Traveling Incognito

(Volume 1 of 1)
Extended Abstract

User privacy, which concerns about compromising the identities of network users and tracking user’s movement and whereabouts, has become one of the big issues in network especially wireless networks along with the increased requirement of user mobility and sensitive data application.

An Anonymous Authenticated Key Exchange for Roaming (AAKE-R) which involves a user and two servers, namely a home server and a foreign server, is an authenticated key exchange scheme, which achieves user anonymity and user untraceability by keeping user's identity secret from any other third party except home server. This scheme also manages to carry out authenticated key exchange between the user and the foreign server with the contribution of home server, at the same time, makes the session key unobtainable for the home server to achieve the data privacy.

This project will provide an instance of AAKE-R protocol with the consideration of key-privacy, real-time run environment, and computational constraints. Moreover, a PC-based application will be built to verify the correctness and realistic of the protocol.
Acknowledgement

I am deeply indebted to my supervisor Prof. Deng for his able guidance, motivation and moral support throughout this work. It was he who led me into the beautiful cryptography world, who always guided me to move forward in the right direction.

I would like to express my heartfelt thanks to Dr. Wong for his kind acceptance to be my co-supervisor. He directed me step by step to go through this project. Each time I encountered a problem, he always gave me a prompt response after carefully reviewing the related references.

My gratitude also goes to Tommy Yang, a Mphil student in CS who gave me a lot of help in detail with his professional knowledge in Cryptography and past FYP experience.

I also want to give my special thanks to my friends, Hong, Josie, Lily, etc. They stayed with me throughout the whole FYP process and gave me lots of support.
Table of Content

1. Introduction .................................................................................................................. 6
   1.1 Background Overview ......................................................................................... 6
   1.2 AAKE-R .............................................................................................................. 8
   1.3 Project Scope ....................................................................................................... 9
   1.4 Report Organization ............................................................................................ 9

2. Background Information ............................................................................................. 10
   2.1 Public-Key Cryptography .................................................................................... 10
   2.2 RSA .................................................................................................................... 11
   2.3 RSA-OAEP ......................................................................................................... 12
   2.4 Hash Function ..................................................................................................... 14
   2.5 Secure Hash Algorithm (SHA-1) ......................................................................... 15
   2.6 MD5 Message Digest Algorithm .......................................................................... 15
   2.7 Message Authentication Code (MAC) .................................................................. 16
   2.8 HMAC .................................................................................................................. 17
   2.9 DES in CBC Mode (DES-CBC) ........................................................................... 18
   2.10 ElGamal Signature ............................................................................................ 19

3. AAKE-R ...................................................................................................................... 21
   3.1 Security Requirements ....................................................................................... 21
      3.1.1 Server Authentication .................................................................................. 21
      3.1.2 Subscription Validation ............................................................................... 22
      3.1.3 Key Establishment ....................................................................................... 22
      3.1.4 User Anonymity ......................................................................................... 23
      3.1.5 User Untraceability ...................................................................................... 23
   3.2 Authenticated Key Exchange (AKE) ..................................................................... 24
   3.3 Protocol Description ............................................................................................ 25
   3.4 Security Analysis .................................................................................................. 28

4. Instantiation of AAKE-R ........................................................................................... 30
   4.1 Protocol Description ............................................................................................ 30
   4.2 Security Analysis .................................................................................................. 34
   4.3 Real-Time Issue ................................................................................................... 37
      4.3.1 Communication Round Complexity ............................................................. 37
      4.3.2 Terminal Computational Ability ................................................................. 38
5. System Design

5.1 System Architecture
5.2 Process Description

6. Implementation

6.1 Implementation Language
6.2 Session Communication
   6.2.1 Socket
   6.2.2 MFC Windows Socket
   6.2.3 Session Communication
6.3 Cryptography Library
6.4 Parameter Setting
   6.4.1 Server Address and Port
   6.4.2 Known List
   6.4.3 ID of Ca, Sh, Sv
   6.4.4 Public and Private Key of Ca, Sh, Sv
6.5 Implementation Result
   6.5.1 User
   6.5.2 Foreign Server
   6.5.3 Home Server
   6.5.4 Communication Procedure
      6.5.4.1 Edit Address and Port
      6.5.4.2 Initiate Socket
      6.5.4.3 Rundown Demo
      6.5.4.4 Test Case
      6.5.4.5 Follow-up Steps
   6.5.5 Result Checking

7. Conclusion

7.1 Critical Review
7.2 Suggestions for Future Improvement

References

Appendix

1. System Environment Setting
2. Interim Report
1. Introduction

1.1 Background Overview

Development of computer networks and telecommunications systems has brought lot of convenience to a user; among which user mobility is one of the main features. In one scenario, a user originally subscribed to a network can travel to another network administered by a different operator and request for services from this network. This capability, named roaming, enables users to enjoy a much wider area of coverage without the limitation of its home server’s coverage.

A typical roaming scenario involves three parties: a roaming user, a foreign server and a home server. The roaming user, who is a subscriber of the home server, is now in a network administered by the foreign server. In cellular networks, roaming services are widely deployed. However, security issues especially the user privacy have become a big concern. The latest generation, 3GPP, is also urging roaming services to be provided with a more promising assurance on the privacy of mobile users. Foremost among them are user anonymity and untraceability.

Based on the general interest of roaming users, it is desirable to keep the users anonymous from eavesdroppers and even the foreign server when roaming unless the identity information becomes critical in some emergency situations or special applications (user anonymity). Tracking a mobile unit’s movements may also expose the identity of a roaming user. One single exposure of the identity of a user will lead to the exposure of all other sessions, both future and past, if all the roaming sessions corresponding to the user can be linked. Hence it is equally important to make sure that no one would be able to tell if two roaming sessions are corresponding to the same mobile unit or not (user untraceability).
Besides cellular networks, there are many other roaming networks sharing the same desires of user privacy. In [1], Ateniese, et al. gave two examples of existing roaming applications which prefer upgrades to have user anonymity and untraceability. One is the inter-bank ATM networks and the other is the credit card payment systems. Ideally, a user should not have to reveal anything to the serving network (i.e. the foreign server) other than the confirmation of his good standing with respect to his ATM card or credit card issued by his home server. However, current systems are having users given out their personal information inevitably. In addition to these systems, there are many other roaming networks emerging which share similar demands on user privacy. For example, hopping across meshed WLANs (Wireless Local Area Networks) administered by different individuals, joining and leaving various wireless ad hoc networks operated by different foreign operators, etc.

Besides user anonymity and untraceability, data confidentiality and authenticity are usually needed to protect communications between user and the foreign server, against adversaries, which include eavesdroppers, other users and network servers, and even the home server. It is reasonable when considering that services are provided by the foreign server to the user, not by the home server. For example, in the WLAN Roaming, when a user accesses the Internet through a foreign server, the user may not want his home server to know which network sites he is visiting. This is one of the key privacy issues that most current systems cannot provide a solution.
1.2 Anonymous and Authenticated Key Exchange for Roaming (AAKE-R)

AAKE-R is a protocol proposed by Dr. Wong, Tommy Yang, and Prof. Deng in Anonymous and Authenticated Key Exchange for Roaming Networks(v2.7)[2]. The protocol involves three parties: a user and two servers, namely a home server and a foreign server. The following five properties should be achieved simultaneously in one protocol run.

1. Server Authentication
   The user is sure about the identity of the foreign server.

2. Subscription Validation
   The foreign server is sure about the identity of the home server of the user.

3. Key Establishment
   The user and the foreign server establish a random session key, which is known only to them and is derived from contributions of both of them. In particular, the home server should not obtain the session key.

4. User Anonymity
   Besides the user himself and his home server, no one including the foreign server can tell the identity of the user.

5. User Untraceability
   Besides the user himself and his home server, no one including the foreign server is able to identify any previous protocol runs, which have the same user involved.
1.3 Project Scope

In this project, I am going to

1. Study AAKE-R protocol proposed by Dr. Wong, etc and other reference papers to identify the issue in wireless roaming.
2. Instantiate and improve the AAKE-R protocol by applying Boyd-Park’s idea on Authenticated Key Exchange (AKE).
3. Design and Implement an application involved three parties to demo the session communication described by AAKE-R.

1.4 Report Organization

The rest of the paper is organized as follows. Chapter 2 introduces some cryptosystems applied in this project. In Chapter 3, details of AAKE-R protocol proposed by Dr. Wong, etc are reviewed, including security requirements, concept of Authenticated Key Exchange (AKE), protocol description and security analysis. On top of that, an instantiation of AAKE-R established in this project is introduced in Chapter 4. This is followed by system design of the application (Chapter 5) and its implementation details (Chapter 6). The report is concluded in Chapter 7 with critical review and suggestions for extensions to the project.
2. Background Information

This chapter will cover the basic and essential cryptography concepts used in the protocol design. Most of the concepts are extracted from the book “Cryptography and Network Security” [3].

2.1 Public-Key Cryptography

Public-key/asymmetric cryptography involves in, for each agent $a$, the use of its associated pair of keys $<K_{UA}, K_{RA}>$:

- The public key $K_{UA}$ which is published under the user’s name in a “public directory” accessible for everyone to read, which can be used to encrypt messages
- The private-key $K_{RA}$ which is known only to the agent $a$, which can be used to decrypt messages

Two keys applied in the Public-Key Cryptography have such characteristics:

- It is computationally infeasible to find decryption key knowing only algorithm and encryption key.
- It is computationally easy to en/decrypt messages when the relevant (en/decrypt) key is known.
- Either of the two related keys can be used for encryption, with the other used for decryption (in some schemes).

Fig 2.1 Public-Key Cryptography
2.2 RSA

RSA was invented by Rivest, Shamir and Adleman at MIT in 1977. It is the best-known and widely used Public-Key Scheme. RSA can be used for encryption and decryption, as well as digital signature.

