {
   us_Gain = g;
      call UltrasoundControl.SetGain(us_Gain);
   return SUCCESS;
  }
```

```
/**
 * Send one word of data
 * @param data Data to be sent, type: uint32_t
 * @return Always returns <code>SUCCESS</code>
 **/
async command result_t usTransceiver.SendWord(uint32_t data) {
     uint32_t i = 0;
     uint32_t mask = 1;
     bool parity = 0;
     uint32_t databuf = 0;

       /* Sending detector to keep communication data rate */
 call UltrasoundControl.StartDetector(US_DETECT_TIMEOUT);
 /* Sending a start bit to sync channel*/
 call UltrasoundControl.SendPulse();

     /**
      * Compute parity bit  of data if enabled
      *
      **/
     if (US_PARITY_CHECK) {
         for (i=0;i<US_DATABITS;i++) {
             parity ^= ((data & mask)?1:0);
             mask *= 2;
         }
     /**
      * parity format: even/odd
      *
      **/
         if (US_PARITYFORMAT == US_ODD_PARITY)
              parity = ((parity)?0:1);
         else
              parity = parity?1:0;

 //add parity into data word
         databuf = data | (0x100 * parity);
         /**
          * Put stop bits into data word, by clearing bit 15~9
          **/
          databuf = databuf & 0x1FF;
     }
     else {
        databuf = data;
         /**
          * Put stop bits into data word, by clearing bit 15~8
          **/
        databuf = databuf & 0xFF;
    }

      atomic{
            /**
             * indicate new Tx_Data ready to send
             **/
              Tx_dataReady = 1;
              Tx_Data = databuf;
            /**
             * data has not been sent
             **/
              Tx_dataSent = 0;
            /**
             * transmitter current state offline
             **/
             Tx_currentState = US_TRANSMIT;
      }
     return SUCCESS;
 }

 command bool usTransceiver.isLastDataSent() {
    bool r;
atomic r = Tx_dataSent;
    return r;
 }

 command bool usTransceiver.isNewDataReady() {
   return Rx_dataReceived;
 }

 command uint32_t usTransceiver.ReceiveWord() {

  atomic Rx_dataReceived = 0;

   return (Rx_data);
 }

 command bool usTransceiver.Rx_dataHealthy() {
     uint32_t i = 0;
     uint32_t mask = 1;
     bool parity = 0;
     uint32_t data;
     atomic data = Rx_buffer;

      /**
       * Compute parity bit if enabled and put into data word
       *
       **/
      if (US_PARITY_CHECK) {
          for (i=0;i<US_DATABITS;i++) {
              parity ^= ((data & mask)?1:0);
              mask *= 2;
          }

          /**
           * Determine parity bit to be added according to pairty format
           *
           **/
           if (US_PARITYFORMAT == US_ODD_PARITY)
                parity = parity?0:1;
            else
                parity = parity?1:0;

       }
       data = data>>US_DATABITS;
      return ( (data&1) == parity);
 }

 // this event should be implemented in the user of sender
 default event  result_t usTransceiver.SendDone(){
  return SUCCESS;
 }



 // this event should be implemented in the user of Receiver
 default event uint32_t usTransceiver.DataReceived(uint32_t usData , uint16_t tStamp){
  return usData;
 }
```

```
    task void reportPD(){
 printf("Signal detected,");
}

  task void usMsgDetected() {
   uint16_t t;
   atomic t = arrivalTime;
       signal usTransceiver.msgDetected(t);
  }

  /**
   * This event is signaled when a Ultrasound pulse is detected.
   *
   * @param timer the time spent since the start of detector, of type uint16_t
   * @return Always returns <code>SUCCESS</code>
   **/
      async event result_t UltrasoundControl.PulseDetected(uint16_t timer)
   {
    bool en;
    atomic en = Rx_enable;
      if ((!Sender) && (en))
           {
          atomic {
            en = 0;
            Rx_enable = en;
            Rx_CurrentBit = 1;
            arrivalTime = timer;

             if (Rx_currentState == US_OFFLINE) {
                    /*
                * The first pulse of the byte
                */
                      post usMsgDetected();

                      call UltrasoundControl.StopDetector();
                      Rx_CurrentBit = 0;
               Rx_dataReceived = 0;
               Rx_currentState = US_ONLINE;
               call UltrasoundControl.StartDetector(US_DETECT_TIMEOUT-US_SETUPTIME);
             }
          }
      }
      return SUCCESS;
   }

   task void reportonline(){
      printf("Online\r\n");
   }

   task void reportrx(){
      printf("receive\r\n");
   }

   task void reportstop(){
       printf("Stop\r\n");
   }

   task void reportz(){
       printf("0");
   }

   task void reporto(){
       printf("1");
   }
   task void usSendDone() {
        signal usTransceiver.SendDone();
   }

   task void usDataReceived() {
    uint16_t r;
    atomic r = Rx_buffer;
        signal usTransceiver.DataReceived(r,0);
   }

  /**
   * This event is signaled when a Ultrasound detector time out.
   *
   *
   * @return Always returns <code>SUCCESS</code>
   **/
   async event result_t UltrasoundControl.DetectorTimeout()
  {
   static uint32_t tx_d = 0;
   // Index for current bit to be sent
    norace static uint32_t Tx_Index = 0;
    // Current sending bit mask
   norace static uint32_t Tx_BitMask = 1;

   uint32_t rx_d;
      // Weight of Current bit received
      //  (data[Rx_Index])
      // 2                  = Rx_CurrentBit * Rx_BitMask
      norace static uint32_t Rx_BitMask = 1;
      // Current receiving bit index
      norace static uint32_t Rx_Index = 0;

      uint8_t state;
  uint32_t tempbit;

   if (!Sender) {
          /*
           * Start detector for the next receiving bit
           */

          atomic state = Rx_currentState;

       switch (state) {
        case US_OFFLINE:
            /*
             *  No signal detected
             */
             break;
        case US_ONLINE:
            /*
             *  First pulse detected, byte synchronized
             */
             atomic {
                          state = US_RECEIVE;
        Rx_CurrentBit = 0;
        Rx_enable = 1;
                   }
             break;
        case US_RECEIVE:

             atomic {
                          rx_d = Rx_CurrentBit;
                          Rx_CurrentBit = 0;
                          Rx_enable = 1;
                   }
                  Rx_Index++;
               /*
 * online and A pulse is received
              * or time out is found
 */
              if  (Rx_Index <= (US_DATABITS+US_STOPBITS+US_PARITY_CHECK)){ //
                       /*
                        * a Rx_data Rx_BitMask of 1
                        */
                        if  (rx_d == 1) {
                              atomic Rx_data = Rx_data | Rx_BitMask;
                  }
else
     ;
                        /*
                         *  no pulse received
                         *  else
                         */
                         Rx_BitMask*=2;
             }
                  else
                  /*
                   *  final stop bit
                   */

                   {
                        atomic {
                                 Rx_buffer = Rx_data&0x1ff;
                                 Rx_data = 0;
                                 Rx_dataReceived = 1;
                                 state = US_OFFLINE;
                                 Rx_Index = 0;
                                 Rx_BitMask = 1;
                                 post usDataReceived();
                               }
                   }
             break;
       /* End switch*/
          }
          atomic Rx_currentState = state;
          call UltrasoundControl.StartDetector(US_DETECT_TIMEOUT);
          //////////////////////////////////////////////////////////////////
      /* End Receiver */
      }
      else {
        /*
         * Sender
         */
         /*
          * restart timer to send next bit
          */
          atomic state = Tx_currentState;

          call UltrasoundControl.StartDetector(US_DETECT_TIMEOUT);
         switch (state) {
          case US_OFFLINE:
               break;
          case US_ONLINE:
               state = US_TRANSMIT;
                break;
            case US_TRANSMIT:
                 /*
                  * ONLINE, Sending Tx_BitMask by Tx_BitMask,
                  * from Tx_BitMask 1 to Tx_BitMask 8
                  */
                 atomic Tx_Index++;
                 if (Tx_Index <= (US_DATABITS + US_STOPBITS + US_PARITY_CHECK) ){
                       /*
                        * send Tx_Data
                        */
atomic tempbit = Tx_Data & Tx_BitMask;
                        /****  send a pulse if '1' ****/
                        if (tempbit>=1)
                           {
                               call UltrasoundControl.SendPulse();
                           }
                        /****  send nothing if '0'   ****/

                        /*
                         *   shift to next Tx_BitMask for examination
                         */
                            Tx_BitMask *= 2;
                 }
                  else
                      /*
                       * ONLINE, Send done
                       */
                      {
                            atomic {
                                     Tx_dataReady = 0;
                                     Tx_dataSent = 1;
                                     state = US_OFFLINE;
                                     tx_d = 0;
                                     Tx_Index = 0;
                                     Tx_BitMask = 1;
                            }
                            post usSendDone();
                      }
            }
       atomic Tx_currentState = state;
       /*
        *  End sender
        */
       }
      return SUCCESS;
  }

  /**
   * Get the current time stamp.
   *
   * @return uint32_t time stamp
   **/
  command uint16_t usTransceiver.getTimeStamp() {
       uint16_t t;
       atomic t = 0;
    return  t;
  }

}
```

usTransceiverC.nc

```
 /**
 * @author Feng Kai s030656@student.dtu.dk
 **/
```

```
includes usTransceiver;

configuration usTransceiverC{
  provides interface usTransceiver;
}
implementation {
   components  usTransceiverM,UltrasoundControlM,LedsC;

  usTransceiver = usTransceiverM.usTransceiver;

  usTransceiverM.UltrasoundControl -> UltrasoundControlM;
  usTransceiverM.Leds -> LedsC.Leds;

}
```

usTransceiver.nc

```
/**
 *
 * @author s030656@student.dtu.dk
 **/


interface usTransceiver {

  command result_t init();

  command result_t start();

  command result_t stop();

  command result_t setRole(bool tx);

  command result_t setGain(uint8_t g);

  /*
   * Send
   */
  async command result_t SendWord(uint32_t d);

  command result_t isLastDataSent();

  event result_t SendDone();

  /*
   * Receive
   */
  event uint16_t msgDetected(uint16_t timeStamp);

  event uint32_t DataReceived(uint32_t Rx_data,uint16_t timeStamp);

  command bool isNewDataReady();

  command uint32_t ReceiveWord();

  command bool Rx_dataHealthy();

  /*
   *  Time stamping services
   */
  command uint16_t getTimeStamp();

}
```

usTranscever.h

```
/*
 * definition.h
 * Feng Kai<s030656@student.dtu.dk>
 *
 */


#ifndef _USTRANSCEIVER_H
#define _USTRANSCEIVER_H

#include <stdio.h>
#include <avr/eeprom.h>


//Configuration
#define US_RECEIVER 0
#define US_SENDER 1

#define US_GAIN_NORMAL 127
// The speed of sound used for distance calculation
#define SPEED_OF_SOUND 342

#define US_OFFLINE 0
#define US_ONLINE  1
#define US_RECEIVE 2
#define US_TRANSMIT 3

// UltraSound transmission bit rate
// unit: ms/bit

#define US_DETECT_TIMEOUT (unsigned int)(60000)

// UltraSound transmission data width
#define US_DATABITS 8
// Define number of Stop bit
#define US_STOPBITS 2
// Define parity bit
#define US_PARITY_CHECK 1
// Define parity check, even
#define US_EVEN_PARITY 0
#define US_ODD_PARITY 1
#define US_PARITYFORMAT US_ODD_PARITY
// US receiver setup time in microsecond (us)
#define US_SETUPTIME 5000

