2004

A new protocol with unbalanced RSA for authentication and key distribution in WLAN.

Zhong. Zheng
University of Windsor

A New Protocol with Unbalanced RSA for Authentication and Key Distribution in WLAN

by

Zhong Zheng

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the Department of Electrical and Computer Engineering in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science at the University of Windsor

Windsor, Ontario, Canada
2004

Your file Votre référence
ISBN: 0-494-00143-7
Our file Notre référence
ISBN: 0-494-00143-7

10187799

A New Protocol with Unbalanced RSA for Authentication and Key Distribution in WLAN

by

Zhong Zheng

APPROVED BY:

A. Ngom (External Reader)
Department of Computer Science

S. Erfani (Internal Reader)
Department of Electrical and Computer Engineering

K. Tepe (Co-supervisor)
Department of Electrical and Computer Engineering

H. Wu (Co-supervisor)
Department of Electrical and Computer Engineering

University of Windsor

September 21, 2004

Abstract

In wireless network, security concerns have haunted 802.11 deployments since the standardization effort began. IEEE attempts to provide confidentiality by using WEP (Wire Equivalent Privacy), and treats WEP as an option during the authentication. Unfortunately, WEP had been proved that neither authentication nor data confidentiality is reliable. For the short-term solution, IEEE offers TKIP (Temporal Key Integrity Protocol) to address the flaws found in 802.11, combined with 802.1X for authentication. In order to provide solid mutual authentication and key-distribution, TLS (Transport Layer Security) handshake protocol has been used in 802.1X. However, since TLS was not designed specifically for 802.11 in WLAN, there are some redundant steps in TLS which is not necessary if used for 802.11. Furthermore, in WLAN, it is normal that the computation abilities between client and server could be significantly different, which make the client a bottleneck during the handshake process. According to those drawbacks, a new protocol for authentication and key-distribution is proposed in this thesis. This new protocol can not only eliminate the redundant steps in TLS handshake, but also reduce the time consumption for client during the authentication and key-distribution by applying "unbalanced RSA". The proposed protocol with the use of "unbalanced RSA" solves the problems in original

802.11 standard, while offering efficiency and security at the same time.

To my well-beloved parents and grandparents.

Acknowledgement

I would like to thank Dr. Kemal E. Tepe and Dr. Huapeng Wu for the invaluable direction and feedback they have given me on this thesis. They have provided the perfect balance of direction and freedom, allowing me to pursue my own ideas and supporting me the whole while. I would also like to thank Wenkai Tang and all the other people who have ever helped me during my research.

Contents

List of Figures

List of Tables

Abbreviations

AES	Advanced Encryption Standard
CA	Certificate Authority
CRC	Cyclic Redundancy Code
EAP	Extensible Authentication Protocol
ICV	Integrity Check Value
IEEE	Institute of Electrical Electronics Engineers
IV	Initialization Vector
LAN	Local Area Network
MD5	Message Digest 5
MIPS	Million Instructions Per Second
OTP	One-time Password
PDA	Personal Digital Assistant
PRNG	Pseudo-Random Number Generator
RADIUS	Remote Authentication Dial-In User Server/Service
RC4	Ron's Code 4
RFC	Request For Comment

RSA	Rivest, Shamir, Adleman (public key encryption technology)
SSL	Secure Sockets Layer
TGi	Task Group i
TKIP	Temporal Key Integrity Protocol
TLS	Transport Layer Security
WEP	Wired Equivalent Protocol
WLAN	Wireless Local Area Network

Chapter 1

Introduction

Over the past several years, the world has become increasingly mobile. As a result, traditional ways of networking have proven inadequate to meet the challenges posed by our modern lifestyle. If users must be connected to a network by physical cables, their movement is dramatically reduced. Wireless connectivity, however, poses no such restriction and allows a much more free movement on the part of the network user.

Unlike wired network, wireless network uses radios instead of wires as the medium for data transmission. For wired LANs, attackers must obtain physical access to the network medium before attempting to eavesdrop on traffic. But for wireless network, physical access to wireless networks is a comparatively simpler matter of using the correct antenna and modulation methods [1].

The most successful wireless networking technology so far is 802.11, which is "a suite of protocols defining a wireless local area network (WLAN)", specified by IEEE Standard 802.11-1999 [2]. IEEE 802.11 offers a wired LAN equivalent data confidentiality algorithm, which is called Wired Equivalent Privacy (WEP). WEP is designed as protecting authorized users of a wireless LAN from casual eavesdropping. This service is intended to provide functionality for wireless LAN equivalent to that provided by the physical security attributes inherent to a wired medium [1]. But unfortunately, WEP had been proved unsecure shortly after it appeared [3][4].