RSA Key Setup:
- Select two large primes at random p, q
- Compute system modulus N=p.q, note \( \varphi(N)=(p-1)(q-1) \)
- Select at random the encryption key e,
  - where \( 1<e<\varphi(N) \), \( \gcd(e, \varphi(N))=1 \)
- Find decryption key d
  - \( E.d=1 \mod \varphi(N) \) and \( 0\leq d \leq N \)
- Publish public encryption key: \( KU = \{e, N\} \)
- Keep secret private decryption key: \( KR = \{d, p, q\} \)

RSA Encryption/Decryption:
- To encrypt a message M, the sender:
  - Obtains public key of recipient \( KU = \{e, N\} \)
  - Computes: \( C = M^e \mod N \), where \( 0\leq M<N \)
- To decrypt the ciphertext C, the owner:
  - Uses its private key \( KR = \{d, p, q\} \)
  - Computes: \( M = C^d \mod N \)
- Note: the message M must be smaller than the modulus N (block if needed)

RSA Digital Signature
- Suppose the public key of A is \( \{N, e\} \) and the private key is \( \{d, p, q\} \)
- To Sign a message M, A:
  - Computes \( S = M^d \mod N \) as the digital signature of M
- To Verify the signature S of a message M, the receiver:
  - Computes \( S^e \) and Check if it is M
2.3 RSA-OAEP

The Optimal Asymmetric Encryption Padding (OAEP) scheme for RSA was proposed by Bellare and Rogaway in 1994 [4]. RSA-OAEP is intended to be both efficient and provably secure. RSA-OAEP is designed to encrypt only short messages—typically secret keys for symmetric encryption or MAC algorithms.

The following notation is used in the description of the encryption scheme:

- A’s RSA public key is \((n, e)\), and \(d\) is A’s corresponding private key. The integer \(n\) is \(k\) octets in length. (For example, if \(n\) is a 1024-bit modulus, then \(k=128\).)
- \(H\) is a hash function which has outputs that are \(l\) octets in length. (For example, \(H\) may be SHA-1, in which case \(l=20\).)
- \(G\) is the following mask generating function: if the inputs to \(G\) are an octet string \(s\) and a positive integer \(t < 2^{32}l\), then the output of \(G\) is an octet string of length \(t\) obtained by concatenating successive hash values \(H(s\|I)\), \(0 \leq i \leq \lfloor t/l \rfloor-1\), and deleting any extra rightmost octets if necessary (\(i\) is a 32-bit counter).
- \(P\) consists of some (optional) encoding parameters.
- A padding string consists of a string of 00 octets (possible empty) followed by the octet 01.

RSA-OAEP Encryption Scheme

\(B\) sends message \(m\) of length at most \(k-2-2l\) octets to \(A\).

**Encryption.** \(B\) does the following:

- Obtain an authentic copy of \(A\)’s public key \((n, e)\).
- Select a random seed \(s\) of length \(l\) octets.
- Apply the OAEP encoding operation (depicted in Figure 1) with inputs \(s, P\) and \(m\) to obtain an octet string \(M\) of length \(k\) octets.
- Compute \(c=M^d \mod n\).
- Send \(c\) to \(A\).
**Decryption.** A does the following:

- Check that $c \in [0, n – 1]$; if not then output “error” and stop.
- Use the private key $d$ to compute $M = c^d \mod n$.
- Apply the OAEP decoding operation (depicted in Figure 2) with input $M$ to obtain octet strings $X$, $Q$, $T$, $M$; here $X$ has octet length 1, $Q$ consists of the first $l$ octets of DB, $T$ is the first non-zero octet following $Q$, and $m$ consists of the octets to the right of $T$.
- Compute $Q' = H(P)$.
- If $Q \neq Q'$, or if $X \neq 00$, or if $T \neq 01$, then output “error” and stop.
- Output $M$. 

---

Fig 2.2 RSA-OAEP Encryption
Hash function in general is a public function that maps a message of any length into a fixed-length hash value. It can be used for message authentication. When a sender sends a message to a receiver, the hash value of the message will be appended to the message. Upon receiving the message, the receiver authenticates the message by re-computing and comparing the hash values.

In order to make a hash function work, some requirements are needed for the hash function:
- The hash value should be easy to compute in implementation
- The hash function should be one-way, which means for a hash function \( H \), given any \( h \), it is computationally infeasible to find \( x \) such that \( H(x) = h \)
- The hash function should be resistant to collisions, which means for a hash function \( H \), it is computationally infeasible to find any pair \( (x, y) \) such that \( H(y) = H(x) \)
2.5 Secure Hash Algorithm (SHA-1)

The Secure Hash Algorithm (SHA) was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993; a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as SHA-1. The actual standard document is entitles Secure Hash Standard. SHA is based on the MD4 algorithm and its design closely models MD4.

SHA-1 Logic:
1. Pad message so its length is 448 mod 512
2. Append a 64-bit length value to message
3. Initialise 5-word (160-bit) buffer (A,B,C,D,E) to
   (67452301,efcdab89,98badcfe,10325476,c3d2e1f0)
4. Process message in 16-word (512-bit) chunks:
   • Expand 16 words into 80 words by mixing & shifting
   • Use 4 rounds of 20 bit operations on message block & buffer
   • Add output to input to form new buffer value
5. Output hash value is the final buffer value

2.6 MD5 Message Digest Algorithm

The MD5 message-digest algorithm (RFC 1321) was developed by Ron Rivest (the “R” in the RSA [Rivest-Shamir-Adleman]) at MIT. It is the latest in a series of MD2, MD4. It produces a 128-bit hash value.

MD5 Logic:
1. Pad message so its length is 448 mod 512
2. Append a 64-bit length value to message
3. Initialise 4-word (128-bit) MD buffer (A,B,C,D)
4. Process message in 16-word (512-bit) blocks:
   • Use 4 rounds of 16 bit operations on message block & buffer
• Add output to buffer input to form new buffer value

5. Output hash value is the final buffer value

2.7 Message Authentication Code (MAC)

It is a prime necessity to check the integrity of information transmitted in communications. Mechanisms that provide such integrity checks based on a secret key are usually called message authentication code (MAC).

A MAC is a cryptographic checksum: \( MAC = C_k(M) \), where

- \( M \) = input message
- \( C \) = MAC function
- \( K \) = shared secret key between two communicating parties
- \( MAC \) = message authentication code

MAC condenses a variable-length message \( M \) to a fixed-sized authenticator with a secret key \( K \). It is a many-to-one function. Potentially many messages have the same MAC, but finding these needs to be very difficult.
Requirements for MACs:

1. Knowing a message and MAC, is infeasible to find another message with same MAC
2. MACs should be uniformly distributed
3. MAC should depend equally on all bits of the message

2.8 HMAC

HMAC is specified as Internet Standard RFC2104.

\[
\text{HMAC}_K = \text{Hash}[(K + \text{XOR opad}) || \text{Hash}[(K + \text{XOR ipad})||M]]
\]

, where

- \( K \) = secret key; if key length is greater than \( b \), the key is input to the hash function to produce an \( n \)-bit key;
- \( K^+ \) = \( K \) padded with zeros on the left so that the result is \( b \) bits in length
- \( \text{ipad} \) = 00110110 (36 in hexadecimal) repeated \( b/8 \) times
- \( \text{opad} \) = 01011100 (5C in hexadecimal) repeated \( b/8 \) times

Fig 2.5 HMAC

HMAC can be used in combination with any iterated cryptographic hash function, such as MD5 and SHA-1. In this project, MD5 is used.

HMAC-MD5 will produces a 128-bit authenticator value.
2.9 DES in CBC Mode (DES-CBC)

DES is the first official U.S. government cipher intended for commercial use.

DES operates on 64-bit blocks of plaintext. After an initial permutation the block is broken into right half and left half, each being 32 bits long. There are 16 rounds of identical operations, call Function f, in which data are combined with 16 keys of 64 bits, one for each round. After the 16th round the right and left halves are joined, and a final permutation (the inverse of the initial permutation) finishes the algorithm. Because DES’s operation is very repetitive, it is readily implementable in hardware, as well as software.

![Fig 2.4 DES Encryption/Decryption](image)

The DES algorithm is defined in FIPS PUB 46-1 [1], and is equivalent to the Data Encryption Algorithm (DEA) provided in ANSI X3.92-1981 [2].

Cipher Block Chaining (CBC) is one of the block cipher modes. Message is broken into blocks, which are linked together in the encryption operation, and each previous cipher blocks is chained with current plaintext block. An Initial Vector (IV) is used to start process:

\[
Ci = DESK1(Pi \ XOR \ Ci-1)
\]

\[
C0 = IV
\]
The CBC mode of operation of DES is defined in FIPS PUB 81 [3], and is equivalent to those provided in ANSI X3.106 [4] and in ISO IS 8372 [5].

With DES-CBC, message text is encrypted using the Data Encryption Standard (DES) algorithm in the Cipher Block Chaining (CBC) mode of operation. It indicates the use of this algorithm/mode combination.