#endif
```

G.2 US sender and receiver

UseTxM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:           UseTxM.nc, 2006/02/27
 */

module UseTxM{
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface usTransceiver;
    interface Timer;
    interface RS232;
  }
}
implementation {

// data to be sent
uint8_t data;

  /**
   * Initialize ultrasound transceiver.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    call usTransceiver.init();
    call usTransceiver.setRole(US_SENDER);
    call usTransceiver.setGain(US_GAIN_NORMAL);
    data = 0x0;
    call RS232.init();
    return SUCCESS;
  }


  /**
   * Start timer and usTransceiver.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    call Timer.start(TIMER_REPEAT, 1000);
    call usTransceiver.start();
    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    call usTransceiver.stop();
    return SUCCESS;
  }

  /**
   * This event is signalled when US packet is sent.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t usTransceiver.SendDone(){
    return SUCCESS;
  }

  /**
   * This event is signalled when a message is detected.
   * It detect the first presents of the US bit.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint16_t usTransceiver.msgDetected(uint16_t timeStamp){

   return timeStamp;
  }

  /**
   * This event is signalled when US packet is received.
   *
   * @return Always returns US Data
   **/
  event uint32_t usTransceiver.DataReceived(uint32_t usData , uint16_t timeStamp){
    return usData;
  }

  /**
   * Periodically send US data.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
   event result_t Timer.fired()
  {

     if (call usTransceiver.isLastDataSent()) {
       // Send US data
            call usTransceiver.SendWord(data);
       // light up LED
      switch(data&0xFF) {
        case 0x30: call Leds.yellowOff();call Leds.greenOff();call Leds.redOff();break;
        case 0x31: call Leds.yellowOn();call Leds.greenOff();call Leds.redOff();break;
        case 0x32: call Leds.yellowOff();call Leds.greenOn();call Leds.redOff();break;
        case 0x33: call Leds.yellowOn();call Leds.greenOn();call Leds.redOff();break;
    }
// incrementing data, data should be between 0x30 and 0x33
            data++;
      }
    return SUCCESS;
  }

  /**
   * This event is signalled when PC send a string.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }
}
```

UseTx.nc

```
/**
 * This application is a ultrasound sender.
 *
 * @author s030656@student.dtu.dk
 **/
includes usTransceiver;
configuration UseTx {
}
implementation {
   // Components used for UseTxM
   components Main, TimerC, LedsC, UseTxM;
   // Components used for usTransceiverC
   components usTransceiverC;
   // Components RS232
   components RS232C;

  // Wire of component UseTxM
  Main.StdControl -> UseTxM.StdControl;
  Main.StdControl -> TimerC.StdControl;
  UseTxM.Leds -> LedsC.Leds;
  UseTxM.Timer -> TimerC.Timer[unique("Timer")];

  // Wire of components usTransceiverC
  UseTxM.usTransceiver -> usTransceiverC.usTransceiver ;

  // RS232
  UseTxM.RS232 -> RS232C.RS232;
}
```

UseRxM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     UseRxM.nc, 2006/02/27
 */


module UseRxM{
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface usTransceiver;
    interface Timer;
    interface RS232;
  }
}
implementation {

uint16_t data;
bool sent;
uint32_t time;

  /**
   * Initialize ultrasound transceiver and RS232 communication.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    call usTransceiver.init();
    call usTransceiver.setRole(US_RECEIVER);
    call usTransceiver.setGain(US_GAIN_NORMAL);
    data = 0;
    sent = 1;
    call RS232.init();
    return SUCCESS;
  }


  /**
   * Start timer and ultrasound receiver.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    call Timer.start(TIMER_REPEAT, 1000);
    call usTransceiver.start();
    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    call usTransceiver.stop();
    return SUCCESS;
  }

  /**
   * This event is signalled when a string is received from PC.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }

  /**
   * This event is signaled when US packet is sent.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t usTransceiver.SendDone(){
    return SUCCESS;
  }

  event uint16_t usTransceiver.msgDetected(uint16_t timeStamp){
   return timeStamp;
  }

  task void reportData(){
      printf("data received: %x\r\n",data);
  }

  task void dataerr(){
     call Leds.redToggle();
     printf("Rcv Wrong: %x\r\n",data);
  }

  /**
   * This event is signalled when Data is recieved from US channel.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint32_t usTransceiver.DataReceived(uint32_t Rx_data , uint16_t timeStamp){
  atomic data = Rx_data;
      post reportData();

    if (call usTransceiver.Rx_dataHealthy()) {
                switch(Rx_data&0xFF) {
                      case 0x30: call Leds.yellowOff();call Leds.greenOff();call Leds.redOff();break;
                      case 0x31: call Leds.yellowOn();call Leds.greenOff();call Leds.redOff();break;
                      case 0x32: call Leds.yellowOff();call Leds.greenOn();call Leds.redOff();break;
                      case 0x33: call Leds.yellowOn();call Leds.greenOn();call Leds.redOff();break;
                     }
       }
       else
         {
            post dataerr();
         }


    return Rx_data;
  }

  /**
   * This is event is signalled when timer fired. Not used in this application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
   event result_t Timer.fired()
  {
     return SUCCESS;
  }

}
```

UseRx.nc

```
/**
 * This application is a ultrasound receiver.
 *
 * @author s030656@student.dtu.dk
 **/
includes usTransceiver;
configuration UseRx {
}
implementation {
   components Main, UseRxM, LedsC;
   components TimerC;
   components RS232C;


   components usTransceiverC;
   components TimerC as usTimerC;

  Main.StdControl -> UseRxM.StdControl;
  Main.StdControl -> TimerC.StdControl;
  UseRxM.Leds -> LedsC.Leds;
  UseRxM.Timer -> TimerC.Timer[unique("Timer")];

  UseRxM.usTransceiver -> usTransceiverC.usTransceiver ;

  UseRxM.RS232 -> RS232C.RS232;

}
```

G.3 RF transceiver

rfTransceiverM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     rfTransceiverM.nc, 2006/03/18 23:00:00
 */


module rfTransceiverM {
  provides {
   interface rfTransceiver;

   }
  uses {
    interface Leds;
    interface StdControl as RadioControl;
    interface BareSendMsg as RadioSend;
    interface ReceiveMsg as RadioReceive;


    async command uint8_t GetRxBitOffset();
  }
```

```
}
implementation {

  /**
   * The role of this transceiver
   * Can be configured as transmitter <code> RF_SENDER</code>
   *             or receiver    <code> RF_RECEIVER</code>
   **/
  bool bRole;

  //Busy flag
  bool senderBusy;


   /**
   * Set the role of this transceiver
   * @param role The role of the transceiver. Can be configured
   *             as transmitter <code> RF_SENDER</code>
   *             or receiver <code> RF_RECEIVER</code>
   *             The transceiver need to be stoped first,
   *             change the role by calling this command
   *             and restarted again by calling start command.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t rfTransceiver.setRole(bool role) {
    bRole = role;
    return SUCCESS;
  }


  /**
   * Initialize the component, and set default role as receiver.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t rfTransceiver.init() {
    //call Leds.init();
    // Init radio
    call RadioControl.init();
    // Default role as receiver
    bRole = RF_RECEIVER;
    // reset variable
    senderBusy = 0;

    return SUCCESS;
  }


  /**
   * Start transceiver according to the role. Therefore
   * the role should be set after init() command
   * and before start() command.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t rfTransceiver.start() {
    // Start the system if it is a receiver
    if (bRole == RF_RECEIVER)
       call RadioControl.start();
    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t rfTransceiver.stop() {
    call RadioControl.stop();
    return SUCCESS;
  }


  /**
   * Send radio packet.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t rfTransceiver.Send(TOS_MsgPtr msg)
  {
    int result = 0;
    atomic senderBusy = 1;
    call RadioControl.start();
    result = call RadioSend.send(msg);

    return SUCCESS;
  }

  /**
   * Default event handler when sending is done.
   *
   **/
  default event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success){

  }

  /**
   * Default event handler when a radio packet arrived.
   *
   **/
  default event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data){

  }

  /**
   * Fetch current sender status by return busy flag.
   *
   * @return Always returns senderBusy
   **/
  command bool rfTransceiver.isSenderBusy() {
     return senderBusy;
  }

  /**
   * This event is signaled from underlying software
   * indicate an packet is sent.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t RadioSend.sendDone(TOS_MsgPtr rmsg, result_t success) {
        call RadioControl.stop();
        atomic senderBusy = 0;
        signal rfTransceiver.sendDone(rmsg, success);
return success;
  }


  /**
   * This event is signaled from underlying software,
   * indicate an actual packet is received.
   *
   * @return Always returns radio packet received.
   **/
  event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr data) {
    signal rfTransceiver.receive(data);
    return data;
  }

}
```

rfTransceiverC.nc

```
 includes rfTransceiver;


configuration rfTransceiverC {
  provides {
   interface rfTransceiver;
   interface RadioCoordinator as RadioSendCoord;
      interface RadioCoordinator as RadioReceiveCoord;
      async command uint8_t GetRxBitOffset();
  }
}
implementation {
  components rfTransceiverM, LedsC;
  components RadioCRCPacket as Comm, CC1000RadioIntM as Radio;


  // rfTransceiverM
  rfTransceiver = rfTransceiverM.rfTransceiver;
  rfTransceiverM.Leds -> LedsC;
  rfTransceiverM.RadioControl -> Comm;
  rfTransceiverM.RadioSend -> Comm;
  rfTransceiverM.RadioReceive -> Comm;
  GetRxBitOffset = Radio.GetRxBitOffset;
  RadioSendCoord = Radio.RadioSendCoordinator;
  RadioReceiveCoord = Radio.RadioReceiveCoordinator;
}
```

rfTransceiver.nc

```
 includes AM;

interface rfTransceiver {

  command result_t setRole(bool role);

  command result_t init();

  command result_t start();

  command result_t stop();

  command result_t Send(TOS_MsgPtr msg);
  command bool isSenderBusy();

  event result_t sendDone(TOS_MsgPtr rmsg, result_t success);

  event TOS_MsgPtr receive(TOS_MsgPtr data);

}
```

rfTransceiver.h

```
#ifndef _RFTRANSCEIVER_H
#define _RFTRANSCEIVER_H

#include <stdio.h>


#define SYN1 0x0f
#define SYN2 0x0e

#define RF_SENDER 0
#define RF_RECEIVER 1

#endif
```

G.4 RF sender and receiver

rfSenderM.nc

```
module rfSenderM {
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;

    async command uint8_t GetRxBitOffset();
  }
}
implementation {

  // Message
  TOS_Msg   msg;

  /**
   * Initialize the LED, rfTransceiver(as sender).
   *
   * @return Always returns <code>SUCCESS</code>
   **/
```

```
command result_t StdControl.init() {
  int i;
  call Leds.init();
  call rfTransceiver.init();
  call rfTransceiver.setRole(RF_SENDER);

  msg.length = 23;
  msg.type = SYN1;
  msg.addr = TOS_BCAST_ADDR;

  for(i=0;i<8;i++)
      msg.data[i] = 0x00;
  msg.data[0] = 0x30;
  return SUCCESS;
}


/**
 * Start 1 second timer, and start radio transceiver.
 *
 * @return Always returns <code>SUCCESS</code>
 **/
command result_t StdControl.start() {
  // Start a repeating timer that fires every 1000ms
  call rfTransceiver.start();
  call Timer.start(TIMER_REPEAT, 1000);
  return SUCCESS;
}

/**
 * Halt execution of the application.
 *
 * @return Always returns <code>SUCCESS</code>
 **/
command result_t StdControl.stop() {
  call rfTransceiver.stop();
  call Timer.stop();
  return SUCCESS;
}


/**
 * Send a radio packet. A number is sent, circulating from 0x30 to 0x37
 *
 * @return Always returns <code>SUCCESS</code>
 **/
event result_t Timer.fired()
{
  int result = 0;
  if (! call rfTransceiver.isSenderBusy()) {
        // Send a packet
        result = call rfTransceiver.Send(&msg);
        if (result == SUCCESS)
          {
           // test pattern generator
           msg.data[0]++;
           if (msg.data[0] == 0x38)
              msg.data[0] = 0x30;
              call Leds.greenToggle();
          }
  }
  return SUCCESS;
}
```

```
  /**
   * This event is signaled when packet is sent.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {

return SUCCESS;
  }


  /**
   * This event is signalled when a radio packet is received
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {

    return data;
  }
}
```

rfSenderC.nc

```
 includes rfSender;
includes rfTransceiver;

configuration rfSenderC {

}
implementation {
  components Main, rfSenderM,TimerC, LedsC;
  components rfTransceiverC;

  Main.StdControl -> rfSenderM;
  Main.StdControl -> TimerC;
  rfSenderM.Timer -> TimerC.Timer[unique("Timer")];
  rfSenderM.Leds -> LedsC;

  // Radio
  rfSenderM.rfTransceiver -> rfTransceiverC.rfTransceiver;
}
```