The main reason which caused WEP flawed is the improper implementation of RC4 algorithm in WEP [3]. Because the authentication in 802.11 treats WEP as an option for confidentiality, it can not be regarded as reliable neither [5]. Another reason is the use of an unkeyed checksum algorithm, which allows the message be modified without detection [3].

The best solution to address WEP's flaws is to keep the secret key fresh, or to design a protocol for key distribution. A new method for authentication is also necessary. Using a keyed checksum algorithm instead of the unkeyed one can address the message authentication flaws [6].

The IEEE 802.11 Working Group recognized the gravity of the security issues in 802.11 and created Task Group "i" (TGi) to resolve them. TGi has designed two solutions for the problems. One is called Temporal Key Integrity Protocol (TKIP) [6], intended as a short-term patch for currently deployed equipment. The other one will use AES (Advanced Encryption Standard) [7], a totally different cryptography algorithm, as a long-term solution.

TKIP offers some new elements, such as a keyed Message Integrity Code (MIC) to defeat forgeries; a key mixing function to defeat FMS (FMS are the initials of the three authors) [4] attacks; a new IV sequencing discipline to remove replay attacks. More importantly, TKIP has adapted IEEE 802.1X [8] to provide both authentication and key distribution for WLAN, in order to address those flaws in original 802.11. Among many options supplied by 802.1X, using TLS handshake protocol is the most secure choice, since it can achieve mutual authentication, and keys can also be distributed during the authentication.

TLS is the third version of SSL, which is originally used as a protocol in transport layer [9]. It assumes that both communication parties have their own SSL version, key-exchange method, and other specific parameters. Therefore, during the TLS handshake, both parties have to negotiate all the parameters first, then do the authentication and key distribution. TLS protocol is good because it is considered compatible to most of the systems which support different SSL versions and different algorithms. However, if using a certain parameter or key-exchange method can enhance the performance for authentication and key distribution in a certain case, then we can force the TLS handshake protocol to use that method, omitting the negotiation part, simplifying the whole process.

In WLAN, the device for communication on client side could be a laptop, a PDA, or even a cellphone. Comparing to the server side, the computation ability on client side is restricted. For example, to finish an RSA encryption with 1024-bit modulus, using a Pentium4 2.1 GHz processor can achieve about 3456 times faster than using a 20 MHz microprocessor [10][11]. Therefore, during the authentication and key distribution in TLS handshake protocol, which involves time-consuming public key algorithms, the computation ability on client side is definitely a bottleneck. No matter how fast the server's processor can achieve, the time consumption only depends on the client, which has a slower processor. If we can find out a way to accelerate the calculation on client side, then the whole process's performance will be enhanced. Fortunately, Adi Shamir, one of the three authors who created RSA, brought out an idea, called "unbalanced RSA" [12], which is especially for solving the problem caused by the unbalanced computation ability of the two communication parties. The basic idea in "unbalanced RSA" is to use different size of p and q for different parties, while remaining the size of modulus n stable. This can significantly reduce the numbers of calculation operations on one side, so that the total time for RSA calculation will shrink. Thus, we should use "unbalanced RSA" during the authentication and key distribution in WLAN, and because this is the only method we will use in TLS handshake protocol, there is no need to negotiate the key exchange method or any other uncertain parameters. Based on unbalanced RSA, we can both reduce the time for RSA decryption and simplify the TLS protocol, then decrease the total time consumption for authentication and key distribution.

In this thesis, a new protocol is designed based on the application of unbalanced RSA. Java is used to simulate this new protocol with the unbalanced RSA on the application layer. Next section is a review of the flaws found in original 802.11. Section 3 discusses the current solutions for those flaws. Section 4 introduces the proposed protocol for authentication and key distribution in WLAN. The application of unbalanced RSA is described in this section, too. Section 5 presents the simulation result, as well as some discussion based on this result. Section 6 is the conclusion of this research and the future work.

Chapter 2

A Review on IEEE 802.11 Privacy

In this chapter, WEP and the WEP-based authentication will be explained in detail. The WEP's flaws will also be presented. By the end of this chapter, we will gain a comprehensive background on how unsecure the 802.11 is, and this can help us to understand the importance of addressing 802.11's problems.

2.1 Wired Equivalent Privacy (WEP)

The IEEE 802.11 standard [2] defines a data confidentiality mechanism known as WEP. WEP works using RC4 encryption with a shared key. The security goal of WEP is data confidentiality equivalent to that of a wired LAN.