2.10 ElGamal Signature [5]

ElGamal is based on the problem of Discrete Logarithms in the group \((\mathbb{Z}_p^*, \cdot)\). Let \(p\) be a prime and let \(\alpha\) be a primitive element in \(\mathbb{Z}_p^*\). The set of all plaintexts \(P = \mathbb{Z}_p^*\), the set of all ciphertexts \(C = \mathbb{Z}_p^* \times \mathbb{Z}_p^*\), and the set of all keys \(K = \{(p, \alpha, a, \beta) : \beta \equiv \alpha^a \mod p\}\). \(p, \alpha, \) and \(\beta\) make up the public key and the private key is \(p, \alpha, \) and \(a\). Alice chooses a secret random number \(k\) in \(\mathbb{Z}_{p-1}\), and given \(m\) as the plaintext to be encrypted we have:
Encryption: $e_k(m, k) = (y_1, y_2)$, where $y_1 = \alpha^k \mod p$ and $y_2 = (m \cdot \beta^k) \mod p$

Decryption: $m = d_k(y_1, y_2) = y_2 (y_1^{a^{-1}}) \mod p$, for all $y_1, y_2$ in $\mathbb{Z}_p^*$ (notice the multiplicative inverse $(y_1)$)

ElGamal Signature Scheme

The ElGamal signature scheme is as follows: Again let $p$ be a prime and let $\alpha$ be a primitive element in $\mathbb{Z}_p^*$. The set of all plaintexts $P = \mathbb{Z}_p^{p-1}$, the set of all ciphertexts $C = \mathbb{Z}_p^{p-1} \times \mathbb{Z}_p^{p-1}$, and the set of all keys $K = \{(p, \alpha, a, \beta) : \beta \equiv \alpha^a \mod p\}$. $p$, $\alpha$, and $\beta$ make up the public key and the private key is $p$, $\alpha$, and $a$. Alice chooses a secret random number $k$ in $\mathbb{Z}_p^{p-1}$, and given $m$ as the plaintext to be signed we have:

Signing: $\text{sig}_k(m, k) = (y_1, y_2)$, where $y_1 = \alpha^k \mod p$ and $y_2 = [(m - a \cdot y_1)(k^{-1})] \mod (p - 1)$

Verification: $\text{ver}_k(m, (y_1, y_2)) \leftrightarrow y_1 y_2 = \alpha^m \mod p$

Security

If the attacker can compute the value $a = \log_\alpha \beta$, then ElGamal signatures can be forged. As long as $p$ is chosen carefully and $\alpha$ is a primitive element modulo $p$, then solving the Discrete Logarithm problem in $\mathbb{Z}_p^*$ is infeasible.

Additionally, $k$ must be secret, only used once and random. As shown above, ElGamal can also be used for encryption as well, but the messages should be relatively small in size.
3. AAKE-R

This Chapter will introduce the details of AAKE-R scheme proposed by Dr. Duncan Wong, etc, which uses conventional authenticated key exchange protocols as building blocks. Security requirements, concept of Authenticated Key Exchange (AKE), protocol description and security analysis are reviewed.

3.1 Security Requirements

Before describing the five security requirements in details, the term ‘Subscription’ should be clearly defined. A user is subscribed to a server called the home server of the user if the server has the privilege to get access to the real identity of the user and track the user’s movements and whereabouts on networks, to which the user has visited, no matter those visited networks are administered by other servers or by itself.

3.1.1 Server Authentication

For a user Ca, he has to make sure that the foreign server Sv is the one he is connecting to. Server Authentication, providing an assurance to the user on the identity of the foreign server, is important in practice. For example, different foreign server may charge differently to the user. The user would like to choose and be sure that he is obtaining services provided by the intended foreign server. Another example, the user may trust some of the foreign servers but not the others. The user does not want to establish a connection with an untrusted foreign server and leak information to that server.

This requirement becomes increasingly important in wireless networks, as impersonation attacks are much easier to launch when compared with the wired networks.
3.1.2 Subscription Validation

For a foreign server Sv, it only requires to make sure the identity of the user’s home server Sh, that is, given Sh, Sv is to make sure that Ca is Sh’s subscribed user without knowing Ca (the identity of the user). This can be considered as a replacement of client authentication.

Since user anonymity and untraceability are needed, the identity of the user would not be exposed to the foreign server. For facilitating billing, access control or other subscription-oriented applications, subscription validation is needed. For example, a server has a security policy, which only allows users subscribed to one particular server to access its services but does not allow users subscribed to another particular server to do so. Another example, a foreign server may charge differently for users subscribed to different servers.

3.1.3 Key Establishment

In most applications, it does not seem to be very useful if a protocol only provides authentication but no key establishment. This is because an authentication-only protocol only provides authenticity between the intended parties when running the protocol. It does not provide any authentication after the protocol is finished.

An authenticated key establishment protocol, instead, allows two intended communicating parties to establish a secret session key, which is known only to the two parties. Hence after the protocol is finished, the two parties can use the session key to communicate securely and in an authenticated way.

The user Ca and the foreign server Sv establish a random session key, which is known only to them and is derived from contributions of both of them. In particular, Sh should not obtain session key.
3.1.4 User Anonymity

Besides the user Ca himself and his home server Sh, no one including the foreign server can tell the identity of the user.

3.1.5 User Untraceability

Besides the user Ca himself and his home server Sh, no one including the foreign server is able to identify any previous protocol runs, which have the same user involved.
3.2 Authenticated Key Exchange (AKE)

AKE is a two-party protocol, which allows the parties to authenticate each other and simultaneously come into possession of a shared session key. To describe one protocol run of an AKE, we need to specify the inputs of the two parties and the generation of the session key. In the following, we introduce a notation to describe an AKE protocol run for the asymmetric case. Let the identities of the two parties be A and B. Let the public key pairs of A and B be \((\hat{S}_A, P_A)\) and \((\hat{S}_B, P_B)\), respectively. Suppose the session key generated is \(\sigma\), the AKE protocol run is denoted by

\[
\sigma \leftarrow \text{AKE}(A, B, (\hat{S}_A, P_A), (\hat{S}_B, P_B))
\]

There is a special type of AKE schemes: authenticated key transport (AKT). The only difference between AKT and AKE is that the former has the session key prepared by one party and ‘transported’ securely to the other party. By using the symbols above, suppose the session key \(\sigma\) is prepared by A and is transported to B, the AKT protocol run is denoted by

\[
\text{AKT}(A, (\hat{S}_A, P_A)) \xrightarrow{\sigma} (B, (\hat{S}_B, P_B))
\]

Anonymous Authenticated Key Exchange

Some conventional AKE schemes can be converted to support one-party anonymity against eavesdroppers. This type of schemes allows one of the two parties to hide its identity in the messages exchanged so that no eavesdropper can identify whom that party Anonymous AKE for Roaming Networks is. Notice that these schemes are fundamentally different from AAKE-Rs. One major difference is that an AAKE-R scheme hides the user’s identity from the foreign server while an AKE scheme which supports one-party anonymity against eavesdroppers does not hide the user’s identity from the other communicating party.

To denote that A is hiding its identity, we use the notation below.

\[
\sigma \leftarrow \text{AAKE}(A, B, (\hat{S}_A, P_A), (\hat{S}_B, P_B))[A]
\]
3.3 Protocol Description

The protocol consists of three entities: user $C_a$, home server $S_h$, and foreign server $S_v$ where $S_h \neq S_v$. Assume that there is a direct link between $C_a$ and $S_v$ and another direct link between $S_v$ and $S_h$. But there is no direct link between $C_a$ and $S_h$. For all communications between $C_a$ and $S_h$, messages are relayed by $S_v$.

Let $(\hat{S}_v, P_v)$ be the public key pair of $S_v$ for some semantically secure public key encryption scheme $E$. We use $E_{P_v}$ to denote the encryption under the public key $P_v$. Let $(\hat{S}_a, P_a)$ and $(\hat{S}_h, P_h)$ be the public key pairs of $C_a$ and $S_h$, respectively. Assume that each user knows its home server’s public key and each server knows the public keys of all its subscribers. In practice, the user and his home server can send their public keys to each other when the user subscribes to the server in some registration phase. For simplicity, we also assume that all servers know the public keys of all other servers. In practice, this can be replaced by a certificate-based solution for providing scalability.

Let $H_1$, $H_2$ and $H_3$ be cryptographically strong hash functions. Each of them maps from $\{0, 1\}^*$ into $\{0, 1\}^k$. Below is the protocol description which is illustrated in Fig 3.1.

1. $C_a$ randomly generates $k_a$ belonging to $\mathbb{R}\{0, 1\}^k$ and sends $m_1 = E_{P_v}(S_h || H_1(k_a))$ to $S_v$. We assume that $S_v$ is broadcasting $P_v$ associated with a certificate and $C_a$ has obtained $P_v$ and checked its validity using the associated certificate before running the protocol.
2. $S_v$ decrypts $m_1$ using the private key $\hat{S}_v$ and separates it into two halves: the first $k$-bit binary string is $S_h$, and the second $k$-bit binary string should be $H_1(k_a)$. Here we denote it by $\alpha$. It then ‘informs’ (by sending a prespecified message) $S_h$ that there is a user who claims to be its subscriber.
3. $C_a$ and $S_h$ start up an AAKE run via $S_v$ and attain

$$c \leftarrow \text{AAKE}(C_a, S_h, (\hat{S}_a, P_a), (\hat{S}_h, P_h))[C_a]$$
if \( C_a \) is the subscriber of \( S_h \). Otherwise, both entities halt with failure. 
\( S_v \) will also halt with failure after being informed by \( C_a \) or \( S_h \) or timeout.

4. \( S_h \) computes \( \pi = H_2(C_a, S_h, S_v, c) \).

5. \( S_h \) and \( S_v \) start up an AKT run and attain

\[
\text{AKT}(S_h, (S_h, P_h)) \overset{\pi}{\longrightarrow} (S_v, (S_v, P_v))
\]

If the AKT fails to complete, both entities halt with failure. \( C_a \) will also halt with failure after being informed by \( S_v \) or timeout.

6. \( S_v \) randomly generates \( k_b \) belonging to \( R \{0, 1\}^k \) and sends \( m2 = \alpha \oplus k_b \) to \( C_a \).

7. \( C_a \) obtains \( k_b \) as \( m2 \oplus H_1(k_a) \) and sends \( m3 = H_1(k_b) \oplus k_a \) to \( S_v \).

8. \( S_v \) obtains \( k_a \) from \( m3 \) and checks if \( H_1(k_a) = \alpha \). If it is true, continue.

Otherwise, \( S_v \) rejects the connection and halts.