rfSender.h

```
#ifndef _RFSENDER_H
#define _RFSENDER_H
#include <stdio.h>
#define SYN1 0x0f
#endif
```

rfReceiverM.nc

```
module rfReceiverM {
 provides {
   interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;

    async command uint8_t GetRxBitOffset();
  }
}
implementation {

  // Message
  TOS_Msg   msg;


  /**
   * Initialize the LED, rfTransceiver(as receiver).
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    int i;
    // init LEDs
    call Leds.init();
    // init RF and Ultrasound transceiver
    call rfTransceiver.init();
    // set RF transceiver as receiver
    call rfTransceiver.setRole(RF_RECEIVER);

    // format RF message
    msg.length = 23;
    msg.type = SYN1;
    msg.addr = TOS_BCAST_ADDR;

    for(i=0;i<8;i++)
        msg.data[i] = 0x00;

    return SUCCESS;
  }


  /**
   * Start timer and radio receiver.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 500ms
    call Timer.start(TIMER_REPEAT, 500);
    // start the RF receiver
    call rfTransceiver.start();

    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    // stop everything
    call rfTransceiver.stop();
    call Timer.stop();
    return SUCCESS;
  }


  /**
   * When timer is fired, this event is signaled. Not used in this application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {

    return SUCCESS;
  }

  /**
   * This event is signaled when packet is sent.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {

return SUCCESS;
  }


  /**
   * This event is signalled when a radio packet is received.
   * It displays the data received. The number should be between 0x30 and 0x37
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {
    switch(data->data[0]) {
                    case 0x30: call Leds.yellowOff();call Leds.greenOff();call Leds.redOff();break;
                    case 0x31: call Leds.yellowOn();call Leds.greenOff();call Leds.redOff();break;
                    case 0x32: call Leds.yellowOff();call Leds.greenOn();call Leds.redOff();break;
                    case 0x33: call Leds.yellowOn();call Leds.greenOn();call Leds.redOff();break;
                    case 0x34: call Leds.yellowOff();call Leds.greenOff();call Leds.redOn();break;
                    case 0x35: call Leds.yellowOn();call Leds.greenOff();call Leds.redOn();break;
                    case 0x36: call Leds.yellowOff();call Leds.greenOn();call Leds.redOn();break;
                    case 0x37: call Leds.yellowOn();call Leds.greenOn();call Leds.redOn();break;
                  }
    return data;
  }
}
```

rfReceiverC.nc

```
 includes rfReceiver;
includes rfTransceiver;

configuration rfReceiverC {

}
implementation {
  components Main, rfReceiverM,TimerC, LedsC;
  components rfTransceiverC;
  Main.StdControl -> rfReceiverM;
  Main.StdControl -> TimerC;
  rfReceiverM.Timer -> TimerC.Timer[unique("Timer")];
  rfReceiverM.Leds -> LedsC;

  // Radio
  rfReceiverM.rfTransceiver -> rfTransceiverC.rfTransceiver;
}
```

rfReceiver.h

```
#ifndef _RFRECEIVER_H
#define _RFRECEIVER_H

#include <stdio.h>
#define SYN1 0x0f
#endif
```

G.5 Ultrasound ranging

KeyReceiverM.nc

```
 /*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:            Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     KeyReceiverM.nc, 2006/03/15
 */


module KeyReceiverM {
 provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;
    interface usTransceiver;
    interface RadioCoordinator as RadioSendCoord;
    interface RadioCoordinator as RadioReceiveCoord;
interface RS232;

    async command uint8_t GetRxBitOffset();
  }
```

```
}
implementation {

  // Message
  TOS_Msg   msg;


  uint8_t count;
  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    int i;
    // init LEDs
    call Leds.init();
    // init RF and Ultrasound transceiver
    call rfTransceiver.init();
    call usTransceiver.init();
    // set RF transceiver as receiver
    call rfTransceiver.setRole(RF_RECEIVER);
    // set Ultrasound transceiver as sender
    call usTransceiver.setRole(US_SENDER);
call RS232.init();
    // format RF message, boardcast
    msg.length = 29;
    msg.type = SYN1;
    msg.addr = TOS_BCAST_ADDR;


    for(i=0;i<8;i++)
        msg.data[i] = 0x00;

    return SUCCESS;
  }


  /**
   * Start Timer, radio receiver and ultrasound transmitter
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 500ms
    call Timer.start(TIMER_REPEAT, 500);
    // start the RF receiver
    call rfTransceiver.start();
    // start the US sender;
    call usTransceiver.start();

    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    // stop everything
    call rfTransceiver.stop();
    call Timer.stop();
    call usTransceiver.stop();
    return SUCCESS;
  }


  /**
   * Timer fire, no action taken here
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
    return SUCCESS;
  }

  /**
   * RF message send done
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {

return SUCCESS;
  }


  /**
   * A RF message received
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {

    return data;
  }


  async event void RadioSendCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {}
  async event void RadioSendCoord.blockTimer() {}
  /**
   * RadioSend Coordinator
   *
   **/
  async event void RadioSendCoord.byte(TOS_MsgPtr rfmsg, uint8_t cnt)
  {

  }

  task void report_rx() {
     uint8_t c;
 atomic c = count;
     printf("byte @%u\r\n",c);
  }


  async event void RadioReceiveCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {};
  async event void RadioReceiveCoord.blockTimer() {};
  /**
   * RadioReceive Coordinate
   *
   **/
  async event void RadioReceiveCoord.byte(TOS_MsgPtr rfmsg, uint8_t cnt)
  {
   uint16_t compensation;

    if (cnt == (offsetof(struct TOS_Msg,type) +
                  sizeof(((struct TOS_Msg*)0)->type))+3) {
```

```
      /*
    * Get radio receiver hardware offset,
    * this is the hardware overhead that experienced.
    * it should be sent back to KeySender for distance correction.
    */
   atomic compensation = call GetRxBitOffset();
   call usTransceiver.SendWord(compensation);
   atomic count = rfmsg->data[0];
   post report_rx();
   }
  }

  /**
   * A US message sent
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t usTransceiver.SendDone(){

     return SUCCESS;
  }

  /**
   * A US message detected. It is not used in this program
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint16_t usTransceiver.msgDetected(uint16_t timeStamp){

   return timeStamp;
  }


  /**
   * US data packet received, not used in this program.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint32_t usTransceiver.DataReceived(uint32_t Rx_data , uint16_t timeStamp){
    return Rx_data;
  }

  /**
   * A RF message received, serial communication.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }
}
```

KeyReceiver.nc

```
 /**
 * @author Feng Kai s030656@student.dtu.dk
 **/

includes KeyReceiver;
includes rfTransceiver;
includes usTransceiver;

configuration KeyReceiver {

}
implementation {
  components Main, KeyReceiverM,TimerC, LedsC;
  components rfTransceiverC;
  components usTransceiverC;
  components RS232C;

  Main.StdControl -> KeyReceiverM;
  Main.StdControl -> TimerC;
  KeyReceiverM.Timer -> TimerC.Timer[unique("Timer")];
  KeyReceiverM.Leds -> LedsC;

  // Radio
  KeyReceiverM.rfTransceiver -> rfTransceiverC.rfTransceiver;
  KeyReceiverM.RadioSendCoord -> rfTransceiverC.RadioSendCoord;
  KeyReceiverM.RadioReceiveCoord -> rfTransceiverC.RadioReceiveCoord;
  KeyReceiverM.GetRxBitOffset -> rfTransceiverC.GetRxBitOffset;


  //Ultra Sound
  KeyReceiverM.usTransceiver -> usTransceiverC.usTransceiver;

  //RS232
  KeyReceiverM.RS232 -> RS232C.RS232;
}
```

KeyReceiver.h

```
#ifndef _KEYRECEIVER_H
#define _KEYRECEIVER_H

#include <stdio.h>

#define SYN1 0x0f
#endif
```

KeySenderM.nc

```
 /*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     KeySenderM.nc, 2006/03/15
 */
```

```
module KeySenderM {
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;
    interface usTransceiver;
    interface RS232;
    interface RadioCoordinator as RadioSendCoord;
    interface RadioCoordinator as RadioReceiveCoord;

    async command uint8_t GetRxBitOffset();
  }
}
implementation {

  // Message
  TOS_Msg   msg;
  // Replay flag for ultrasound message
  bool usReply = 1;
  // The data to be sent
  uint8_t data;
  // Radio sending time
  uint16_t sendT;
  // ultrasound back time
  uint16_t backT;
  // The time corresponding to the true distance
  uint16_t integrityTime;

  /**
   * Initialize the LED, radio transceiver, ultrasound transceiver and RS232 link.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    int i;
    call Leds.init();
    call rfTransceiver.init();
    call rfTransceiver.setRole(RF_SENDER);
    call usTransceiver.init();
    call usTransceiver.setRole(US_RECEIVER);
    call RS232.init();

    msg.length = 29;
    msg.type = SYN1;
    msg.addr = TOS_BCAST_ADDR;

    for(i=0;i<29;i++)
        msg.data[i] = i;
    return SUCCESS;
  }


  /**
   * Start radio transceiver and timer.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 2000ms
    call rfTransceiver.start();
    call Timer.start(TIMER_REPEAT, 2000);
    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    call rfTransceiver.stop();
    call usTransceiver.stop();
    call Timer.stop();
    return SUCCESS;
  }


  /**
   * When timer fired, send a radio packet.
   * And incremental data.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
    int result = 0;
int i;
    if ((! (call rfTransceiver.isSenderBusy())) && usReply) {
         usReply = 0;

          result = call rfTransceiver.Send(&msg);
          if (result == SUCCESS)
            {
for (i= 0;i<29;i++) {
                   msg.data[i]++;
}
            }
      }

    return SUCCESS;
  }


  task void report_sendtime() {
     uint16_t st;
     atomic st = sendT;
     printf("Sent Time: %u us\r\n",st);
  }

  /**
   * This event is signalled when radio packet is sent.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {

return SUCCESS;
  }


  /**
   * This event is signalled when radio packet is received.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr rfdata) {

    return rfdata;
  }

  /**
   * This event is signalled when US data is sent.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t usTransceiver.SendDone() {

    return SUCCESS;
  }



  task void report_backTime() {
      printf("backtime: %u us\r\n",backT);
  }

  task void report_integrity(){

    uint16_t distance = 0;
    uint16_t measuredTime;
    uint16_t truetime;

     atomic {
     // The time-of-flight
   integrityTime = backT - sendT;
         measuredTime = integrityTime;
      // The time-of-flight, corrected with processint time
   truetime = integrityTime - (data - 7)*48;
   // corrected with timer offset
truetime -= 900;
      }
   if ((data<=7)&&(truetime>10)) {
        // s = v/t
        distance = (uint16_t)(truetime)/10 * 34;
        // distance in centimeter
        distance /=100;

    printf("uncorrected time: %u us,integrity time: %u us, Distance: %u cm\r\n",
           measuredTime,truetime,distance);
    }
    else
    {
     printf("No update, integrity time: %u us, Distance: %u cm\r\n",integrityTime,distance);
    }

  }

  /**
   * This event is signalled when a message is detected.
   * This gives a time when the packet is first reached receiver.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint16_t usTransceiver.msgDetected(uint16_t timeStamp){
    call Leds.greenOff();
   atomic backT = timeStamp;
   post report_backTime();

   return timeStamp;
  }

  task void report_data() {
     printf("offset: %x\r\n",data);
  }


  /**
   * This event is signalled when a US packet is received.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint32_t usTransceiver.DataReceived(uint32_t Rx_data , uint16_t timeStamp){
     atomic data = Rx_data;

     post report_data();
     post report_integrity();
     call usTransceiver.stop();
      if (call usTransceiver.Rx_dataHealthy()) {


       }
       else
         {
        }

   usReply = 1;
    return Rx_data;
  }


  async event void RadioSendCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {}
  async event void RadioSendCoord.blockTimer() {}
  async event void RadioSendCoord.byte(TOS_MsgPtr rfmsg, uint8_t cnt)
  {
    if (cnt == (offsetof(struct TOS_Msg,data) + 3)){
   // When synchronized, start US receiver.
 call Leds.greenOn();
    call usTransceiver.start();
    sendT = call usTransceiver.getTimeStamp();
    post report_sendtime();
      }

  }

  async event void RadioReceiveCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {};
  async event void RadioReceiveCoord.blockTimer() {};
  async event void RadioReceiveCoord.byte(TOS_MsgPtr rfmsg, uint8_t cnt)
  {

  }

  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }
}
```