Figure 2.1 shows how WEP encrypts a plaintext into a ciphertext. First, we choose an initialization vector (IV). We then concatenate the shared secret key with IV. The result is treated as a seed and put into a WEP PRNG (Pseudo Random Number Generator), which is a part of the RC4 algorithm. The outcome is a key sequence, or

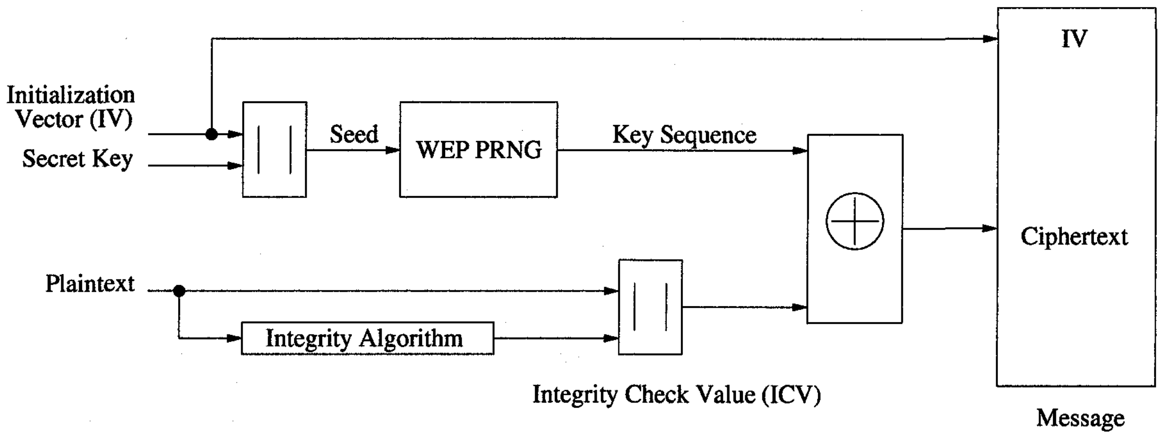

Figure 2.1: WEP Encryption Block Diagram

keystream. At the same time, an integrity check algorithm is applied to the plaintext in order to get the ICV (Integrity Check Value). We XOR the concatenation of plaintext and ICV with the keystream, and the result is the ciphertext. In the end, we concatenate the plain IV with the ciphertext, and send them as a message to the receiver.

Symbolically, the encryption process may be represented as follows:
Sender → Receiver:

$$IV, (M||CRC(M)) \oplus RC4(K, IV)$$

Where C is Ciphertext, K is Secret Key, M is Plaintext. Or, we can refer to the following Figure 2.2:

To decrypt a frame protected by WEP, the recipient simply reverses the encryption process. First, he regenerates the keystream $RC4(K, IV)$, then he can get the plaintext with the ICV by

$$\begin{aligned} & C \oplus RC4(K, IV) \\ = & (M||CRC(M)) \oplus RC4(K, IV) \oplus RC4(K, IV) \\ = & (M||CRC(M)) \end{aligned}$$

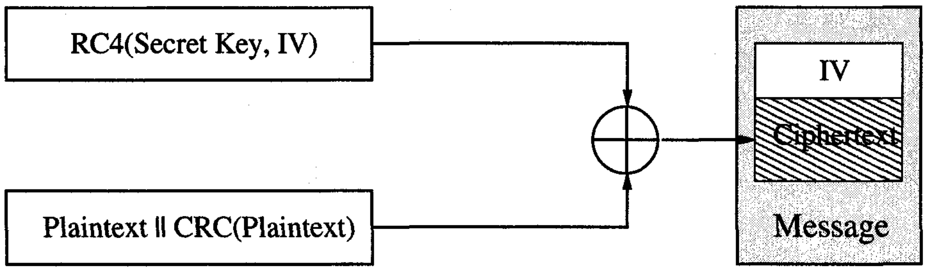

Figure 2.2: WEP encryption symbolical figure

Next, the recipient verifies the ICV by doing $CRC(M)$, if the result matches the one he got from the message, then he assume the message has not been modified during the transmission. This ensures that the only frames with a valid ICV will be accepted by the receiver.

2.2 The Flaws in WEP

Many of the headlines about 802.11 over the past years were due to WEP. As networks become important to business, security has become an increasingly prominent worry. WEP was initially marketed as the security solution for wireless LANs, but as a matter of fact, its design had been found flawed as to make that impossible. There are some very good references which have given comprehensive explanations about the flaws in WEP, such as [3] and [4]. Since the 802.11 authentication is based on WEP, reference [5] has taken a deep look at the problems in authentication, which will be explained later in this chapter.

2.2.1 The Risks of Keystream Reuse

WEP provides data confidentiality by using a stream cipher called RC4. Normally, stream ciphers operate by expanding a secret key (or, as in the case of WEP, a public IV and a secret key) into an arbitrarily long "keystream" of pseudo random bits (it is done by using a PRNG in WEP). Encryption is performed by XORing the keystream with the plaintext. On the other hand, the decryption consists of generating the identical keystream based on the IV and secret key and XORing it with the ciphertext.