9. Each of \( C_a \) and \( S_v \) computes the session key \( \sigma = H_2(S_h, S_v, k_a, k_b, \pi) \) and jointly conduct the following key confirmation steps.

   a) \( S_v \) sends \( m4 = H_3(S_h, S_v, H_1(k_a), k_b, \pi) \) to \( C_a \).

   b) \( C_a \) checks if \( m4 = H_3(S_h, S_v, H_1(k_a), k_b, \pi) \). If it is true, \( C_a \) sends \( m5 = H_3(S_h, S_v, k_a, k_b, \pi) \) back to \( S_v \) and accepts the connection.

   Otherwise, \( C_a \) rejects and halts.

   c) \( S_v \) checks if \( m5 = H_3(S_h, S_v, k_a, k_b, \pi) \). If it is correct, \( S_v \) considers that the connection is established. Otherwise, \( S_v \) halts with failure.

The messages can be piggybacked in the last two message flows.

10. Both \( C_a \) and \( S_v \) destroy their copies of \( k_a \) and \( k_b \) after accepting the connection.
Fig 3.1. The AAKE-R Protocol

\[ C_a \quad S_v \quad S_h \]

\[ \text{Enc}_{P_v}(S_h \parallel H_1(k_a)) \]

\[ c \leftarrow \text{AAKE}(C_a, S_h, (\hat{S}_a, P_a), (\hat{S}_h, P_h))|C_a \]

\[ H_1(k_a) \text{XOR} k_b \parallel H_3(S_h, S_v, H_1(k_a), k_b, \Pi) \]

\[ H_1(k_b) \text{XOR} k_a \parallel H_3(S_h, S_v, k_a, k_b, \Pi) \]
3.4 Security Analysis

Server authentication is done by the following challenge-response pair:

\[(E_{\mathcal{P}_V} (S_h||H_1(k_a)), H_3(S_h, S_v, H_1(k_a), k_b, \pi))\]

Only \(S_v\), who has \(\hat{S}_v\), can obtain the value of \(H_1(k_a)\) from the first item of the pair, and the response containing the digest of \(H_1(k_a)\) lets \(C_a\) authenticate \(S_v\).

Subscription validation is done in three steps. First, \(S_h\) is involved to authenticate \(C_a\). This is done by using the AAKE mechanism. Second, \(S_h\) sends \(S_v\) a credential, which comprises the entire transcript of the AKT protocol run, for testifying that the user who has involved in the AAKE in the first step is a subscriber of \(S_h\). Third, \(S_v\) ensures that the user communicating with \(S_h\) in the first step is also the one who is currently communicating with. This is done by having the user send the last message component, \(H_3(S_h, S_v, k_a, k_b, \pi)\) to \(S_v\). Since besides \(S_h\) and \(S_v\), only the user who has communicated with \(S_h\) in the first step can compute the value of \(\pi\).

For key establishment, we will show that only \(C_a\) and \(S_v\) are sharing the fresh session key after one protocol run. First, only \(C_a\), \(S_v\) and \(S_h\) know the value of \(\pi\) and therefore only these three parties are able to compute the session key \(\sigma = H_2(S_h, S_v, k_a, k_b, \pi)\) if they also know \(k_a\) and \(k_b\). As in the subscription validation, we exclude the scenario that \(S_h\) is impersonating its own subscriber. So in the following, we only need to show that \(S_h\) cannot obtain at least \(k_b\) from the transcript of one protocol run. Notice that both \(H_1(k_a)\) XOR \(k_b\) and \(H_1(k_b)\) XOR \(k_a\) do not help get \(k_b\) since \(H_1(k_a)\) and \(H_1(k_b)\) are some unknown pseudorandom strings and no any bit information of \(k_a\) or \(k_b\) can be obtained from them. In addition, the first message flow does not leak any information of \(H_1(k_a)\) due to the semantic security property of the underlying encryption function \(E\). Hence \(S_h\) cannot obtain session key \(\sigma\) from the transcript of the protocol run. On key control, it can be seen that joint key control is achieved and no party can predetermine the value of the session key when generating their session key component.
In the protocol, we can see that besides AAKE and the value of $\pi$, there is no information related to the identity of $C_a$. Without knowing $C_a$ and $c$, which is the secret output of AAKE but not known to the adversary, $\pi$ is just the digest of two unknown values and does not help the adversary obtain any additional information of $C_a$. Therefore, the degree of user anonymity of the protocol reduces to that of the AAKE scheme. Similarly, user untraceability is also ensured by the security assumption of the underlying AAKE scheme.
4. Instantiation of AAKE-R

This Chapter will give a detailed description of an instantiation of AAKE-R established in this project, which extends the Authenticated Key Exchange (AKE) on top of the AAKE-R scheme introduced in Chapter 3. Security issues concerned in the extension will be analyzed.

4.1 Protocol Description

The protocol consists of three entities: user \( C_a \), home server \( S_h \), and foreign server \( S_v \) where \( S_h \neq S_v \). Assume that there is a direct link between \( C_a \) and \( S_v \) and another direct link between \( S_v \) and \( S_h \). But there is no direct link between \( C_a \) and \( S_h \). For all communications between \( C_a \) and \( S_h \), messages are relayed by \( S_v \).

Symbols used in this protocol:

Let \((\hat{S}_a, P_a)\), \((\hat{S}_v, P_v)\) and \((\hat{S}_h, P_h)\) be the public key pair of \( C_a \), \( S_v \) and \( S_h \) for public key encryption scheme OAEP[RSA], respectively. We assume that \( S_v \) is broadcasting \( P_v \) associated with a certificate; \( C_a \) and \( S_h \) have obtained \( P_v \) and checked its validity using the associated certificate before running the protocol. We use \( \text{Enc}_{P_v} \) to denote the encryption under the public key \( P_v \). We use \( \text{Sig}_{S_a} \) to denote the ElGamal-type signature under the secret key \( \hat{S}_a \).

Let \( C \) be the session key of \( C_a \) and \( S_h \), we use \( \text{MAC}_C \) (Message) to denote the MAC of message under the key \( C \) with DES-CBC scheme.

Let \( H_1, H_2, H_3, H_4, H_5, H_6, \) and \( H_7 \) be distinct hash functions. Each of them maps from \( \{0, 1\}^* \) into \( \{0, 1\}^k \).

Let \( r_a, r_v, r_h, k_a, \) and \( k_v \) be the random generated strings belonging to \( \mathbb{R} \{0,1\}^k \).
Below is the protocol description, which is illustrated in Fig 4.1.

1. $C_a$ randomly generates $k_a$, $r_a$, and sends $m_1 = Enc_{Pv}(S_h || H_1(k_a)) || Enc_{Ph}(C_a || r_a || S_v || Sig_{Sv}(r_a || C_a || S_v || S_h))$ to $S_v$.

2. On receiving $m_1$, $S_v$ decrypts the first part $Enc_{Pv}(S_h || H_1(k_a))$ using the private key $\dot{S}_v$ to obtain $S_h$ and $H_1(k_a)$. Here we denote $H_1(k_a)$ by $\alpha$. It then informs $S_h$ that there is a user who claims to be its subscriber by sending the second part $m_2 = Enc_{Ph}(C_a || r_a || S_v || Sig_{Sv}(r_a || C_a || S_v || S_h))$.

3. $S_h$ decrypts $m_2$ using the private key $\dot{S}_h$ and separates it into four halves: the first $k$-bit binary string is $C_a$, the second $k$-bit binary string is $r_a$, the third $k$-bit binary string is $S_v$, and the fourth part is $Sig_{Sv}(r_a || C_a || S_v || S_h)$. It then verifies the signature. If verification is not successful, $S_h$ rejects the connection and halts. Otherwise, $S_h$ randomly generates $r_h$, computes $C = H_2(r_a || r_h)$, $\Pi = H_3(C_a || S_h || S_v || C)$, and then sends $m_3 = r_h || MAC_C(H_4(r_a || C_a || S_v || S_h)) || Enc_{Pv}(\Pi)$ to $S_v$.

4. $S_v$ obtains $Enc_{Pv}(\Pi)$ from $m_3$, decrypts it using the private key $\dot{S}_v$ to get $\Pi$, and then forwards $m_4 = r_h || MAC_C(H_4(r_a || C_a || S_v || S_h))$ to $C_a$.

5. $C_a$ obtains first $k$-bit binary string $r_h$ from $m_4$, computes $C = H_2(r_a || r_h)$ and $H_4(r_a || C_a || S_v || S_h)$, then performs MAC function of $H_4$ under the key $C$. The result should be the same as the second $k$-bit binary string of $m_4$. If they don’t match, $C_a$ rejects the connection and halts. Otherwise, it sends $m_5 = [Sig_{Sa}(H_5(r_h || C || C_a || S_v || S_h)) || Padding] XOR G(C)$ to $S_v$. Here, fixed length strategy is used to achieve the key privacy of $Sig_{Sa}$.

6. $S_v$ appends $r_v || MAC_{\Pi}(r_v || S_v || S_h)$ to $m_5$, and sends $m_6 = [Sig_{Sa}(H_5(r_h || C || C_a || S_v || S_h)) || Padding] XOR G(C) || r_v || MAC_{\Pi}(r_v || S_v || S_h)$ to $S_h$.

7. $S_h$ separates $m_6$ into three parts, the first $n$-bit binary string is $[Sig_{Sa}(H_5(r_h || C || C_a || S_v || S_h)) || Padding] XOR G(C)$, here we denote it as $\beta$,
the second k-bit binary string is $r_v$, and the third k-bit binary string is

$\tau = \text{MAC}_\Pi(r_v||S_v||S_h)$. $S_h$ computes $\beta \ XOR G(C)$, removes the padding to get $\chi = \text{Sig}_{S_h}(H_5(r_h||C||C_a||S_v||S_h))$. It then verifies the signature $\chi$ and checks if $\text{MAC}_\Pi(r_v||S_v||S_h) = \tau$. If successful, $S_h$ sends $m_7 = \text{Sig}_{S_h}(H_6(\Pi||r_v||S_v||S_h))$ to $S_v$, otherwise, it rejects the connection and halts.