KeySender.nc

```
/**
```

```
 * @author Feng Kai s030656@student.dtu.dk
 **/

includes KeySender;
includes rfTransceiver;
includes usTransceiver;

configuration KeySender {

}
implementation {
  components Main, KeySenderM,TimerC, LedsC;
  components rfTransceiverC;
  components usTransceiverC, TimerC as usTimerC;
  components RS232C;

  Main.StdControl -> KeySenderM;
  Main.StdControl -> TimerC;
  KeySenderM.Timer -> TimerC.Timer[unique("Timer")];
  KeySenderM.Leds -> LedsC;

  // Radio
  KeySenderM.rfTransceiver -> rfTransceiverC.rfTransceiver;
  KeySenderM.RadioSendCoord -> rfTransceiverC.RadioSendCoord;
  KeySenderM.RadioReceiveCoord -> rfTransceiverC.RadioReceiveCoord;

  // Ultrasound
  KeySenderM.usTransceiver -> usTransceiverC.usTransceiver;

  //RS232
  KeySenderM.RS232 -> RS232C.RS232;
}
```

KeySender.h

```
#ifndef _KEYSENDER_H
#define _KEYSENDER_H

#include <stdio.h>
#define SYN1 0x0f
#endif
```

G.6 TinySec and Integrity

MACsenderM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */
```

```
/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     MACsenderM.nc, 2006/03/08
 */


module MACsenderM {
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;
    interface MAC;

    async command uint8_t GetRxBitOffset();
  }
}
implementation {

  // Message
  TOS_Msg   msg;

  // MAC variables
  MACContext macContext;
  uint8_t key_tmp[2*TINYSEC_KEYSIZE] = {TINYSEC_KEY};
  uint8_t MACKey[TINYSEC_KEYSIZE];

  // My text
  uint8_t myText[30];
  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    int i;
    call Leds.init();
    call rfTransceiver.init();
    call rfTransceiver.setRole(RF_SENDER);

    //MAC
    memcpy(MACKey,key_tmp+TINYSEC_KEYSIZE,TINYSEC_KEYSIZE);

    call MAC.init(&macContext,TINYSEC_KEYSIZE,MACKey);

    msg.length = 29;
    msg.type = SYN1;
    msg.addr = TOS_BCAST_ADDR;

    for(i=0;i<msg.length;i++)
        myText[i] = i+0x30;

    for(i=0;i<msg.length;i++)
        msg.data[i] = 0;

    return SUCCESS;
  }


  /**
   * Start timer and RF transceiver
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {

    call rfTransceiver.start();
    call Timer.start(TIMER_REPEAT, 3000);
    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    call rfTransceiver.stop();
    call Timer.stop();
    return SUCCESS;
  }


  /**
   * Each time the Timer fired, send a message to receiver.
   * The clear text message is sent first, MAC is sent at the next round.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
    static int mac = 0;
    int i;
    result_t result;
    if (! call rfTransceiver.isSenderBusy()) {
         // Prepare clear text message
          if (!mac) {
             memcpy(msg.data,myText,30);

             mac = 1;
            }
          else {
             // compute MAC
              call MAC.MAC(&macContext,
                          (uint8_t*) myText, CLEARTEXTSIZE,
    msg.data,CIPHERSIZE);
               for(i=0;i<msg.length;i++)
                 myText[i] +=1;
   mac = 0;
          }

          // Send actual message over radio link
          result = call rfTransceiver.Send(&msg);
          if (result == SUCCESS)
            {
                call Leds.greenToggle();
            }
    }
    return SUCCESS;
  }


  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {

return SUCCESS;
  }
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {

    return data;
  }

}
```

MACsenderC.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/

includes MACsender;
includes rfTransceiver;
includes CryptoPrimitives;

configuration MACsenderC {

}
implementation {
  components Main, MACsenderM,TimerC, LedsC;
  components rfTransceiverC;
  components CBCMAC as MAC;
  components SkipJackM as Cipher;

  Main.StdControl -> MACsenderM;
  Main.StdControl -> TimerC;
  MACsenderM.Timer -> TimerC.Timer[unique("Timer")];
  MACsenderM.Leds -> LedsC;

  // Radio
  MACsenderM.rfTransceiver -> rfTransceiverC.rfTransceiver;

  //MAC
  MACsenderM.MAC -> MAC.MAC;
  MAC.BlockCipher -> Cipher.BlockCipher;
  MAC.BlockCipherInfo -> Cipher.BlockCipherInfo;

}
```

MACsender.h

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/
#ifndef _MACSENDER_H
#define _MACSENDER_H

#include <stdio.h>
#define SYN1 0x0f
#define CLEARTEXTSIZE 29
#define CIPHERSIZE 4
```

```
#endif
```

.tinyos_keyfile

```
# TinySec Keyfile. By default, the first key will be used.
# You can import other keys by appending them to the file.

default 2D1565C6B1138D394CF1B5D45801E75E
```

MACreceiverM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     MACreceiverM.nc, 2006/03/08
 */


module MACreceiverM {
 provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;
    interface MAC;
    interface RS232;

    async command uint8_t GetRxBitOffset();
  }
}
implementation {

  // Message
  TOS_Msg   msg;

  // MAC variables
  MACContext macContext;
  uint8_t key_tmp[2*TINYSEC_KEYSIZE] = {TINYSEC_KEY};
  uint8_t MACKey[TINYSEC_KEYSIZE];

  // My text
  uint8_t myText[30];
  uint8_t cipher[30];
```

```
/**
 * Initialize the component.
 *
 * @return Always returns <code>SUCCESS</code>
 **/
command result_t StdControl.init() {
  int i;
  // init LEDs
  call Leds.init();
  // init RF and Ultrasound transceiver
  call rfTransceiver.init();
  // set RF transceiver as receiver
  call rfTransceiver.setRole(RF_RECEIVER);
  // RS232
  call RS232.init();

  //MAC
  memcpy(MACKey,key_tmp+TINYSEC_KEYSIZE,TINYSEC_KEYSIZE);

  call MAC.init(&macContext,TINYSEC_KEYSIZE,MACKey);

  // format RF message
  msg.length = 29;
  msg.type = SYN1;
  msg.addr = TOS_BCAST_ADDR;

  for(i=0;i<msg.length;i++)
      msg.data[i] = 0x00;

  for(i=0;i<msg.length;i++)
     myText[i] = i+0x30;

  return SUCCESS;
}


/**
 * Start timer and RF transceiver
 *
 * @return Always returns <code>SUCCESS</code>
 **/
command result_t StdControl.start() {
  // Start a repeating timer that fires every 500ms
  call Timer.start(TIMER_REPEAT, 500);
  // start the RF receiver
  call rfTransceiver.start();

  return SUCCESS;
}

/**
 * Halt execution of the application.
 *
 * @return Always returns <code>SUCCESS</code>
 **/
command result_t StdControl.stop() {
  // stop everything
  call rfTransceiver.stop();
  call Timer.stop();
  return SUCCESS;
}


event result_t Timer.fired()
{
    return SUCCESS;
  }


  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {

return SUCCESS;
  }


  /**
   * Receive radio message containing cipher text and clear text.
   * This event handler also verifies the integrity of the message.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {
    char buf[80];
    static int mac = 0;
    int i;
    if (!mac) {
     // clear text received, compute MAC over clear text
       call MAC.MAC(&macContext,
                      (uint8_t *)data->data , CLEARTEXTSIZE,
                     myText, CIPHERSIZE );
        printf("Clear text: ");
       for (i=0;i<CLEARTEXTSIZE;i++)
         printf("%x",data->data[i]);
       printf("\r\n");
       mac = 1;
    }
    else {
     // MAC received, compare MAC computed with the one received
      mac = 0;
      printf("MAC received: ");
      for (i=0;i<CIPHERSIZE ;i++)
         printf("%x ",(uint8_t)(data->data[i]));
       printf("\r\n");
       printf("MAC computed: %x,%x,%x,%x\r\n",myText[0],myText[1],myText[2],myText[3]);

     }
  }


  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }

}
```

MACreceiverC.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


includes MACreceiver;
includes rfTransceiver;
includes CryptoPrimitives;
```

```
configuration MACreceiverC {

}
implementation {
  components Main, MACreceiverM,TimerC, LedsC;
  components rfTransceiverC;
  components CBCMAC as MAC;
  components SkipJackM as Cipher;
  components RS232C;

  Main.StdControl -> MACreceiverM;
  Main.StdControl -> TimerC;
  MACreceiverM.Timer -> TimerC.Timer[unique("Timer")];
  MACreceiverM.Leds -> LedsC;

  // Radio
  MACreceiverM.rfTransceiver -> rfTransceiverC.rfTransceiver;

  //MAC
  MACreceiverM.MAC -> MAC.MAC;
  MAC.BlockCipher -> Cipher.BlockCipher;
  MAC.BlockCipherInfo -> Cipher.BlockCipherInfo;

  //RS232
  MACreceiverM.RS232 -> RS232C.RS232;

}
```

MACreceiver.h

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/
#ifndef _MACRECEIVER_H
#define _MACRECEIVER_H

#include <stdio.h>
#define SYN1 0x0f
#define CLEARTEXTSIZE 29
#define CIPHERSIZE 4
#endif
```

.tinyos_keyfile

```
# TinySec Keyfile. By default, the first key will be used.
# You can import other keys by appending them to the file.

default 2D1565C6B1138D394CF1B5D45801E75E
```

G.7 Debugging

UseRS232M.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     UseRS232M.nc, 2006/02/25
 */


module UseRS232M{
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
    interface RS232;
  }
}
implementation {

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    call RS232.init();
    return SUCCESS;
  }


  /**
   * Start things up.  This just sets the rate for the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }
  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }

  /**
   * Toggle the red LED in response to the <code>Timer.fired</code> event.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
    char data[10];
    sprintf(data,"Hello!!\r\n");
    call RS232.SendString(data);
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

UseRS232.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/

configuration UseRS232 {
}
implementation {
   components Main, UseRS232M, RS232C,LedsC, TimerC;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> UseRS232M.StdControl;
  UseRS232M.Timer -> TimerC.Timer[unique("Timer")];
  UseRS232M.Leds -> LedsC.Leds;
  UseRS232M.RS232 -> RS232C.RS232;
}
```

RS232M.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     RS232M.nc, 2006/02/25
 */

module RS232M {
  provides {
    interface RS232;
  }
  uses {
    interface Serial;
  }
}
implementation {

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t RS232.init() {
   // Enable RS-232 Chip
      TOSH_CLR_US_IN_EN_PIN();
TOSH_SET_BAT_MON_PIN();

    // Wait until the RS-232 start
      TOSH_uwait(2000);

    // Init serial port
      call Serial.SetStdoutSerial();

    return SUCCESS;
  }

  async command result_t RS232.SendString(char *data)
  {
      printf("%s",data);
      return SUCCESS;
  }

  event result_t Serial.Receive(char * buf, uint8_t data_len) {
   signal RS232.Receive(buf,data_len);
   return SUCCESS;
  }
}
```

RS232C.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/

configuration RS232C {
  provides interface RS232;
}

implementation {
  components RS232M;
  components SerialM, HPLUARTC,LedsC;

  RS232      = RS232M.RS232;

  RS232M.Serial   -> SerialM.Serial;
  SerialM.HPLUART -> HPLUARTC;
  SerialM.Leds    -> LedsC;
}
```

RS232.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/
includes RS232;

interface RS232 {

  command result_t init();

  async command result_t SendString(char data[]);

  event result_t Receive(char * buf, uint8_t data_len);
}
```

RS232.h

```
#ifndef _RS232_H
#define _RS232_H

#include <stdio.h>

uint8_t debug_out = 1;

#define UARTOutput(__x,__y,__args...) do { \
    static char __s[] PROGMEM = __y; \
    if (debug_out) \
        printf_P(__s, ## __args);       \
        } while (0)

#endif
```