There is a well-known problem about stream ciphers that encrypting two messages under the same keystream can reveal the original messages. In the case of WEP, if

$$C_1 = P_1 \oplus RC4(IV, K)$$

and

$$C_2 = P_2 \oplus RC4(IV, K),$$

then

$$\begin{aligned} & C_1 \oplus C_2 \\ = \ & (P_1 \oplus RC4(IV, K)) \oplus (P_2 \oplus RC4(IV, K)) \\ = \ & P_1 \oplus P_2, \end{aligned}$$

where P is $(M||CRC(M))$.

In other words, XORing two ciphertexts (C_1 and C_2) can get the result of $(P_1 \oplus P_2)$. There are several ways to get the plaintext from $(P_1 \oplus P_2)$. For instance, if one of the plaintext P_1 is known, then the other one will be revealed instantly by

$$P_2 = P_1 \oplus (P_1 \oplus P_2).$$

Even without knowing any of these two plaintexts, there are still many techniques to reveal the messages [13][14].

In order to avoid the "keystream reuse", WEP uses a per-packet IV to vary the keystream for each frame of data transmitted. As mentioned before, the keystream in WEP is $RC4(IV, K)$. Although the secret key K changes rarely, the IV changes for every frame. Thus, for every different frame, the keystream is different, so there is no way to get $(P_1 \oplus P_2)$ from $(C_1$ and $C_2)$ based on the same keystream. But unfortunately, WEP does not achieve this goal.

From Figure 2.1 we know that IV is transmitted in plain, so duplicate IVs can be easily detected by anyone including attackers. Based on the fact that the shared secret key rarely changes, any reuse of old IV will expose the system to keystream reuse attacks.

Even worse, 802.11 standard does not even require the IV to be changed after every packet. In other words, there is no IV management at all, which let some implementations do it poorly. For example, a particular PCMCIA card resets the IV to 0 each time it is re-initialized, and then increase the IV by one for each packet transmitted. The card is re-initialized each time it is inserted into a laptop, which can happens frequently. Consequently, keystreams corresponding to low-valued IVs are likely to be reused many times during the lifetime of the key.

However, even if a perfect method could be found to manage the IVs very well, the keystream reuse still can not be avoided. This is because the IV field used by WEP is only 24 bits wide, which nearly guarantees that the same IV will be reused for multiple messages. For example, assuming there is a busy access point which can send 1500 byte packets and achieve an average of 5Mbps bandwidth. Then, the time used for exhausting a 24-bit IV's space is:

$$\frac{\frac{5Mbps}{8(bit/byte)}}{1500byte} = 416.625(packets/sec)$$

$$\frac{2^{24}}{416.625} = 40269.34(sec) = 11.19(hour)$$

An old keystream will be used again in less than half a day according to the calculation above.

Therefore, to extend the life time of IV, or let's say to extend the length of IV, is the way to reduce the risk of keystream reuse, and this is what some so-called "enhanced WEP" did. However, no matter how long the life time of IV is, the risk of keystream reuse may be reduced, but other flaws still exist, such as the flaw caused by FMS attack [4]. There are two requirements for FMS attack to reveal the keystream:

- Enough ciphertexts which used the same secret key with numerous different initialization vectors.
- Obtain the first word of RC4 output corresponding to each IV.

Since WEP changes its secret key very rarely (in fact, some implementations never change it at all), the first requirement is easy to achieve. The first word of plaintext is often an easily guessed constant which could be the date, the sender's identity, etc. With the corresponding ciphertext, attackers can get the first word of RC4 output, or the first word of a keystream without lots of work [3]. The detail of FMS attack is out of the scope of this thesis, reference [4] has more information about this attack.

The best countermeasure to address the flaws caused by keystream reuse and FMS attack is to use fresh shared secret key, which is never mentioned in the 802.11 standard. In other words, the key-distribution is very important for WLAN in terms of security. Changing the secret key frequently can avoid the reuse of IV, and the first requirement of FMS attack will never be achieved.

The key-distribution is a very important issue for addressing 802.11 security flaws. This has been considered in TKIP which will be discussed in the next chapter. But what has been done in TKIP is still not the best solution in the case of WLAN, and the reason of this is in chapter 4 and 5.

2.2.2 Message Authentication

The WEP protocol uses an integrity checksum field to ensure that packets do not get modified during the transmission. The algorithm used is CRC-32 (Cyclic Redundancy Code 32 bits), and the ICV is part of the encrypted payload of the packet. However, CRC is not a cryptographically secure authentication code, and it is designed to detect random errors in the message, instead of malicious attacks. Actually, CRC is exacerbated by the fact that the message payload is encrypted using a stream cipher.