8. $S_v$ verifies $m_7$ using the public key $P_h$, the result should be $H_6(\Pi||r_v||S_v||S_h)$. If true, $S_v$ randomly generates $k_v$, and sends $m_8 = H_1(k_a)XOR k_v||H_7(S_h||S_v||H_1(k_a)||k_v||\Pi)$ to $C_a$. Otherwise, $S_v$ rejects the connection and halts.

9. $C_a$ separates $m_8$ into two parts, the first k-bit string is $H_1(k_a)XOR k_v$, here we denote it as $\mu$. $C_a$ obtains $k_v$ as $\mu \ XOR H_1(k_a)$, and check if the second part is $H_7(S_h||S_v||H_1(k_a)||k_v||\Pi)$. If so, it sends $m_9 = H_1(k_v)XOR k_a||H_7(S_h||S_v||k_a||k_v||\Pi)$ to $S_v$, computes the session key $\sigma$ as $H_8(S_h||S_v||k_a||k_v||\Pi)$, and accepts the connection. Otherwise, $C_a$ rejects and halts.

10. On receiving $m_9$, $S_v$ separates $m_9$ into two parts, the first k-bit string is $H_1(k_v)XOR k_a$, here we denote it as $\nu$. $C_a$ obtains $k_a$ as $\nu \ XOR H_1(k_v)$, and check if the second part is $H_7(S_h||S_v||k_a||k_v||\Pi)$. If so, computes the session key $\sigma$ as $H_8(S_h||S_v||k_a||k_v||\Pi)$, and consider the connection is accepted. Otherwise, $S_v$ halts with failure. Two distinguished hash functions are used: one for key confirmation ($H_7$) and the other for key generation ($H_8$). The value of $H_7$ in $m_8$ and $m_9$ can be considered as message authentication tags.

11. After completing the key confirmation steps, $C_a$ and $S_v$ should destroy $k_a$ and $k_v$ immediately.
Fig 4.1 Instance of AAKE-R Protocol
4.2 Security Analysis

In the theory of authentication and key establishment, Boyd and Park [6] provided two-version protocols: one is key transport protocol and the other key agreement protocol. Key transport occurs when one party in the protocol chooses the session key and sends it encrypted to the other party. In contrast, key agreement means both entities contribute to the session key. Their protocols were described in the following where $\text{Enc}_b(X)$ denotes encryption of the field $X$ using the public key of $B$.

**Key Transport Protocol:** A chooses and sends session key $K_{ab}$ to $B$

1. $A \rightarrow B$: $\text{Enc}_b(A, K_{ab}, \text{COUNT})$
2. $B \rightarrow A$: $\{\text{COUNT}, r_b\}_{K_{ab}}$
3. $A \rightarrow B$: $\text{Sig}_a(B, h(\text{COUNT}, K_{ab}, r_b))$

**Key Agreement Protocol:** both A and B contribute to the session key, calculated as $K_{ab} = h(r_a, r_b)$ for a suitable hash function $h$.

1. $A \rightarrow B$: $\text{Enc}_b(A, r_a, \text{COUNT})$
2. $B \rightarrow A$: $r_b, \{\text{COUNT}, r_a\}_{K_{ab}}$
3. $A \rightarrow B$: $\text{Sig}_a(B, h(\text{COUNT}, K_{ab}, r_b))$

Later, Newe and Coffey [7] made some modification of Boyd-Park Key Agreement Protocol by adding a signature of A to enable authentication of A by B. User anonymity is achieved by encrypting the message with the public key of B.

1. $A \rightarrow B$: $\text{Enc}_b(A, \text{Sig}_a(r_a, \text{COUNT}))$
2. $B \rightarrow A$: $r_b, \{\text{COUNT}, r_a\}_{K_{ab}}$
3. $A \rightarrow B$: $\text{Sig}_a(B, h(\text{COUNT}, K_{ab}, r_b))$

Basically, our protocol adopted Boyd and Park’s idea that using different key establishment schemes to achieve the authority and user anonymity. We used key agreement protocol to set up a session key $C$ between user and home server, then home server calculated session key $\Pi$ basing on $C$, and
sent it to foreign server by using key transport protocol. Finally, user and foreign server calculated session key $\sigma$ individually by using $\Pi$ and the security numbers randomly generated by user and foreign server. In this way, our protocol achieved user anonymity, data confidentiality and authenticity, which were explained in the previous chapter. Home server acted as a guarantor for giving a promise to foreign server that the user is a legitimate subscriber. User’s identity was anonymous to the foreign server. Moreover, data exchanged between the user and the foreign server was protected against the home server.

With some practical considerations and further security concerns, we have some modifications on Boyd-Park Key establishment protocols.

1. Use identities $C_a$, $S_v$, $S_h$ instead of COUNT:
   COUNT is quite difficult to implement in practice. Each party needs to maintain a table of COUNTs corresponding to different transactions with different partners, which waste lots of memory in distributing networks. Moreover, operation of COUNT is very complex. In a typical message-resending scenario, it is hard to decide whether the COUNT should remain the same or increase. If COUNT remains the same, receiver cannot easily distinguish the resent message with a replay attack. If receiver is designed to accept the message with increased COUNT only, then a masquerader can simply generate a COUNT large enough to cheat. Using identities to replace COUNT could escape the complex issues along with it. What’s more, identities can be used to label different participants in the transaction. It is a way of authentication combined with the encryption function.

2. Use MAC-CBC instead of the encryption of session key
   The traditional encryption scheme such as DES has a repeating leakage, which means same plaintext always comes out a same ciphertext over the time. It raises a security issue when sending a message several times. Moreover, if the length of each message component is the same as that of the
message block, leakage of private data is more significant. With MAC-CBC scheme, these considerations are not necessary.

3. Add padding after C_r’s signature to perform fixed length

Bellare et. al. [8] raised an observation that standard RSA encryption does not provide anonymity, even when all modules in the system have the same length. In all popular schemes, the ciphertext is (or contains) an element $y = x^e \mod N$ where $x$ is a random member of $\mathbb{Z}^*N$. Suppose an adversary knows that the ciphertext is created under one of two keys $N_0; e_0$ or $N_1; e_1$, and suppose $N_0 \leq N_1$. If $y \geq N_0$ then the adversary bets it was created under $N_1, e_1$; else it bets it was created under $N_0, e_0$. With this concern, we add padding after the signing message to achieve key-privacy.
4.3 Real-Time Issue

When a security model is applied in reality, many real-time issues should be concerned, such as the communication rounds of the protocol, the computational ability of the terminals, etc.

4.3.1 Communication Round Complexity

The obvious result of the multiple communication rounds is the communication delay especially in wide area networks (WANs), which is a big concern in today’s fast-speed world. Most of the current security models applied in wireless network have about 3~5 communication rounds only.

Another consequence caused by the complex communication rounds is the waste of bandwidth. Our protocol is on top of the existing TCP/IP layer, and hence, each message in the protocol will append a header to indicate the sender, receiver and other information. The header itself will occupy a lot of bandwidth resources. Considering with the real environment that lots of the users will try to build the communication with foreign server at the same time, the cost will be significant.

Apparently, the more communication rounds of a protocol, the more volatility it will expose to the adversary. An attacker may modify a message in transaction to terminate the communication between a valid user and server.

With all the considerations above, we put lots of effort in the reducing of the communication rounds, and combine the messages without inter-crossed data together, at the same time keep the messages in a logical order.

Fig 4.2 is the first version of protocol instantiation with 12 communication rounds:

1. m2 to m7 is the instantiation of AAKE running between C_a and S_h via S_v (see Chapter 2.3) :
   
   \[ c \leftarrow \text{AAKE}(C_a, S_h, (\hat{S}_a, P_a), (\hat{S}_h, P_h))[C_a] \]
2. m8 to m10 is the instantiation of AKT running between S_h and S_v:

\[ \text{AKT}(S_h, (\hat{S}_h, P_h)) \rightarrow \pi (S_v, (\hat{S}_v, P_v)) \]

---

**Fig 4.2 Initiate Format of Protocol Instantiation**

Later we combine m1 and m2, m4 and m8, m7 and m9 to simplify the communication to 9 rounds, which is the finial version in Fig 4.1.

### 4.3.2 Terminal Computational Ability

Computational ability is verified between different terminals; mobile, pocket PC, Palm are less powerful than PC. For communications on top of those terminals, we should balance complexity and security of security schemes.
5. System Design

5.1 System Structure

Three PC terminals are involved in our system; each represents a party in the protocol run, user, home server, and foreign server.

In a particular roaming scenario, user is out of the network coverage of home server, while both user and home server are in the network coverage of foreign server. To demo this network environment, we create a communication tunnel between user and foreign server, also between foreign server and home server. However, there is no direct communication tunnel between user and home server.

![System Architecture](image-url)
5.2 Process Description

In the following figures, process of the communication among user, foreign server and home server are described step by step.

Assumptions:

At the set-up of the system, the identities of user, home server and foreign server are known by each other except that foreign server has no idea of the user with anonymous feature. Same reason is concerned for the distribution of public key. User knows the public key of foreign server and home server, home server knows the public key of user and foreign server, but foreign server knows the public key of home server only.

Terms and symbols used in the figures:

In the figures, Known list in yellow box is used to keep the parameters obtained by each party. Parameter in black means its value was known in the previous processes. Parameter in red means its value is known in the current process. Parameter in blue means its value is used in the current process.

Orange boxes show the operations in each party.

Green boxes show the message sent from one party to the other.
Fig 5.2 Process Description, Step 1
Description: \( C_a \) randomly generates \( k_a, r_a \), compute \( H_1(k_a) \), adds \( k_a, r_a \) and \( H_1(k_a) \) to the known list, and then sends \( m_1 \) to \( S_v \).