SerialM.nc

```
/*
 * SerialM.nc
 * David Moore <dcm@mit.edu>
 *
 * Module to connect standard output to the serial port.  Allows use
 * of printf() for debugging and status messages.
 *
 * Usage:
 *   call Serial.SetStdoutSerial() to attach stdout to the Serial UART.
 *   Any further commands that use stdout such as printf() will output
 *   to the serial port.
 */

/*
      This software may be used and distributed according to the terms
      of the GNU General Public License (GPL), incorporated herein by reference.
      Drivers based on this skeleton fall under the GPL and must retain
      the authorship (implicit copyright) notice.
      This program is distributed in the hope that it will be useful, but
      WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
      General Public License for more details.
*/

module SerialM {
    provides {
        interface Serial;
    }
    uses {
        interface HPLUART;
        interface Leds;
    }
}
implementation {
#define IDLE    0xff
#define WAITING 0xfe
#define BUF_LEN 32
#define RECV_BUF_LEN 32
    volatile uint8_t tx_head = IDLE;
    volatile uint8_t tx_tail = 0;
    uint8_t txbuf[BUF_LEN];
    uint8_t rx_state = IDLE;
    uint8_t rx_len = 0;
    uint8_t rxbuf[RECV_BUF_LEN];

    int put(char c) __attribute__ ((C,spontaneous))
    {
        uint8_t start_tx = 0;
        uint8_t new_tx_tail = 0;
        uint8_t new_tx_head = 1;

        /* Do the busy wait if the tx buffer is full */
        while (new_tx_head != IDLE &&
                (new_tx_tail+1)%BUF_LEN == new_tx_head) {
            atomic {
                new_tx_tail = tx_tail;
                new_tx_head = tx_head;
            }
        }

        atomic {
            /* Initialize queue */
            if (tx_head == IDLE) {
                txbuf[0] = c;
                tx_head = tx_tail = 0;
                start_tx = 1;
            }
            /* Enqueue */
            else {
                txbuf[tx_tail] = c;
                tx_tail = (tx_tail+1)%BUF_LEN;
            }
        }
        if (start_tx)
            call HPLUART.put(txbuf[0]);

        return 0;
    }

    async event result_t HPLUART.putDone() {
        uint8_t do_tx = 0;
        uint8_t tx_ch = 0;
        atomic {
            if (tx_head == tx_tail || tx_head == IDLE) {
                tx_head = IDLE;
            }
            else {
                do_tx = 1;
                tx_ch = txbuf[tx_head];
                tx_head = (tx_head+1)%BUF_LEN;
            }
        }
        if (do_tx)
            call HPLUART.put(tx_ch);
        return SUCCESS;
    }

    void task recv_str()
    {
        uint8_t len;
        atomic len = rx_len;
        signal Serial.Receive(rxbuf, len);
        atomic {
            rx_len = 0;
            rx_state = IDLE;
        }
    }

#ifndef PLATFORM_PC
    void outc(uint8_t c)
    {
        loop_until_bit_is_set(UCSR0A, UDRE);
        outp(c, UDR0);
    }

    void printbyte(uint8_t b)
    {
        uint8_t n;
        n = (b & 0xf0) >> 4;
        if (n <= 9)
            outc('0' + n);
        else
            outc(n - 10 + 'A');
        n = (b & 0x0f);
        if (n <= 9)
            outc('0' + n);
        else
            outc(n - 10 + 'A');
        outc(' ');
    }

    void stack_trace(uint8_t extra, uint16_t val) __attribute__ ((C,spontaneous))
    {
        uint8_t * ptr = (uint8_t *) inw(SPL);

        /* Disable serial interrupts */
        uint8_t tmp = inb(UCSR0B);
        outp((1 << RXEN) | (1 << TXEN), UCSR0B);

        outc('\n');
        if (extra) {
            outc('<');
            printbyte(val & 0xff);
            printbyte((val >> 8) & 0xff);
            outc('>');
        }

        printbyte(inb(SPH));
        printbyte(inb(SPL));
        outc(':');

        for (ptr += 27; ptr < (uint8_t *)0x1100; ptr++) {
            printbyte(*ptr);
        }
        outc('\n');

        outp(tmp, UCSR0B);
    }
#endif

    async event result_t HPLUART.get(uint8_t data)
    {
        uint8_t state;

        atomic state = rx_state;

//#ifndef PLATFORM_PC
//        if (data == 'q')
//            stack_trace(0, 0);
//#endif

//        if (data == '\r' || state == WAITING)
//            stack_trace(0, 0);
//            return SUCCESS;

        if (data == '\r' || data == '\n') {
            if (post recv_str()) {
                atomic {
                    rx_state = WAITING;
                    rxbuf[rx_len] = '\0';
                }
            }
            else {
                atomic rx_len = 0;
            }
            return SUCCESS;
        }

        atomic {
            if (rx_len < RECV_BUF_LEN-1)
                rxbuf[rx_len++] = data;
        }

        return SUCCESS;
    }

    command result_t Serial.SetStdoutSerial() {
#ifndef PLATFORM_PC
        /* Initialize the serial port */
        call HPLUART.init();
        /* Use the put() function for any output to stdout. */
        fdevopen(put, NULL, 0);
#endif
        return SUCCESS;
    }
}
```

Serial.nc

```
/*
 * Serial.nc
 * David Moore <dcm@mit.edu>
 *
 * Module to connect standard output to the serial port.  Allows use
 * of printf() for debugging and status messages.
 *
 * Usage:
 *   call Serial.SetStdoutSerial() to attach stdout to the Serial UART.
 *   Any further commands that use stdout such as printf() will output
 *   to the serial port.
 */

/*
      This software may be used and distributed according to the terms
      of the GNU General Public License (GPL), incorporated herein by reference.
      Drivers based on this skeleton fall under the GPL and must retain
      the authorship (implicit copyright) notice.

      This program is distributed in the hope that it will be useful, but
      WITHOUT ANY WARRANTY; without even the implied warranty of
      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
      General Public License for more details.
*/

interface Serial {
    command result_t SetStdoutSerial();
    event result_t Receive(char * buf, uint8_t len);
}
```

G.8 Protocol implementation

PtlSenderM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     PtlSenderM.nc, 2006/02/27 22:00:00
 */


module PtlSenderM {
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;
    interface usTransceiver;
    interface MAC;
    interface RadioCoordinator as RadioSendCoord;
    interface RadioCoordinator as RadioReceiveCoord;
    interface RS232;

    async command uint8_t GetRxBitOffset();
  }
}
implementation {

/*
 * System variables
 */
   /* Protocol state */
   uint8_t state;

/*
 * Radio
 */
   /* RF package */
   TOS_Msg   msg;
   /* RF receive bit offset */
   uint8_t rf_offset;
/*
 *MAC variables
 */
   /* MAC context */
   MACContext macContext;
   /* Keys */
   uint8_t key_tmp[2*TINYSEC_KEYSIZE] = {TINYSEC_KEY};
   uint8_t MACKey[TINYSEC_KEYSIZE];
   uint8_t KeyReceived[TINYSEC_KEYSIZE];

   /* Clear texts */
   char myText[4][30] = TEXT;
   int textID;

   /* NA index */
   uint8_t idxNA;

/* watch dog timer*/
   uint8_t wdTime;

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    int i;
    /* init LEDs */
    call Leds.init();
    /* init RF transceiver set as sender */
    call rfTransceiver.init();
    call rfTransceiver.setRole(RF_SENDER);
    /* US receiver, set role as sender */
    call usTransceiver.init();
    call usTransceiver.setRole(US_SENDER);

    /* MAC */
    memcpy(MACKey,key_tmp+TINYSEC_KEYSIZE,TINYSEC_KEYSIZE);
    call MAC.init(&macContext,TINYSEC_KEYSIZE,MACKey);

    /* RS232 */
    call RS232.init();

    /* format RF message */
    msg.length = RF_DATA_LENGTH;
    msg.type = SYN1;
    msg.addr = TOS_BCAST_ADDR;

    for(i=0;i<RF_DATA_LENGTH;i++)
        msg.data[i] = 0x00;

   atomic {
    /* Initial system state */
    state = PHASE_START;
    /* Text string ID */
    textID = 0;
    /* initialize NA index */
    idxNA = 0;
/* Initial watch timer*/
wdTime = 0;
}
    return SUCCESS;
  }


  /**
   *  Start system FSM, start RF sender and US sender
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    /* start the RF receiver */
    call rfTransceiver.start();
    call usTransceiver.start();
    /* start FSM */
    call Timer.start(TIMER_REPEAT, SYSTEM_REACTION_RATE);
    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    /* stop everything */
    call rfTransceiver.stop();
    call usTransceiver.stop();
    call Timer.stop();
    return SUCCESS;
  }

 /**
  *  Commitment
  **/
  task void compute_cipher(){
      /* compute MAC */
      printf("\r\n");
       call MAC.MAC(&macContext,
                    (uint8_t*) myText[textID], strlen(myText[textID]),
                     msg.data ,MACSIZE);
      /* bring the system into PHASE 1*/
      atomic state = PHASE1;

  }
```

```
/**
 *  Watchdog
 **/
uint8_t evaluateWatchdog() {
     uint8_t t;
 uint8_t result;
 atomic t = (++wdTime);
 if (t >= WATCHDOGTIME)
    {  /* time out */
    result = 1;
atomic wdTime = 0;
    }
 else
    result = 0;
 return result;
 }

/**
 *  reset watch dog
 **/
 void resetWatchdog() {
     atomic wdTime = 0;
 }


/**
 *  watchdog time out
 **/
 task void WatchdogFired() {
     int i;
     printf("Session time out! Key establishment failed, restart...\r\n");
 atomic {
              /* init RF transceiver set as sender */
              call rfTransceiver.setRole(RF_SENDER);
              /* format RF message */
              msg.length = RF_DATA_LENGTH;
              msg.type = SYN1;
              msg.addr = TOS_BCAST_ADDR;

              for(i=0;i<RF_DATA_LENGTH;i++)
              msg.data[i] = 0x00;

              /* Text string ID */
              textID = 0;
              /* initialize NA index */
              idxNA = 0;
              /* Initial watch timer*/
              wdTime = 0;
     }
   }

 /**
  * System FSM
  *
  * @return Always returns <code>SUCCESS</code>
  **/
 event result_t Timer.fired()
 {
   result_t result;
   uint8_t tempState;

   atomic tempState = state;

   switch (tempState) {
   case PHASE1 :  /* Sending Commitment */
```

```
                   if (! call rfTransceiver.isSenderBusy())
                   {
                         /* send msg */
                         call rfTransceiver.Send(&msg);
 /* wait until a response to bring into 2nd phase*/
                         tempState = PHASE1_WAIT;
                   }
                   break;
    case PHASE1_WAIT:
           /* phase 1 wait state */
           break;
    case PHASE2 : /* Receive NA */
              if (evaluateWatchdog())
  {
      tempState = PHASE_START;
  post WatchdogFired();
  }
                  break;

    case PHASE3 : /* Send NA xor NB by ultrasound */
                  tempState = PHASE4;
                  break;
    case PHASE4 : /* Sending message in clear text by radio */
                 call Leds.greenToggle();
                 /* change role to receiver */
                 call rfTransceiver.stop();
                 call rfTransceiver.setRole(RF_SENDER);
                 call rfTransceiver.start();
                 /* Send message */
                 memcpy(msg.data,myText[textID],RF_DATA_LENGTH);
                 call rfTransceiver.Send(&msg);

                 textID ++;
                 if (textID == 4)
                    textID = 0;
                 tempState = PHASE_START;
                 break;
    case PHASE_IDLE:
           break;

    case PHASE_START :
           //compute commitment
            post compute_cipher();
            atomic idxNA = 0;
            tempState = PHASE_IDLE;
            break;
    }

atomic state = tempState;
    return SUCCESS;
  }

  /**
   * Signaled when RF package send done
   * @param rmsg Message pointer
   * @param success state of the transmission
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {
     uint8_t tempState;
     atomic tempState = state;
   /* the message from phase 1 is sent*/
       if (tempState == PHASE1_WAIT) {
       /* change role to receiver */
       call rfTransceiver.stop();
       call rfTransceiver.setRole(RF_RECEIVER);
       call rfTransceiver.start();
       call Leds.greenToggle();
   /* bring the system into phase 2, distance bounding */
       tempState = PHASE2;
      }
  // state transition
  atomic state = tempState;

return SUCCESS;
  }


  /**
   * Signaled when a new data package being received by Radio.
   * @param data New data package being received
   * @return Always returns <code>SUCCESS</code>
   **/
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {

     if (idxNA == (TINYSEC_KEYSIZE*8-1)) {
    /* bring the system into phase 3 to end distance bounding */
        atomic state = PHASE3;
      }else
    /* continue distance bounding */
        atomic idxNA++;