Readers can find the details of this flaw in [3]. The solution of this problem is to use an encrypted integrity check algorithm to take place of CRC-32 used in 802.11. This has been done by TKIP using a keyed MIC (Message Integrity Check) called Michael, which will be explained in the next chapter.

2.3 802.11 Authentication and Its Flaws

On a wired network, authentication is implicitly provided by physical access. While this is a weak definition of authentication, and one that is clearly inappropriate for high-security environments, it works reasonably well as long as the physical access control procedures are strong. Wireless networks are attractive in large part because physical access is not required to use network resources. Therefore, a major component of maintaining network security is ensuring that stations attempting to associate with the network are allowed to do so. Two major approaches are specified by 802.11: open-system authentication and shared-key authentication. Shared-key authentication is based on WEP and requires that both stations implement WEP.

2.3.1 Open System Authentication

Open System authentication is the simplest of the available authentication algorithms, and is the only method required by 802.11. Essentially, it is a null authentication algorithm. Any station that requests authentication with this algorithm may become authenticated if 802.11 Authentication Type at the recipient station is set to Open System Authentication. No doubt, this method can provide no security service at all.

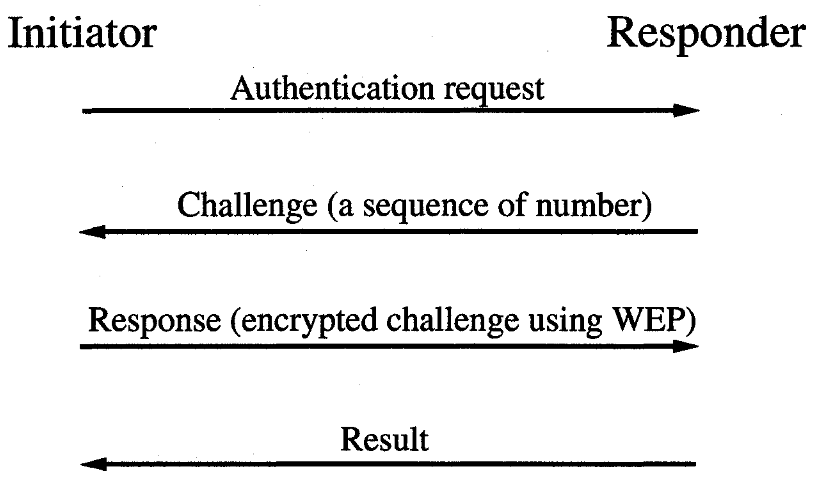
Initiator
Responder
Authentication request
Challenge (a sequence of number)
Response (encrypted challenge using WEP)
Result

Figure 2.3: Shared-Key Authentication

2.3.2 Shared-Key Authentication

Shared-Key authentication uses a challenge and a response along with a shared secret key to provide authentication.

The entire shared-key authentication is shown as Figure 2.3. The initiator sends an authentication request management frame indicating that he wishes to use "shared-key" authentication. The responder responds by sending an authentication management frame containing 128 octets of challenge text to the initiator. Once the initiator receives the management frame, he copies the contents of the challenge text into a new management frame body. This new management frame body is then encrypted with WEP using the "shared secret" along with a new IV selected by the initiator. The encrypted management frame is then sent to the responder. The responder decrypts the received frame and verifies that the ICV is valid, and that the challenge text matches the one sent in the first message. If the result is positive, the authentication is successful.

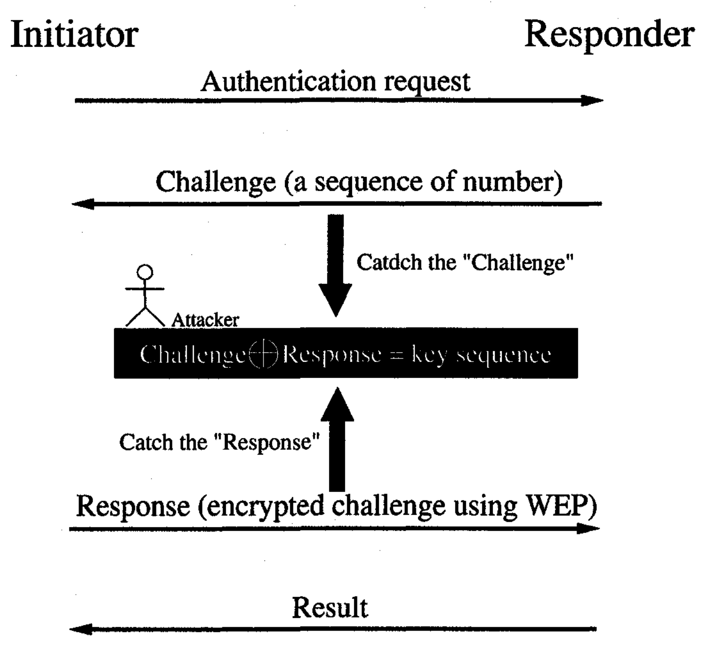
Initiator
Responder
Authentication request
Challenge (a sequence of number)
Catdch the "Challenge"
Attacker
Challenge ⊕ Response = key sequence
Catch the "Response"
Response (encrypted challenge using WEP)
Result

Figure 2.4: Shared-Key authentication is attacked by a third person

2.3.3 Flaws in Authentication

The current protocol for shared-key authentication is easily exploded through a passive attack by eavesdropping the authentication. The attack works because of the previously reported weaknesses in WEP.