Fig 5.3 Process Description, Step 2
Description: On receiving \( m_1 \), \( S_v \) decrypts the first part \( Enc_{Pv} (S_h// H_1(k_a)) \) with the private key \( \hat{S}_v \) to obtain \( S_h \) and \( H_1(k_a) \). It adds \( H_1(k_a) \) to the known list. If \( S_h \) is matched in its known list, \( S_v \) sends the second part \( m_2 = Enc_{Ph}(C_a // r_a // S_v // Sig_{Sa}(r_a // C_a // S_v // S_h)) \) to \( S_h \), otherwise, it halts the communication.
Fig 5.4 Process Description, Step 3
Description: $S_h$ decrypts $m_2$ using the private key $\hat{S}_h$, adds $r_a$ to the known list, and verifies the signature $\text{Sig}_{S_a}(r_a || C_a || S_v || S_h)$. If they don’t match, $S_h$ rejects the connection and halts. Otherwise, $S_h$ randomly generates $r_h$, computes $C = H_2(r_a || r_h)$, $\Pi = H_3(C_a || S_h || S_v || C)$, and then sends $m_3 = r_h || \text{MAC}_C(H_4(r_a || C_a || S_v || S_h)) || \text{Enc}_{Pv}(\Pi)$ to $S_v$.

Fig 5.5 Process Description, Step 4
Description: $S_v$ obtains $\text{Enc}_{Pv}(\Pi)$ from $m_3$, decrypts it with the private key $\hat{S}_v$ to get $\Pi$, and then forwards $m_4 = r_h || \text{MAC}_C(H_4(r_a || C_a || S_v || S_h))$ to $C_a$. 
Fig 5.6 Process Description, Step 5
Description: C_a obtains r_h from m_4, computes C = H_2 (r_a, r_h) and H_4(r_a// C_a || S_v // S_h), then performs MAC function of H_4 under the key C. The result should be the same as the second part of m_4. If they don’t match, C_a rejects the connection and halts. Otherwise, it sends m_5 = [Sig_s_a (H_5(r_h|| C||C_a || S_v || S_h)||Padding)|XOR G(C)] to S_v.

Fig 5.7 Process Description, Step 6
Description: S_v generates r_v, computes MAC_(r_v|| S_v || S_h), and then sends m_6 to S_h by appending r_v|| MAC_(r_v|| S_v || S_h)||Padding)|XOR G(C) to m_5.
Fig 5.8 Process Description, Step 7
Description: $S_h$ obtains $r_v$ from $m_6$, and then verifies the correctness of signature and MAC function. If either one is wrong, $S_h$ rejects the connection. Otherwise, it sends $m_7$ to $S_v$.

Fig 5.9 Process Description, Step 8
Description: $S_v$ verifies $m_7$. If it is false, $S_v$ rejects the connection and halts. Otherwise, $S_v$ randomly generates $k_v$, and sends $m_8 = H_f(k_a)XOR k_v \| H_f(k_a) || H_f(k_a) || k_v || r_v || H_f(k_a) || r_v || S_v || S_h)$ to $C_a$. 

- 44 -
Fig 5.10 Process Description, Step 9

Description: C_a computes $k_v = H_1(k_a) \oplus k_v \oplus H_1(k_a)$, and $\Pi = H_3(C_a || S_v || S_a || C)$, and then verifies $H_7(S_h || S_v || H_1(k_a) || k_v || \Pi)$. If the hash function is correct, it computes session key $\sigma = H_8(S_v || S_v || k_a || k_v || \Pi)$, and then sends $m_9 = H_1(k_v) \oplus k_a \oplus H_7(S_h || S_v || k_a || k_v || \Pi)$ to $S_v$. Otherwise, $C_a$ rejects the connection and halts.

Fig 5.11 Process Description, Step 10

Description: On receiving $m_9$, $S_v$ computes $k_a = H_1(k_v) \oplus k_a \oplus H_1(k_v)$, and verifies the hash function $H_7(S_h || S_v || k_a || k_v || \Pi)$. If the hash function is correct, it computes the session key $\sigma = H_8(S_v || S_v || k_a || k_v || \Pi)$, and considers the connection is accepted. Otherwise, $S_v$ halts with failure.
Description: For practical reasons, key confirmation processes should be achieved after both user and foreign server compute the session key $\sigma$. After that, user and foreign server destroy $k_v$ and $k_a$ for the security reason.
6. Implementation

6.1 Implementation Language

The application tool used in this project is Visual C++ 6.0 with Microsoft Foundation Class Library (MFC).

MFC is an "application framework" for programming in Microsoft Windows. Written in C++, MFC provides much of the code necessary for managing windows, menus, and dialog boxes; performing basic input/output; storing collections of data objects; and so on. Given the nature of C++ class programming, it's easy to extend or override the basic functionality the MFC framework supplies.

MFC provides a platform for us to design the interface of the system. Besides, it is compatible with the C++ library, which provides much convenience in the function implementation.
6.2 Session Communication

To run the AAKE-R protocol, communication channel should be built between user and foreign server, as well as between foreign server and home server, so that messages can be sent through the channel.

6.2.1 Socket \[^9\]

Socket is a communication endpoint, which an application can send message to or receive from. A socket is usually associated with an Internet address and port number, as well as a particular protocol (UDP or TCP).

![Fig 6.1 Socket](image-url)
6.2.2 MFC Windows Socket \[10\]

In MFC, a CAsyncSocket object represents a Windows Socket, which is an endpoint of network communication. The CAsyncSocket class encapsulates the Windows Sockets API, providing an object-oriented abstraction for programmers who want to use Windows Sockets in conjunction with MFC.

The CSocket class derives from CAsyncSocket and inherits its encapsulation of the Windows Sockets API. A CSocket object represents a higher level of abstraction of the Windows Sockets API than that of a CAsyncSocket object. CSocket works with classes CSocketFile and CArchive to manage the sending and receiving of data.

![Fig 6.2 MFC Socket Architecture](image-url)
Table 6.1 CAsyncSocket Class Members

<table>
<thead>
<tr>
<th>Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept</td>
<td>Accepts a connection on the socket.</td>
</tr>
<tr>
<td>AsyncSelect</td>
<td>Requests event notification for the socket.</td>
</tr>
<tr>
<td>Bind</td>
<td>Associates a local address with the socket.</td>
</tr>
<tr>
<td>Close</td>
<td>Closes the socket.</td>
</tr>
<tr>
<td>Connect</td>
<td>Establishes a connection to a peer socket.</td>
</tr>
<tr>
<td>IOCTL</td>
<td>Controls the mode of the socket.</td>
</tr>
<tr>
<td>Listen</td>
<td>Establishes a socket to listen for incoming connection requests.</td>
</tr>
<tr>
<td>Receive</td>
<td>Receives data from the socket.</td>
</tr>
<tr>
<td>ReceiveFrom</td>
<td>Receives a datagram and stores the source address.</td>
</tr>
<tr>
<td>Send</td>
<td>Sends data to a connected socket.</td>
</tr>
<tr>
<td>SendTo</td>
<td>Sends data to a specific destination.</td>
</tr>
<tr>
<td>ShutDown</td>
<td>Disables Send and/or Receive calls on the socket.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overridable Notification Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnAccept</td>
<td>Notifies a listening socket that it can accept pending connection requests by calling Accept.</td>
</tr>
<tr>
<td>OnClose</td>
<td>Notifies a socket that the socket connected to it has closed.</td>
</tr>
<tr>
<td>OnConnect</td>
<td>Notifies a connecting socket that the connection attempt is complete, whether successfully or in error.</td>
</tr>
<tr>
<td>OnOutOfBandData</td>
<td>Notifies a receiving socket that there is out-of-band data to be read on the socket, usually an urgent message.</td>
</tr>
<tr>
<td>OnReceive</td>
<td>Notifies a listening socket that there is data to be retrieved by calling Receive.</td>
</tr>
<tr>
<td>OnSend</td>
<td>Notifies a socket that it can send data by calling Send.</td>
</tr>
</tbody>
</table>
6.2.3 Session Communication

Since both user and home server should communicate with foreign server, we create server socket for foreign server, client sockets for user and home server. To manage unique operations on the event Accept, we create class CAsynsIntrn and class CAsynSvSk deriving from CAsynsocket as the server socket, and class CClntSock deriving from CSocket as the client socket. CAsynsIntrn is used for listening and accepting the connection request from client. CAsynSvSk is used for receiving from and sending message to client. CClntSock is used for creating connection with server, as well as receiving from and sending message to server.

Each communication session between server and client has three steps: Initiate Socket, Communication and Close Socket.

1) Initiate Socket
   a. Server creates a socket by assign a port, and then listens for the request from client.
   b. Client creates a socket, and then sends the connection request to server by specifying the server address and server port.
   c. Server accepts client’s connection request by assigning the specified port for communication.

2) Communication
   a. Client creates a message, and sends it to server
   b. On receiving the message, the server processes the data, creates a new message, and then sends it to client.
   c. On receiving the message, the client processes the data, creates a new message, and then sends it to server.
   d. Server and client repeat step b, c until the communication is completed.

3) Close Socket
   a. Server closes the sockets
   b. Client closes the sockets
**CAynslstn::Accept(CAsynSvSk)**

This member function accepts a connection on a socket. When CAsynslstn accepts a connection from client socket, it creates a new socket CAsynSvSk with the same properties as itself. CAsynSvSk cannot be used to accept more connections. The original socket CAsynslstn remains open and listening.

Since each socket with a specified port can only accept one communication at a time, so we create two pairs of server-client socket, one is between foreign server and user, the other is between foreign server and home server.
6.3 Cryptography Library

Microsoft Visual C++ itself never provides any powerful security libraries, however, there are many free C++ libraries of cryptography schemes, among which Crypto++ [11] developed by Weidai is a high-ranked one. The latest version is Crypto++ 5.2.1.