 /* Reset watch dog timer */
     resetWatchdog();
    return data;
  }

  /**
   * Signaled when message sent over ultrasound transmitter.
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t usTransceiver.SendDone() {
   call Leds.redToggle();
    return SUCCESS;
  }

  /**
   * Signaled when synchronization bit is detected in Ultrasound transciver
   * @param timeStamp Time Stamp when sync signal is detected, this value is uncorrected.
   *                 This value may contains hardware overhead.
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint16_t usTransceiver.msgDetected(uint16_t timeStamp){
   return timeStamp;
  }


  /**
   * Signaled when entire data byte is received from Ultrasound transceiver.
   * @param Rx_data   The data being received.
   * @param timeStamp Time Stamp entire data word is received,
   *                 this value is uncorrected.
   *                 This value may contains hardware overhead.
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint32_t usTransceiver.DataReceived(uint32_t Rx_data , uint16_t timeStamp){
    return Rx_data;
  }

  norace uint8_t po;
  norace uint8_t count;
  task void reportCNT() {
   printf("%x, cnt:%u, ",count,po);
  }

  async event void RadioSendCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {}
  async event void RadioSendCoord.blockTimer() {}
  async event void RadioSendCoord.byte(TOS_MsgPtr pmsg, uint8_t cnt){}

  async event void RadioReceiveCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {};
  async event void RadioReceiveCoord.blockTimer() {};
  async event void RadioReceiveCoord.byte(TOS_MsgPtr pmsg, uint8_t cnt)
  {
    static uint8_t i = 0;
    uint8_t tempState;
    // cnt == 6
    if (cnt == (offsetof(struct TOS_Msg,type) +
                  sizeof(((struct TOS_Msg*)0)->type) +3)){

         atomic tempState = state;
         ///call Leds.yellowToggle();
     /* Phase 2: distance bounding in place */
        if (tempState == PHASE2) {
             i =(MACKey[idxNA/8] >>(idxNA%8)) & 0x1;

 /* combined with response */
             if (pmsg->data[0] != 0)
                 i = (1^i);

 /* Get RF receive bit offset */
             rf_offset = call GetRxBitOffset();
             rf_offset *=2;
 i|=rf_offset;
             call usTransceiver.SendWord(i);

        }
   }
  }

  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }

}
```

PtlSender.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


/* Radio transciver */
includes rfTransceiver;
/* Ultrasound transciver */
includes usTransceiver;
/* Cryptogrohpic primitives for MAC*/
includes CryptoPrimitives;
/* Protocol */
includes PtlSender;
includes MyMessages;
```

```
configuration PtlSender {

}
implementation {
/*
 * Use of components
 */
  /* Application components */
  components Main, PtlSenderM,TimerC, LedsC;
  /* RF */
  components rfTransceiverC;
  /* Encryption */
  components CBCMAC as MAC;
  //components RC5M as Cipher;
  components SkipJackM as Cipher;
  /* Ultrasound */
  components usTransceiverC;
  /* Debug by RS232 */
  components RS232C;

/*
 *  Wire of components
 */
  /* Application control */
  Main.StdControl  -> PtlSenderM;
  Main.StdControl  -> TimerC;
  PtlSenderM.Timer -> TimerC.Timer[unique("Timer")];
  PtlSenderM.Leds  -> LedsC;

  /* Radio transceiver and coordinate */
  PtlSenderM.rfTransceiver     -> rfTransceiverC.rfTransceiver;
  PtlSenderM.RadioSendCoord    -> rfTransceiverC.RadioSendCoord;
  PtlSenderM.RadioReceiveCoord -> rfTransceiverC.RadioReceiveCoord;
  PtlSenderM.GetRxBitOffset    -> rfTransceiverC.GetRxBitOffset;

  /* MAC */
  PtlSenderM.MAC      -> MAC.MAC;
  MAC.BlockCipher     -> Cipher.BlockCipher;
  MAC.BlockCipherInfo -> Cipher.BlockCipherInfo;

  // Ultrasound
  PtlSenderM.usTransceiver -> usTransceiverC.usTransceiver;

  /* RS232 */
  PtlSenderM.RS232 -> RS232C.RS232;
}
```

PtlSender.h

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


#ifndef _PROTOCOLSENDER_H
#define _PROTOCOLSENDER_H

#include <stdio.h>

/*
```

```
 * Radio package setup
 */
#define RF_DATA_LENGTH 29
#define SYNC 0x16


#define MACSIZE 8

/*
 * System finite state machine
 *     Phase         Work
 *       0         Start, computing cipher
 *       1         Sending cipher
 *       2         Receiving Nounce Na
 *       3         Sending Na xor Nb
 *       4         Sending message in clear text
 *      10         Waiting task to complete
 */
#define PHASE_START 0
#define PHASE1 1
#define PHASE1_WAIT 10
#define PHASE2 2
#define PHASE3 3
#define PHASE4 4
#define PHASE_IDLE  11

/* System response time, controls FSM, in ms */
#define SYSTEM_REACTION_RATE 1000
/* watchdog timer value, unit is response time */
#define WATCHDOGTIME 7

#endif
```

MyMessages.h

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/
#ifndef _MYMESSAGES_H
#define _MYMESSAGES_H
#define TEXT {"This is test string 1","This Is Test String 2",
"Computer System Engineering","Computer Science Engineering"}
#endif
```

.tinyos_keyfile

```
# TinySec Keyfile. By default, the first key will be used.
# You can import other keys by appending them to the file.

default 35C9991B589AF29F168D7676FCE61559
```

PtlReceiverM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     PtlReceiverM.nc,
 */

module PtlReceiverM {
 provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer as FSMtimer;
    interface rfTransceiver;
    interface usTransceiver;
    interface MAC;
    interface RS232;
    interface RadioCoordinator as RadioSendCoord;
    interface RadioCoordinator as RadioReceiveCoord;
  }
}
implementation {
/**
 * System variables
 **/
   /* Protocol state */
   uint8_t state;
   /* Message in clear text */
   uint8_t TextInClear[30];
   /* the commitment */
   uint8_t cipher[30];
   /* computed cipher text */
   uint8_t computedCipher[30];
   /* Integrity region */
   uint16_t sendT;
   uint16_t backT;
   uint16_t integrityTime;
   uint16_t integrityDistance;

/**
 * Radio
 **/
   /* RF package */
   TOS_Msg   msg;
   /* RF receive bit offset */
   uint8_t rf_offset;

/**
 * MAC variables
 **/
   /* MAC context */
   MACContext macContext;
   /* Keys */
   uint8_t key_tmp[2*TINYSEC_KEYSIZE] = {TINYSEC_KEY};
   /* random nounce */
   uint8_t NA[TINYSEC_KEYSIZE];
   uint8_t KeyReceived[TINYSEC_KEYSIZE];
```

```
/**
 *  Phase 2 and 3, distance bounding protocol
 **/
  /* index of NA to be sent */
  uint8_t idxNA;
  uint8_t curData;

/**
 *   System watchdog timer
 **/
   uint8_t wdTime;

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    int i;
    /* init LEDs */
    call Leds.init();
    /* init RF transceiver set as receiver */
    call rfTransceiver.init();
    call rfTransceiver.setRole(RF_RECEIVER);
    /* US receiver, set role as receiver */
    call usTransceiver.init();
    call usTransceiver.setRole(US_RECEIVER);
    call usTransceiver.setGain(US_GAIN_NORMAL);
    /* RS232 */
    call RS232.init();

    /* MAC, copy the random nounce */
    memcpy(NA,key_tmp+TINYSEC_KEYSIZE,TINYSEC_KEYSIZE);

    /* format RF message */
/* set message length, address to broadcast*/
    msg.length = RF_DATA_LENGTH;
    msg.type = SYN1;
    msg.addr = TOS_BCAST_ADDR;
    for(i=0;i<msg.length;i++)
        msg.data[i] = 0x00;

    /* Initial system state */
    state = PHASE_START;

    /* Initial NA index*/
    idxNA = 0;

/* Initial watch timer*/
wdTime = 0;
    return SUCCESS;
  }


  /**
   *  Start system FSM, start RF receiver
   *  Bring the system ready for 1st phase
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    /* Start FSM */
    call FSMtimer.start(TIMER_REPEAT, SYSTEM_REACTION_RATE);
    /* start the RF receiver */
    call rfTransceiver.start();

    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    /* stop everything */
    call rfTransceiver.stop();
    call usTransceiver.stop();
    call FSMtimer.stop();
    return SUCCESS;
  }


  /* */
  task void report_phase2() {
    char buf[80];
    int msglen = 0;
    int i;
    // Print out received value
     msglen += sprintf(buf,"Sending NA:");
      for (i=0;i<TINYSEC_KEYSIZE;i++)
           msglen += sprintf(buf + msglen,"%x_",NA[i]);
     sprintf(buf+msglen,"\r\n");
     printf(buf);


  }

 /**
  *  Watchdog
  **/
  uint8_t evaluateWatchdog() {
      uint8_t t;
  uint8_t result;
  atomic {
     wdTime++;
 t = wdTime;
  }

  if (t >= WATCHDOGTIME)
     {  /* time out */
     result = 1;
 atomic wdTime = 0;
     }
  else
     result = 0;
  return result;
  }

 /**
  *  reset Watchdog
  **/
  void resetWatchdog() {
      atomic wdTime = 0;
  }

 /**
  *  Watchdog time out
  **/
  task void WatchdogFired() {
      int i;
      printf("Session time out! Key establishment failed, restart...\r\n");
  atomic {
       /* Reinitialize the system */

   /* init RF transceiver set as receiver */
   call rfTransceiver.stop();
               call rfTransceiver.setRole(RF_RECEIVER);
   call rfTransceiver.start();

  /* format RF message */
        /* set message length, address to broadcast*/
        msg.length = RF_DATA_LENGTH;
        msg.type = SYN1;
        msg.addr = TOS_BCAST_ADDR;
        for(i=0;i<msg.length;i++)
            msg.data[i] = 0x00;

        /* Initial NA index*/
        idxNA = 0;

        /* Initial watch timer*/
        wdTime = 0;
  }
  }
  /**
   * System FSM
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t FSMtimer.fired()
  {
   uint8_t tState;

atomic tState = state;
    switch(tState) {
     case PHASE1 :/* Waiting for commitment  */
                   break;
     case PHASE2 : /* initiating distance bouding */
                    /* Send NA first bit by radio */
                   msg.data[0] = (NA[idxNA/8] >>(idxNA%8)) & 0x1;
                   call rfTransceiver.Send(&msg);
   tState = PHASE3;
                   break;
     case PHASE3 :/* Distance bounding in progress */
               /* Evaluate watchdog timer, restart when time out */
  if (evaluateWatchdog())
      {
     tState = PHASE_START;
 post WatchdogFired();
  }
                  break;
     case PHASE4 :/* Receive message in clear text */
                  break;
     case PHASE_WAIT :
                  break;
     default :
              /* bring the system into phase 1*/
                  tState = PHASE1;
                  idxNA = 0;
                  printf("========Ready for commitment=========\r\n");
   }
   /* FSM state transition*/
    atomic state = tState;
    return SUCCESS;
  }

  task void report_sendtime() {
     printf("Sent Time: %u us\r\n",sendT);
  }

  /**
   * Signaled when RF package send done
   * No action taken in this implementation.
   * @param rmsg Message pointer
   * @param success state of the transmission
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {
return SUCCESS;
  }