As shown in Figure 2.4, while two stations are doing shared-key authentication, an attacker can capture both the second and the third messages, which are the random challenge (M) in clear and the encrypted challenge (C) by WEP, respectively. Noting the WEP encryption equation is $C = M \oplus Keystream$, we XOR M and C, the result should be the keystream ($Keystream = M \oplus C$).

Without the shared-key but only the keystream, the attacker can request authentication of any AP (access point) it wishes to. As usual, the access point responds with an authentication challenge in the clear. The only thing the attacker needs to do is to take this challenge, encrypt it by XORing it with the keystream, and send the result (ciphertext) back to the access point. This response must be approved because the keystream it used is the one shared with the AP. Consequently, the attacker will be allowed to join the network.

So far, what we have discussed is based on an assumption that the access point is in a privileged position. In other words, the authentication is a one-way authentication instead of authenticating each other (mutual authentication). 802.11 does not restrict authentication to any particular scenario. Any station can authenticate with any other station. So the two parties during an authentication should be treated as peer to peer, or let's say they should authenticate each other. But 802.11 does not restrict a mutual authentication. 802.11 implicitly assumes that access points are in a privileged position by virtue of the fact that they are typically under control of network administrators. Network administrators may wish to authenticate mobile stations to ensure that only authorized users access the 802.11 networks, but mobile stations can't authenticate the access point. Stations wishing to join a network must authenticate to it, but networks are under no obligation to authenticate themselves to a station. The designers of 802.11 probably felt that access points are part of the network infrastructure and thus in a more privileged position, but this curious omission makes a man-in-the-middle attack possible. A rogue access point could certainly send beacon frames for a network that it is not a part of and attempt to steal authentication credentials.

Therefore, mutual authentication is a requirement in the case of a man-in-the-middle attack. In the following chapter, we will see how TKIP adapts 802.1X to do the mutual authentication for WLAN. In the proposed protocol in chapter 4, mutual authentication is a requirement, as well.

2.4 Summary

In this chapter, 802.11's privacy and authentication, as well as their flaws have been well described. The flaws found in WEP is basically caused by the improper implementation of RC4 algorithm. In order to address them, keeping the secret key fresh is a requirement. Therefore, a key-distribution approach is needed and should be well designed. Since 802.11's authentication is based on a flawed WEP, it can not be treated secure. Furthermore, only an one-way authentication can not avoid the man-in-the-middle attack. Those flaws will be addressed in the following chapters, and a more efficient protocol which can not only provide key-distribution and mutual authentication, but also enhance their performance in terms of time consumption will be proposed later.

Chapter 3

Combine TLS with 802.1X in TKIP

TKIP is TGi's response for the need to improve security for already-deployed 802.11 equipments. TKIP is a suite of algorithms wrapping WEP. It adds four new algorithms to WEP:

- A cryptographic message integrity code, called Michael, to defeat forgeries;
- A new IV sequencing discipline, to remove replay attacks from the attacker's arsenal;
- A per-packet key mixing function, to de-correlate the public IVs from weak keys; and
- A rekeying mechanism, to provide fresh encryption and integrity keys, undoing the threat of attacks stemming from key reuse; at the same time, providing authentication service.

For defeating forgeries, TKIP uses a cryptographic message integrity code, called Michael. The basic idea of Michael is to apply a hash function into the plaintext to get a ICV, and verify this ICV on the receiver side. This can avoid the vulnerable property in CRC-32 such as “it is a linear and unkeyed function of the message” [3]. The detail of algorithm “Michael” is in [15].

For defeating replays, TKIP reuses the WEP IV field as a packet sequence number. Both transmitter and receiver initialize the packet sequence space to zero whenever new TKIP keys are set, and the transmitter increases the sequence number with each packet it sends. TKIP requires the receiver to enforce proper IV sequencing of arriving packets. TKIP defines a packet as out-of-sequence if its IV is the same or smaller than a previous correctly received message associated with the same encryption key. If a message arrived is out of order, then it is considered to be a replay, and the receiver discards it and increases a replay counter. For details, please refer to [16].