Table 6.2 Crypto++5.2.1 Reference Manual

<table>
<thead>
<tr>
<th>Abstract Base Classes</th>
<th>cryptlib.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric Ciphers</td>
<td>SymmetricCipherDocumentation</td>
</tr>
<tr>
<td>Hash Functions</td>
<td>HAVAL, MD2, MD4, MD5, PanamaHash, RIPEMD160, RIPEMD320, RIPEMD128, RIPEMD256, SHA, SHA256, SHA384, SHA512, Tiger, Whirlpool</td>
</tr>
<tr>
<td>Non-Cryptographic Checksums</td>
<td>CRC32, Adler32</td>
</tr>
<tr>
<td>Message Authentication Codes</td>
<td>MD5MAC, XMACC, HMAC, CBC_MAC, DMAC, PanamaMAC, TTMAC</td>
</tr>
<tr>
<td>Random Number Generators</td>
<td>NullRNG(), LC_RNG, RandomPool, BlockingRng, NonblockingRng, AutoSeededRandomPool, AutoSeededX917RNG</td>
</tr>
<tr>
<td>Password-based Cryptography</td>
<td>PasswordBasedKeyDerivationFunction</td>
</tr>
<tr>
<td>Public Key Cryptosystems</td>
<td>DLIES, ECIES, LUCES, RSAES, RabinES, LUC_IES</td>
</tr>
<tr>
<td>Public Key Signature Schemes</td>
<td>DSA, GDSA, ECDSA, NR, ECNR, LUCSS, RSASS, RabinSS, RWSS, ESIGN</td>
</tr>
<tr>
<td>Key Agreement</td>
<td>DH, DH2, MQV, ECDH, ECMQV, XTR_DH</td>
</tr>
<tr>
<td>Algebraic Structures</td>
<td>Integer, PolynomialMod2, PolynomialOver, RingOfPolynomialsOver, ModularArithmetic, MontgomeryRepresentation, GFP2_ONB, GF2NP, GF256, GF2_32, EC2N, ECP</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Compression</td>
<td>Deflator, Inflator, Gzip, Gunzip, ZlibCompressor, ZlibDecompressor</td>
</tr>
<tr>
<td>Input Source Classes</td>
<td>StringSource, FileSource, SocketSource, WindowsPipeSource, RandomNumberSource</td>
</tr>
<tr>
<td>Output Sink Classes</td>
<td>StringSinkTemplate, ArraySink, FileSink, SocketSink, WindowsPipeSink</td>
</tr>
<tr>
<td>Filter Wrappers</td>
<td>StreamTransformationFilter, HashFilter, HashVerificationFilter, SignerFilter, SignatureVerificationFilter</td>
</tr>
<tr>
<td>Binary to Text Encoders and Decoders</td>
<td>HexEncoder, HexDecoder, Base64Encoder, Base64Decoder, Base32Encoder, Base32Decoder</td>
</tr>
<tr>
<td>Wrappers for OS features</td>
<td>Timer, Socket, WindowsHandle, ThreadLocalStorage, ThreadUserTimer</td>
</tr>
<tr>
<td>FIPS 140 related</td>
<td>fips140.h</td>
</tr>
</tbody>
</table>

Since compiling of the latest versions of Crypto++ 5.0, 5.1, and 5.2.1 in Microsoft Visual C++6.0 needs the support of Processor Pack [12], we apply Crypto++ 4.2 in this project.

For the setting of system environment, please see the Appendix 1.
6.4 Parameter Setting

This part will cover the parameters we will use for our real system. Apart from some integers and words, all the values generated or computed are presented with Hexadecimal format.

6.4.1 Server Address and Port

In the socket initiation stage, server address and ports should be provided for clients to connect the communication. (See 6.2.3 Session Communication) Here, foreign server is the server, and user and home server are the clients. The foreign server should assign two distinguished ports for user and home server.

6.4.2 Known List

There are two arrays with respect of Known List:

1) arrayKnownID: store the ID of Known items
2) arrayKnownValue: store the values of Known items

The size of arrayKnownID and arrayKnownValue is set to be 20. It is because the number of known items is limited to 20.

6.4.3 ID of Ca, Sh and Sv

The identifiers of Ca, Sh and Sv are the random strings meaningless to the unauthorised parties. The length of IDs, which are represented as Hexadecimal, is 8.

6.4.4 Public and Private Key of Ca, Sh, and Sv

Generation of Public key and private key requires the participation of key seed. The keys are stored in the .key files. File path is stored in the Known List for later access.
void CTralIncDlg::GenKeys()
{
    //Generate RSA public Key and private Key of Sv
    const char *privFilenameSv = KeyPriSv;
    const char *pubFilenameSv = KeyPubSv;
    const char *seedSv = KeySeedSv;

    RandomPool randPoolKeySv;
    randPoolKeySv.Put((byte *)seedSv, strlen(seedSv));

    RSAES_OAEP_SHA_Decryptor privSv(randPoolKeySv, keyLength);
    HexEncoder privFileSv(new FileSink(privFilenameSv));
    privSv.DEREncode(privFileSv);
    privFileSv.MessageEnd();

    RSAES_OAEP_SHA_Encryptor pubSv(privSv);
    HexEncoder pubFileSv(new FileSink(pubFilenameSv));
    pubSv.DEREncode(pubFileSv);
    pubFileSv.MessageEnd();

    aryKnownValue.SetAt(aryIndex, KeyPriSv);
    aryKnownID.SetAt(aryIndex, KeyPriSvID);
    KeyPriSvIndex = aryIndex;
    aryIndex++;

    aryKnownValue.SetAt(aryIndex, KeyPubSv);
    aryKnownID.SetAt(aryIndex, KeyPubSvID);
    KeyPubSvIndex = aryIndex;
    aryIndex++;

    aryKnownValue.SetAt(aryIndex, KeySeedSv);
    aryKnownID.SetAt(aryIndex, KeySeedSvID);
    KeySeedSvIndex = aryIndex;
    aryIndex++;

    ......
6.5 Implementation Result

6.5.1 Ca (User)

The user will start its execution as follows.

![Fig 6.6 User](image)

On execution, the user (Ca) knows the identities of its home server (Sh) and current foreign server (Sv). Besides of its own private key and public key of RSA scheme, Ca also knows the public key of Sh and Sv.

Purpose of running this protocol for Ca:

1) Create communication session with foreign server
2) Pretend release of personal identification
6.5.2 Sv (Foreign Server)

The foreign server will start its execution as follows:

On execution, the foreign server (Sv) have no idea of the service request user, but knows its cooperated server (Sh), that is, the home server of service request user. Besides of its own private key and public key of RSA scheme, Sv also knows the public key of Sh.

Purpose of running this protocol for Sv:

1) Build up communication session with service request user
2) Verify the identification of service request user through his claimed server
3) Have no authority to know the identification of service request user
6.5.3 Sh (Home Server)

The home server will start the execution as follows:

![Fig 6.8 Home Server](image)

On execution, the home server (Sh) knows the identities of its subscribed user (Ca) and cooperated server (Sv), that is, the current foreign server of Ca. Besides of its own private key and public key of RSA scheme, Sh also knows the public key of Ca and Sv.

Purpose of running this protocol for Ca:

1) Verify the identification of subscribed user for cooperated server on demand
2) Pretend release of identification of subscribed user
6.5.4 Communication Procedure

6.5.4.1 Edit Address and Port

On the interface of Sv, Ca and Sh, click button “Edit Detail”, “Data Update Dialogue” will display. This dialogue shows the data values known to each terminal at the initiation stage of the execution.

<table>
<thead>
<tr>
<th>Terminal</th>
<th>Known List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ca</td>
<td>Data of Ca: ID, RSA Public Key, RSA Private Key</td>
</tr>
<tr>
<td></td>
<td>Data of Sh: ID, RSA Public Key</td>
</tr>
<tr>
<td></td>
<td>Data of Sv: ID, RSA Public Key, Address, Port for User</td>
</tr>
<tr>
<td>Sh</td>
<td>Data of Sh: ID, RSA Public Key, RSA Private Key</td>
</tr>
<tr>
<td></td>
<td>Data of Ca: ID, RSA Public Key</td>
</tr>
<tr>
<td></td>
<td>Data of Sv: ID, RSA Public Key, Address, Port for Server</td>
</tr>
<tr>
<td>Sv</td>
<td>Data of Sv: ID, RSA Public Key, RSA Private Key, Port for User, Port for Server</td>
</tr>
<tr>
<td></td>
<td>Data of Sh: ID, RSA Public Key</td>
</tr>
</tbody>
</table>

All the data with same name among three terminals should have the same data value. The address and ports are editable, so that the application can be run on any machine.

In the run time, address of Sv must be the IP address of the machine that Sv is running on. Port for user that is set in both Ca and Sv should keep the same, as well as port for server that is set in both Sh and Sv. For the interface, please see Fig 6.9 Edit Address and Port.
Fig 6.9 Edit Address and Port
6.5.4.2 Initiate Socket

Step 1: Click the button “Initiate Socket” of Sv, Sv creates two communication sockets with different ports, and listens for the request.

![Initiate Socket, Step 1, Sv](Fig 6.10 Initiate Socket, Step 1, Sv)
Step 2: Click the button “Initiate Socket” of Sh, Sh builds the connection with Sv with the specified port.
Step 3: Click the button “Initiate Socket” of Ca, Ca builds the connection with Sv with the specified port.
6.5.4.3 Rundown Demo

All Ca, Sv and Sh select “Rundown Demo”.

Step 1: Click “GenM1” in Ca, the main process of computing message 1 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M1Log.txt”.

![Fig 6.15 Rundown Demo, Step 1](image-url)
Step 2: Follow the message flow, click “GenM2” in Sv, the main process of computing message 2 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M2Log.txt”.

Fig 6.16 Rundown Demo, Step 2

Step 3: Follow the message flow, click “GenM3” in Sh, the main process of computing message 3 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M3Log.txt”.

Fig 6.17 Rundown Demo, Step 3
Step 4: Follow the message flow, click “GenM4” in Sv, the main process of computing message 4 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M4Log.txt”.