  /**
   *   In phase 1, show the commitment received
   **/
  task void report_RFphase1() {
         char buf[80];
         int msglen = 0;
         int i;
         // Print out received value
         msglen += sprintf(buf,"Commitment received:");
          for (i=0;i<MACSIZE;i++)
          msglen += sprintf(buf + msglen,"%x_",cipher[i]);
          sprintf(buf+msglen,"\r\n");
          printf(buf);
  }

  /**
   *   In phase 4, show the summary
   **/
  task void report_phase4() {
    int i;
    char buf[80];
    int msglen = 0;
    uint8_t match = 1;
          // Print out received value

          sprintf(buf,"Message in clear text: %s\r\n",TextInClear);
          call RS232.SendString(buf);

          msglen = sprintf(buf,"MAC key of B is: ");
          for (i=0;i<TINYSEC_KEYSIZE;i++)
              msglen += sprintf(buf + msglen,"%x_",KeyReceived[i]);
  sprintf(buf+msglen,"\r\n");
          printf(buf);

  msglen = sprintf(buf,"Received MAC: ");
  for (i=0;i<MACSIZE;i++)
          msglen += sprintf(buf + msglen,"%x_",cipher[i]);
          sprintf(buf+msglen,"\r\n");
          printf(buf);

  msglen = sprintf(buf,"Computed MAC: ");
          for (i=0;i<MACSIZE;i++) {
              msglen += sprintf(buf + msglen,"%x_",computedCipher[i]);
  if (cipher[i]!=computedCipher[i])
     match = 0;
      }
```

```
        sprintf(buf+msglen,"\r\n");
        printf(buf);

 if (match)
    printf("Key established!\r\n");
     else
    printf("Key establishment failed!!\r\n");
 }
 /**
  * Signaled when commitment package being received by Radio
  * @param data data package being received
  * @return Always returns <code>SUCCESS</code>
  **/
 event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {

   uint8_t tempState;
   uint8_t i;

   atomic tempState = state;
   /* Phase 1, commitment received */
   if (tempState == PHASE1){
       /* copy commitment text */
        memcpy(cipher,data->data,MACSIZE);
        /* report the commitment */
        post report_RFphase1();
        /* Radio transceiver turn over */
        /* wait for 10ms in order to make Bob change into receiver */
        TOSH_uwait(10000);
        /* change to RF sender */
        call rfTransceiver.stop();
        call rfTransceiver.setRole(RF_SENDER);
        call rfTransceiver.start();
  /* state transition */
        tempState = PHASE2;
     }
   /* Phase 4, */
   if (tempState == PHASE4) {
        printf("NA xor NB: ");
          for (i=0;i<TINYSEC_KEYSIZE;i++) {
              printf("%x_",KeyReceived[i]);
              KeyReceived[i] ^=NA[i];

          }
          printf("\r\n");
         /* copy clear text message */
          memcpy(TextInClear,data->data,RF_DATA_LENGTH);
         /* compute h(m||Ns) */
         call MAC.init(&macContext,TINYSEC_KEYSIZE,KeyReceived);
         call MAC.MAC(&macContext,
                      (uint8_t*) data->data, strlen(data->data),
          computedCipher ,MACSIZE);
         tempState = PHASE_START;
         post report_phase4();
    }
   /* State transition */
   atomic state = tempState;
   return data;
 }

/**
  * Signaled when message come from serial port.
  * No action taken in this implementation.
  * @param buf data buffer
  * @param data_len length
  * @return Always returns <code>SUCCESS</code>
```

```
   **/
  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }

  /**
   * Signaled when a message sent over Ultrasound transmitter
   * No action taken in this implementation.
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t usTransceiver.SendDone(){
    return SUCCESS;
  }

    /**
     *   Report the decommit response time, uncorrected.
     **/
    task void report_backTime() {
      printf("bt:%uus, ",backT);
  }

 /**
   * Signaled when synchronization bit is detected in Ultrasound transciver
   * @param timeStamp Time Stamp when sync signal is detected, this value is uncorrected.
   *                  This value may contains hardware overhead.
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint16_t usTransceiver.msgDetected(uint16_t timeStamp){
   atomic backT = timeStamp;
   post report_backTime();
   return timeStamp;
  }

 uint8_t curBit;
/**
 *   report the decommit bit received, and the bit index
 **/
 task void reportUSdata() {
     printf("%u @%u, ",curData,curBit);
 }
/**
 *  Calculate Integrity region
 **/
 task void CalculateIntegrityRegion(){
   uint16_t truetime;
   uint16_t distance;
   uint16_t rawtime;
   uint16_t bitoffset;

   atomic {
   rawtime = backT - sendT;
   bitoffset = rf_offset;
   }
   if ((bitoffset <=7) && rawtime != 0){
         /* the corrected time */
      truetime = rawtime - (bitoffset - 7) * RF_COMPENSATION- 900;
/* the diatance */
            distance = truetime/10 * US_SPEED;
            distance /=100;
      atomic {
       integrityTime = truetime;
       integrityDistance = distance;
        }
      }
  }
 /**
  *   Show the result for Integrity region of current bit
  **/
  task void report_integrity(){
    if (rf_offset<=7) {
     printf("ofst:%u, ",rf_offset);
        printf("t:%uus, D:%ucm\r\n",integrityTime,integrityDistance);
    }
    else{
     printf("!ofst:%u, t:%uus, D:%ucm\r\n",rf_offset,integrityTime,integrityDistance);
    }
  }

 /**
   * Signaled when entire data byte is received from Ultrasound transceiver.
   * @param Rx_data   The data being received.
   * @param timeStamp Time Stamp entire data word is received,
   *                  this value is uncorrected.
   *                  This value may contains hardware overhead.
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint32_t usTransceiver.DataReceived(uint32_t Rx_data , uint16_t timeStamp){

    uint8_t tempState;
    static uint8_t ti =0;

    atomic {
      tempState = state;
      curData = Rx_data&0x1;
      curBit = idxNA;
  rf_offset = Rx_data;
  rf_offset = rf_offset/2;
   }
   /* Reset watch dog timer */
    resetWatchdog();

    post reportUSdata();
    post CalculateIntegrityRegion();
    post report_integrity();

    if (tempState == PHASE3) {
     /* Record current diatance bounding response */
        if (curData == 0)
            KeyReceived[idxNA/8] = KeyReceived[idxNA/8] & (0xff - (uint8_t)(pow(2,(idxNA%8))));
        else
            KeyReceived[idxNA/8] |=(uint8_t)( (uint8_t)(pow(2,(idxNA%8)))?pow(2,(idxNA%8)):1);

        if (idxNA == (TINYSEC_KEYSIZE*8 -1)) {
          /* Enough response received */
          /* After Phase2, NA is sent, Change to RF receiver */
            call rfTransceiver.stop();
            call rfTransceiver.setRole(RF_RECEIVER);
            call rfTransceiver.start();
  /* when enough bit received, go to phase 4 */
            tempState = PHASE4;
         }
         else {
          /* Send next NA challenge bit over radio */
            idxNA++;
            msg.data[0] = (NA[idxNA/8] >>(idxNA%8)) & 0x1;
            call rfTransceiver.Send(&msg);
         }
    }
/* state transition */
atomic state = tempState;
    return SUCCESS;
  }


  async event void RadioSendCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {}
  async event void RadioSendCoord.blockTimer() {}
  async event void RadioSendCoord.byte(TOS_MsgPtr tmsg, uint8_t cnt)
  {
    /* Synchronized at the first data byte */
      // cnt == 8
   if (cnt == (offsetof(struct TOS_Msg,data) +3)){
    /* start US receiver to reset the watch*/
    call usTransceiver.start();
    }
  }

  async event void RadioReceiveCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {};
  async event void RadioReceiveCoord.blockTimer() {};
  async event void RadioReceiveCoord.byte(TOS_MsgPtr tmsg, uint8_t cnt){}

}
```

PtlReceiver.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


/* Radio transciver */
includes rfTransceiver;
/* Ultrasound transciver */
includes usTransceiver;
/* Cryptogrohpic primitives for MAC*/
includes CryptoPrimitives;
/* Protocol */
includes PtlReceiver;
includes MyMessages;

configuration PtlReceiver {

}
implementation {
/*
 * Use of components
 */
  /* Application components*/
  components Main, PtlReceiverM,TimerC, LedsC;
  /* RF */
  components rfTransceiverC;
  /* Encryption */
  components CBCMAC as MAC;
  //components RC5M as Cipher;
  components SkipJackM as Cipher;
  /* Debug by RS232 */
  components RS232C;
  /* Ultrasound */
  components usTransceiverC;

/*
```

```
 *  Wire of components
 */
  /* Application control */
  Main.StdControl       -> TimerC;
  Main.StdControl       -> PtlReceiverM;
  PtlReceiverM.FSMtimer -> TimerC.Timer[unique("Timer")];
  PtlReceiverM.Leds     -> LedsC;

  /* Radio transceiver and coordinate */
  PtlReceiverM.rfTransceiver     -> rfTransceiverC.rfTransceiver;
  PtlReceiverM.RadioSendCoord    -> rfTransceiverC.RadioSendCoord;
  PtlReceiverM.RadioReceiveCoord -> rfTransceiverC.RadioReceiveCoord;

  /* MAC */
  PtlReceiverM.MAC    -> MAC.MAC;
  MAC.BlockCipher     -> Cipher.BlockCipher;
  MAC.BlockCipherInfo -> Cipher.BlockCipherInfo;

  /* RS232 */
  PtlReceiverM.RS232 -> RS232C.RS232;

  /* Ultrasound */
  PtlReceiverM.usTransceiver -> usTransceiverC.usTransceiver;

}
```

PtlReceiver.h

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


#ifndef _PTLRECEIVER_H
#define _PTLRECEIVER_H

#include <stdio.h>
#include <math.h>

/*
 * Radio package setup
 */
#define RF_DATA_LENGTH 29
#define SYNC 0x16

/*
 * MAC parameters
 */
/* MAC key size, byte */
#define MACSIZE 8


/*
 * System finite state machine
 *     Phase         Work
 *       0         Idle
 *       1         Receiving cipher
 *       2         Sending Nounce Na, and start distance bounding
 *       3         distance dounding in place
 *       4         Receiving message in clear text
 */
```

```
#define PHASE_START 0
#define PHASE1 1
#define PHASE2 2
#define PHASE3 3
#define PHASE4 4
#define PHASE_WAIT 10
/* System response time, controls FSM, in ms */
#define SYSTEM_REACTION_RATE 1000
/* watchdog timer value, unit is response time */
#define WATCHDOGTIME 5

/*
 * Distance measurement
 */
/* Radio send receive offset in microsecond(us) */
#define RF_OFFSET 0
/* Ultrasound transmitter overhead, microsecond(us) */
#define US_SENDDELAY 0
/* Speed of sound mm/100us */
#define US_SPEED 34
/* The time a bit takes to travel over the radio
 * the radio monitor works at 20kbps            */
#define RF_COMPENSATION 48

#endif
```

MyMessages.h

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/
#ifndef _MYMESSAGES_H
#define _MYMESSAGES_H
#define TEXT {"This is test string 1","This Is Test String 2",
"Computer System Engineering","Computer Science Engineering"}
#endif
```

.tinyos_keyfile

```
# TinySec Keyfile. By default, the first key will be used.
# You can import other keys by appending them to the file.

default 49A28CB92F5A23E776D69EDACB52FA4E
```