For defeating weak key attacks (FMS attacks), TKIP offers a new per-packet key construction, called the TKIP key mixing function, substitutes a temporal key for the WEP base key and constructs the WEP per-packet key in a novel fashion. Temporal keys are named so because they have a fixed life time and are replaced frequently. Basically, the TKIP key mixing function transforms a temporal key and packet sequence counter into a perpacket key and IV. Reference [17] specifies the key mixing function in detail. The mixing function operates in two phases, with each phase compensating for a particular WEP design flaw. Phase 1 eliminates the same key from use by all links, while phase 2 de-correlates the public IV from knowing the per-packet key.

Since my research focuses on this so-called rekeying mechanism, which provides both key-distribution and authentication, the following of this chapter will discuss this topic in detail.

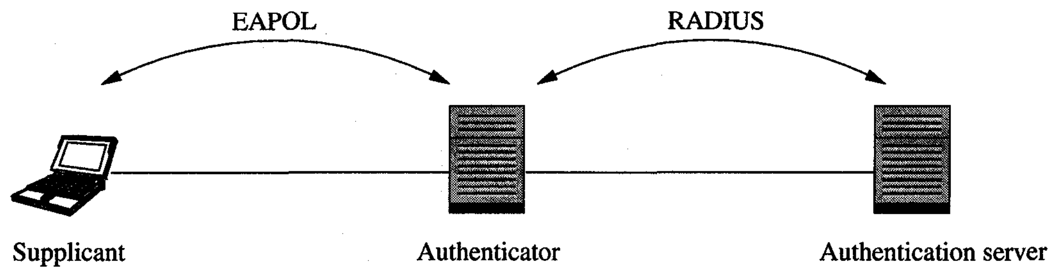

Figure 3.1: 802.1X architecture

3.1 Combine TLS and 802.1X for Authentication and Key-Distribution

As mentioned in Chapter 2, the WEP IVs can never be reused with the same key without voiding the RC4 privacy guarantees, and that the TKIP key mixing function can construct at most 2^{16} IVs. This implies that TKIP requires a key-update mechanism operating at least every 2^{16} packets. The original authentication protocol that is based on a flawed WEP is vulnerable, thus it needs to be addressed by a well designed algorithm.

3.1.1 802.1X

802.1X was originally designed for network port authentication. It defines three components to the authentication conversation: supplicant, authenticator and authentication server, which are all shown in Figure 3.1.

The supplicant is the end user machine that seeks access to network resources. Network access is controlled by the authenticator, it serves the same role as the access server in a traditional dial-up network. The authenticator does not maintain any user information. Any incoming requests are passed to an authentication server, such as a RADIUS server, for actual processing.

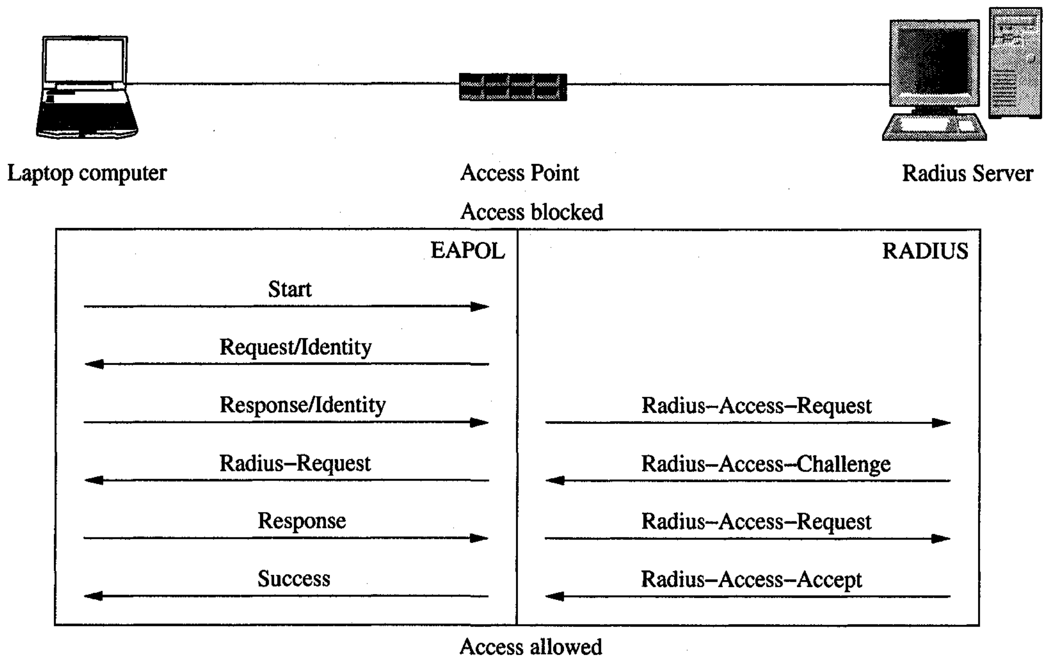

Figure 3.2: 802.1X conversation

The authentication exchange is logically carried out between the supplicant and the authentication server, with the authenticator acting only as a bridge. A derivation of EAP (Extensible Authentication Protocol) is used by the authenticator to pass challenges and reponses back and forth. From the supplicant to the authenticator, the protocol is EAP over LANs (EAPOL) or EAP over wireless (EAPOW). From authenticator to the authentication server, the protocol used is RADIUS.