![Fig 6.18 Rundown Demo, Step 4](image1)

Step 5: Follow the message flow, click “GenM5” in Ca, the main process of computing message 5 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M5Log.txt”.

![Fig 6.19 Rundown Demo, Step 5](image2)
Step 6: Follow the message flow, click “GenM6” in Sv, the main process of computing message 6 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M6Log.txt”.

Step 7: Follow the message flow, click “GenM7” in Sh, the main process of computing message 7 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M7Log.txt”. Sh has completed its duty in protocol, “Known List” shows the data values he owns after the protocol run.
Step 8: Follow the message flow, click “GenM8” in Sv, the main process of computing message 8 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M8Log.txt”.

Fig 6.22 Rundown Demo, Step 8

Step 9: Follow the message flow, click “GenM9” in Ca, the main process of computing message 9 will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M9Log.txt”.

Fig 6.23 Rundown Demo, Step 9
Step 10: Follow the message flow, click “Compute SessionKey” in Sv, the main process of computing session key will be displayed in “Status” Box. Detailed data will be saved in the file “\Log\M10Log.txt”. Sh has completed its duty in protocol, “Known List” shows the data values it owns after the protocol run. Sh then sends the “Success” statement to Ca.

Fig 6.24 Rundown Demo, Step 10

Step 11: Click “Final Status” in Ca, Ca shows the communication statement from Sv. The communication session is completed. “Known List” shows the data values Ca owns after the protocol run.

Fig 6.25 Rundown Demo, Step 11
6.5.4.4 Test Case

Instead of “Rundown Demo”, all Ca, Sv and Sh select “Test Case”.

Click button “Test” in Ca, all the procedures in the session 6.5.4.3 will run automatically.

![Fig 6.26 Test Case, Ca](image-url)
Traveling Incognito

Fig 6.27 Test Case, Sv

- Purpose
  1. Build up Communication Session with Service Request User
  2. Verify the Identity of Service Request User through his Claimed Server
  3. No Authority to Know the Identity of Service Request User

- Status
  - Create Communication Socket, Port 2565, Listening...
  - Build the connection with Sv, Port 2500
  - Build the connection with Sh, Port 2600

- Receive Message 1 from Ca: EncPv (H1[A] || EncPv(Ca || Sv || Sv || Sh))
  - Decrypt EncPv (H1[A])
  - Match Sh

- Receive Message 2 from Sh: EncPv(Ca || Sv || Sv || Sh)
  - Decrypt EncPv (H1[A])
  - Send Message 4 to Ca: EncPv (H4[Ca] || Sv || Sv || Sh)

- Receive Message 3 from Sh: EncPv (H4[Ca] || Sv || Sv || Sh)
  - Decrypt EncPv (H4[A])
  - Send Message 5 to Ca: EncPv (H5[Ca] || Sv || Sv || Sh)

- Receive Message 4 from Ca: EncPv (H5[A] || Sv || Sv || Sh)
  - Decrypt EncPv (H5[A])
  - Compute MAC PI (H5[A] || Sv || Sv || Sh)
  - Send Message 7 to Sh: SigSh[H5[A] || Sv || Sv || Sh]

- Receive Message 5 from Sh: SigSh[H5[A] || Sv || Sv || Sh]
  - Decrypt SigSh[H5[A] || Sv || Sv || Sh]

- Generate Sv

Fig 6.26 Test Case, Sh

- Purpose
  1. Verify the Identity of Subscribed User for Cooperated Server on Demand
  2. Pretend Release the Identity of Subscribed User

- Status
  - Build connection with Sv, Port 2500
  - Build connection with Sh, Port 2600

- Receive Message 2 from Sh: EncPv (H2[A] || EncPv(Ca || Sv || Sv || Sh))
  - Decrypt Message 2
  - Verify SigSh[A] (Ca || Sv || Sv || Sh)
  - Generate Sv
  - Compute C = H2[A] || Sv || Sv || Sh
  - Compute MAC H2[A] (H2[A] || Ca || Sv || Sv || Sh)

- Receive Message 4 from Sv: SigSh[H2[A] || Ca || Sv || Sv || Sh]
  - Verify SigSh[H2[A] || Ca || Sv || Sv || Sh]
  - Compute Message 7: SigSh[H5[A] || Sv || Sv || Sh]

- Send Message 7 to Sv

- KNOWN LIST
  - Ca
  - Sv
  - Sh
  - KeyPubCa
  - KeyPubSh
- Length: 14
  - AEBC3D1
  - 449854B0
  - 529BED5D
  - KeyPubCa
  - KeyPubSh
  - This is the Seed of RSA Key of Ca

- Process
  - Edit Detail
  - Initiate Socket
6.5.4.5 Follow-up steps

Save Status: save the data in status box into a text file.

Close Socket: close this communication session.

Refresh: clear the status box; initiate the data to the original value. And then can click “Initiate Socket” to start another communication session.

Close Window: close the application window.

Fig 6.27 Follow-Up Steps, Ca
6.5.5 Result Checking

After the communication run, this project has following achievements:

1) In the Sv’s known list, we cannot find any information of Ca. User anonymity is achieved.

2) Session key is shared between Ca and Sv. Sh has no idea of it. Moreover, if we run the communication sessions several times, the session keys generated are different.

3) If we put an illegal user who is not subscribed to Sh into the communication, Sh cannot verify the identity of user and will inform Sv to terminate the communication.
7. Conclusion

7.1 Critical Review

My first touch with cryptography was from the course “Internet Security and E-Commerce Protocols”, which was instructed by Prof. Deng in semester B of Year 2003/2004. The material of Chapter 2 in this paper comes from the textbook “Cryptography and Network Security”.

Later, Prof. Deng recommended me to follow Dr. Wong’s project, AAKE-R. To understand the essence of the protocol, I put lots of effort in the extensive readings. The most helpful knowledge came from the weekly discussion with Dr. Wong and Tommy. I witnessed and participated in the polishing process of the protocol, from AAKE-R (v1.0) to AAKE-R (2.7). At the same time, I studied several AKE schemes, and focused on my approach to instantiate the protocol, so that it can be implemented in a reality.

The process of instantiation is not straightforward. Many issues should be taken into account due to security and practical reasons. Based on Boyd and Park’s AKE scheme, I made several modifications (introduced in section 4.2) and tried to reduce the communication complexity.

After that, I started to design the system based on the finalized protocol. System implementation is a world different with paper work on theory. Many components, physical or non-physical, should be well considered. Finding out a workable cryptography library was the first step. At the very beginning, I wanted to apply Crypto++ 5.2.1 in our system, but it cannot run on Microsoft Visual C++ 6.0 (see the reason in 6.3). Dr. Wong suggested me to use another library cryptolib 1.1, however, it was implemented in VC and was not compilable in VC++. Later, I found out other separated cryptosystem components like RSA, MD5, but linking between these components was a challenge. Finally, I adopted an early version of Weidai’s Crypto++ series, which is Crypto++ 4.2 in our project.
Crypto++ 4.2 provides powerful cryptography classes. However, to familiarize with and select useful classes from a bundle of them is a job in detail. On top of that, I should build my own functions with standard interface, such as Encryption, Decryption, Signature, Verification, Hash, MAC, XOR etc. I also need to make the data represented in a uniform way, from bit, byte, word, to hexadecimal.

Besides, implementation of sockets was another challenge. In order to make the protocol rundown automatically and smoothly, how to trigger the ‘send’ and ‘listen’ events on each message should be well designed.

Most critical issue is to keep so many messages and parameters integrity and find out the relationships between them. It is essential for the practical proof of the protocol, and also is the main object of this project.

This project not only gave me a deep insight in the cryptography world, but also trained my thinking in a systematic and logical way.
7.2 Suggestion for Future Improvement

Here is a short list of suggestions for extensions to the project:

1. Further reduce the communication rounds
   Based on AAKE-R (v2.7), the protocol instance applied in this project has 9 communications rounds. That is still too complex. Most of the current security models applied in wireless network only have about 3~5 communication rounds. To achieve that, the latest version AAKE-R (v3.5)[13] can be taken into account, which has 4 rounds only.

2. Achieve multiple users and servers
   This project demo a very ideal case: one user, one home server, and one foreign server. Multiple users and servers can be involved in the later extension to imitate the real communication environment. To achieve that, socket handling and message head management is the key.

3. Make the system a mobile application
   To do that, computation power of mobile devices should be well cared.
References


[11] Download Crypto++:
    http://www.eskimo.com/~weidai/cryptlib.html

[12] Visual C++ 6.0 Processor Pack:

Appendix

1. Setting System Environment

1. Download the Crypto++ v4.2 Library:
   \[\text{http://www.eskimo.com/~weidai/cryptlib.html}\]
2. Compile the Crypto++ Library (both Debug/Release).
3. Take the Debug version of the library and rename it to \textit{cryptlib42d.lib}.
4. Take the Release version of the library and rename it to \textit{cryptlib42r.lib}.
5. Copy the \*.lib files to both of the \textit{LIB} directories where VC++ is installed.
6. Unzip the demo and put it into a directory of your choice.
7. Go to the Menu -> Tools | Options | Directory Tab, and enter the directory path where you unzipped the Crypto++ Library to.
8. Go to the Menu -> Project | Settings | Link Tab, and enter the build library (\textit{cryptlib42d.lib}/\textit{cryptlib42r.lib}) in the "Object/Library modules" edit box.
9. Make sure the above library matches the build (*.lib=Release/ *.lib=Debug) for the Cryptest demo.
10. Compile the Cryptest demo (do not run it from here).
11. For the validation portion to work, you need to copy the .dat files from the Crypto++ location to the build directory (\textit{Debug}/\textit{Release}).
12. Execute \textit{TralncCa.exe}, \textit{TralncSv.exe}, and \textit{TralncSh.exe}.

2. Interim Report