G.9 The attacker

PtlSenderM.nc

```
/*
 * "Copyright (c) 2006 The Technical University of Denmark."
 * All rights reserved.
 *
 */

/*
 *
 * Authors:             Feng Kai <s030656@student.dtu.dk>
 *
 * Revision:     PtlSenderM.nc, 2006/02/27 22:00:00
 */


module PtlSenderM {
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Timer;
    interface rfTransceiver;
    interface usTransceiver;
    interface MAC;
    interface RadioCoordinator as RadioSendCoord;
    interface RadioCoordinator as RadioReceiveCoord;
    interface RS232;

    async command uint8_t GetRxBitOffset();
  }
}
implementation {

/*
 * System variables
 */
   /* Protocol state */
   uint8_t state;

/*
 * Radio
 */
   /* RF package */
   TOS_Msg   msg;
   /* RF receive bit offset */
   uint8_t rf_offset;
/*
 *MAC variables
 */
   /* MAC context */
   MACContext macContext;
   /* Keys */
   uint8_t key_tmp[2*TINYSEC_KEYSIZE] = {TINYSEC_KEY};
   uint8_t MACKey[TINYSEC_KEYSIZE];
   uint8_t KeyReceived[TINYSEC_KEYSIZE];

   /* Clear texts */
   char myText[4][30] = TEXT;
   int textID;

   /* NA index */
   uint8_t idxNA;

/* watch dog timer*/
```

```
   uint8_t wdTime;

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    int i;
    /* init LEDs */
    call Leds.init();
    /* init RF transceiver set as sender */
    call rfTransceiver.init();
    call rfTransceiver.setRole(RF_SENDER);
    /* US receiver, set role as sender */
    call usTransceiver.init();
    call usTransceiver.setRole(US_SENDER);

    /* MAC */
    memcpy(MACKey,key_tmp+TINYSEC_KEYSIZE,TINYSEC_KEYSIZE);
    call MAC.init(&macContext,TINYSEC_KEYSIZE,MACKey);

    /* RS232 */
    call RS232.init();

    /* format RF message */
    msg.length = RF_DATA_LENGTH;
    msg.type = SYN1;
    msg.addr = TOS_BCAST_ADDR;

    for(i=0;i<RF_DATA_LENGTH;i++)
        msg.data[i] = 0x00;

   atomic {
    /* Initial system state */
    state = PHASE_START;
    /* Text string ID */
    textID = 0;
    /* initialize NA index */
    idxNA = 0;
/* Initial watch timer*/
wdTime = 0;
}
    return SUCCESS;
  }


  /**
   *  Start system FSM, start RF sender and US sender
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    /* start the RF receiver */
    call rfTransceiver.start();
    call usTransceiver.start();
    /* start FSM */
    call Timer.start(TIMER_REPEAT, SYSTEM_REACTION_RATE);
    return SUCCESS;
  }

  /**
   * Halt execution of the application.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    /* stop everything */
    call rfTransceiver.stop();
    call usTransceiver.stop();
    call Timer.stop();
    return SUCCESS;
  }

 /**
  *  Commitment
  **/
  task void compute_cipher(){
      /* compute MAC */
      printf("\r\n");
       call MAC.MAC(&macContext,
                    (uint8_t*) myText[textID], strlen(myText[textID]),
                     msg.data ,MACSIZE);
      /* bring the system into PHASE 1*/
      atomic state = PHASE1;

  }

/**
  *  Watchdog
  **/
 uint8_t evaluateWatchdog() {
      uint8_t t;
  uint8_t result;
  atomic t = (++wdTime);
  if (t >= WATCHDOGTIME)
     {  /* time out */
     result = 1;
 atomic wdTime = 0;
     }
  else
     result = 0;
  return result;
  }

 /**
  *  reset watch dog
  **/
  void resetWatchdog() {
      atomic wdTime = 0;
  }

 /**
  *  watchdog time out
  **/
  task void WatchdogFired() {
      int i;
      printf("Session time out! Key establishment failed, restart...\r\n");
  atomic {
               /* init RF transceiver set as sender */
               call rfTransceiver.setRole(RF_SENDER);
               /* format RF message */
               msg.length = RF_DATA_LENGTH;
               msg.type = SYN1;
               msg.addr = TOS_BCAST_ADDR;

               for(i=0;i<RF_DATA_LENGTH;i++)
               msg.data[i] = 0x00;

               /* Text string ID */
```

```
              textID = 0;
              /* initialize NA index */
              idxNA = 0;
              /* Initial watch timer*/
              wdTime = 0;
      }
    }

  /**
   * System FSM
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
    result_t result;
    uint8_t tempState;

    atomic tempState = state;

    switch (tempState) {
    case PHASE1 :  /* Sending Commitment */
                  if (! call rfTransceiver.isSenderBusy())
                  {
                        /* send msg */
                        call rfTransceiver.Send(&msg);
 /* wait until a response to bring into 2nd phase*/
                        tempState = PHASE1_WAIT;
                  }
                  break;
    case PHASE1_WAIT:
           /* phase 1 wait state */
           break;
    case PHASE2 : /* Receive challenge */
              if (evaluateWatchdog())
  {
      tempState = PHASE_START;
  post WatchdogFired();
  }
                 break;

    case PHASE3 : /* Send (response = NA xor NB) by ultrasound */
                 tempState = PHASE4;
                 break;
    case PHASE4 : /* Sending message in clear text by radio */
                call Leds.greenToggle();
                /* change role to receiver */
                call rfTransceiver.stop();
                call rfTransceiver.setRole(RF_SENDER);
                call rfTransceiver.start();
                /* Send message */
                memcpy(msg.data,myText[textID],RF_DATA_LENGTH);
                call rfTransceiver.Send(&msg);

                textID ++;
                if (textID == 4)
                   textID = 0;
                tempState = PHASE_START;
                break;
    case PHASE_IDLE:
           break;

    case PHASE_START :
            //compute commitment
            post compute_cipher();
            atomic idxNA = 0;
            tempState = PHASE_IDLE;
            break;
    }

atomic state = tempState;
    return SUCCESS;
  }

  /**
   * Signaled when RF package send done
   * @param rmsg Message pointer
   * @param success state of the transmission
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t rfTransceiver.sendDone(TOS_MsgPtr rmsg, result_t success) {
     uint8_t tempState;
     atomic tempState = state;
       /* the message from phase 1 is sent*/
       if (tempState == PHASE1_WAIT) {
       /* change role to receiver */
       call rfTransceiver.stop();
       call rfTransceiver.setRole(RF_RECEIVER);
       call rfTransceiver.start();
       call Leds.greenToggle();
 /* bring the system into phase 2, distance bounding */
       tempState = PHASE2;
      }
  // state transition
  atomic state = tempState;

return SUCCESS;
  }


  /**
   * Signaled when a new data package being received by Radio.
   * @param data New data package being received
   * @return Always returns <code>SUCCESS</code>
   **/
  event TOS_MsgPtr rfTransceiver.receive(TOS_MsgPtr data) {

     if (idxNA == (TINYSEC_KEYSIZE*8-1)) {
    /* bring the system into phase 3 to end distance bounding */
        atomic state = PHASE3;
      }else
    /* continue distance bounding */
        atomic idxNA++;

 /* Reset watch dog timer */
     resetWatchdog();
    return data;
  }

  /**
   * Signaled when message sent over ultrasound transmitter.
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t usTransceiver.SendDone() {
   call Leds.redToggle();
    return SUCCESS;
  }

  /**
   * Signaled when synchronization bit is detected in Ultrasound transciver
   * @param timeStamp Time Stamp when sync signal is detected, this value is uncorrected.
   *                 This value may contains hardware overhead.
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint16_t usTransceiver.msgDetected(uint16_t timeStamp){
   return timeStamp;
  }


  /**
   * Signaled when entire data byte is received from Ultrasound transceiver.
   * @param Rx_data   The data being received.
   * @param timeStamp Time Stamp entire data word is received,
   *                 this value is uncorrected.
   *                 This value may contains hardware overhead.
   * @return Always returns <code>SUCCESS</code>
   **/
  event uint32_t usTransceiver.DataReceived(uint32_t Rx_data , uint16_t timeStamp){
    return Rx_data;
  }

  norace uint8_t po;
  norace uint8_t count;
  task void reportCNT() {
   printf("%x, cnt:%u, ",count,po);
  }

  async event void RadioSendCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {}
  async event void RadioSendCoord.blockTimer() {}
  async event void RadioSendCoord.byte(TOS_MsgPtr pmsg, uint8_t cnt){}

  async event void RadioReceiveCoord.startSymbol(uint8_t bitsPerBlock, uint8_t offset, TOS_MsgPtr msgBuff) {};
  async event void RadioReceiveCoord.blockTimer() {};
  async event void RadioReceiveCoord.byte(TOS_MsgPtr pmsg, uint8_t cnt)
  {
    static uint8_t i = 0;
    uint8_t tempState;

   /****
    *   Attack on Sync mechanism, 2 byte earlier + large bit offset
    ****/
    if (cnt == (offsetof(struct TOS_Msg,type) +
                  sizeof(((struct TOS_Msg*)0)->type) +3-2)){
         printf("cnt:%d\r\n",cnt);
         atomic tempState = state;

     /* Phase 2: distance bounding in place */
        if (tempState == PHASE2) {
             i =(MACKey[idxNA/8] >>(idxNA%8)) & 0x1;

 /* combined with response */
             if (pmsg->data[0] != 0)
                 i = (1^i);

 /* Get RF receive bit offset */
             //rf_offset = call GetRxBitOffset();
             rf_offset = 7;
             rf_offset *=2;
 i|=rf_offset;
             call usTransceiver.SendWord(i);
        }
   }
  }

  event result_t RS232.Receive(char * buf, uint8_t data_len){
    return SUCCESS;
  }

}
```

PtlSender.nc

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


/* Radio transciver */
includes rfTransceiver;
/* Ultrasound transciver */
includes usTransceiver;
/* Cryptogrohpic primitives for MAC*/
includes CryptoPrimitives;
/* Protocol */
includes PtlSender;
includes MyMessages;

configuration PtlSender {

}
implementation {
/*
 * Use of components
 */
  /* Application components */
  components Main, PtlSenderM,TimerC, LedsC;
  /* RF */
  components rfTransceiverC;
  /* Encryption */
  components CBCMAC as MAC;
  //components RC5M as Cipher;
  components SkipJackM as Cipher;
  /* Ultrasound */
  components usTransceiverC;
  /* Debug by RS232 */
  components RS232C;

/*
 *  Wire of components
 */
  /* Application control */
  Main.StdControl  -> PtlSenderM;
  Main.StdControl  -> TimerC;
  PtlSenderM.Timer -> TimerC.Timer[unique("Timer")];
  PtlSenderM.Leds  -> LedsC;

  /* Radio transceiver and coordinate */
  PtlSenderM.rfTransceiver     -> rfTransceiverC.rfTransceiver;
  PtlSenderM.RadioSendCoord    -> rfTransceiverC.RadioSendCoord;
  PtlSenderM.RadioReceiveCoord -> rfTransceiverC.RadioReceiveCoord;
  PtlSenderM.GetRxBitOffset    -> rfTransceiverC.GetRxBitOffset;

  /* MAC */
  PtlSenderM.MAC      -> MAC.MAC;
  MAC.BlockCipher     -> Cipher.BlockCipher;
  MAC.BlockCipherInfo -> Cipher.BlockCipherInfo;
  // Ultrasound
  PtlSenderM.usTransceiver -> usTransceiverC.usTransceiver;

  /* RS232 */
  PtlSenderM.RS232 -> RS232C.RS232;
}
```

PtlSender.h

```
/**
 * @author Feng Kai s030656@student.dtu.dk
 **/


#ifndef _PROTOCOLSENDER_H
#define _PROTOCOLSENDER_H

#include <stdio.h>

/*
 * Radio package setup
 */
#define RF_DATA_LENGTH 29
#define SYNC 0x16


#define MACSIZE 8

/*
 * System finite state machine
 *     Phase         Work
 *       0         Start, computing cipher
 *       1         Sending cipher
 *       2         Receiving Nounce Na
 *       3         Sending Na xor Nb
 *       4         Sending message in clear text
 *      10         Waiting task to complete
 */
#define PHASE_START 0
#define PHASE1 1
#define PHASE1_WAIT 10
#define PHASE2 2
#define PHASE3 3
#define PHASE4 4
#define PHASE_IDLE  11

/* System response time, controls FSM, in ms */
#define SYSTEM_REACTION_RATE 1000
/* watchdog timer value, unit is response time */
#define WATCHDOGTIME 7

#endif
```

MyMessages.h

```
/**
```

```
 * @author Feng Kai s030656@student.dtu.dk
 **/
#ifndef _MYMESSAGES_H
#define _MYMESSAGES_H
#define TEXT {"This is test string 1","This Is Test String 2",
"Computer System Engineering","Computer Science Engineering"}
#endif
```

.tinyos_keyfile

```
# TinySec Keyfile. By default, the first key will be used.
# You can import other keys by appending them to the file.

default 35C9991B589AF29F168D7676FCE61559
```