802.1X is a framework, not a complete specification in and of itself. The actual authentication mechanism is implemented by the authentication server. 802.1X supplies a mechanism for issuing challenges and confirming or denying access, but it does not pass judgment on the offered credentials. A typical message exchange between supplicant and authentication server is showed in Figure 3.2.

As mentioned before, EAP is the protocol used in 802.1X for data transfer. For a particular explanation of EAP, please refer to the book [1]. Through EAP, several authentication algorithms can be chosen, such as MD-5 challenge, One-time password (OTP), Generic Token Card, and TLS.

MD-5 Challenge is defined in RFC 1994. Its requests contain a challenge to the end user. For successful authentication, the challenge must be successfully encoded with a shared secret. You can also find the explanation of OTP and Generic Token Card authentication method in RFC 1938 and [1], respectively.

TLS is a well designed protocol for both authentication and key-distribution. The following section will give a brief review about this protocol.

3.1.2 TLS Handshake Protocol

RFC 2716 describes the use of Transport Layer Security (TLS) for authentication and key-distribution. TLS is the standardized successor to the widely deployed Secure Socket Layer (SSL), and TLS authentication inherits a number of useful characteristics from SSL. Most notably, mutual authentication is possible with TLS. Rather than issuing a one-side challenge to the client, EAP-TLS can ensure that the client is communicating with a legitimate authenticator. In addition to mutual authentication, TLS provides a method to protect the authentication between the client and authenticator. It also provides a method to exchange a session key securely between the client and authenticator, which limits the impact of a compromised WEP key.

Figure 3.3 gives us a basic idea of how the TLS handshake protocol works. The exchange can be viewed as having four phases.

In phase 1, the client starts a conversation by sending a *client_hello* message including highest understood SSL version, a random number for preventing replay attacks, session ID, ciphersuite and compression method. The ciphersuite is a list that contains the combinations of cryptographic algorithms supported by the client. The algorithms include key exchange algorithms and cipherspecs (which include cipher algorithms, MAC algorithms, etc). Then, the server will send back the *server_hello*

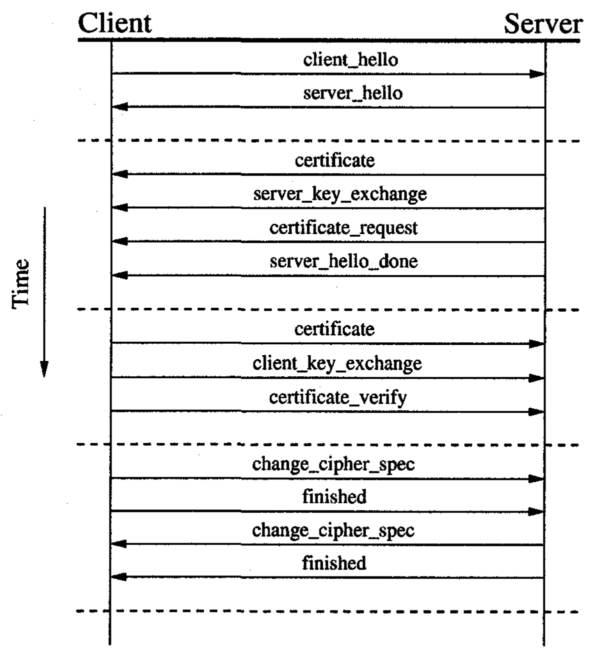

Figure 3.3: TLS handshake protocol

message which contains the same parameters as the *client_hello* message. The purpose of phase 1 is to negotiate a certain protocol version, and a bunch of certain algorithms and parameters in order to transfer the data after the handshake.

Phase 2 is called server authentication and key exchange. The server begins this phase by sending its certificate. Then a *server_key_exchange* message may be sent if it is required. If mutual authentication is needed, the server will send a *certificate_request* message. Next, a *server_done* message is sent indicating the end of the second phase.

In phase 3, as a response, the client will send back his own certificate if necessary, followed by the *client_key_exchange* message for key exchange. The *certificate_verify* message is sent to tell the server the verification result.

During phase 4, both client and server will send *change_cipher_spec* message to indicate that from now on, they will use the new negotiated algorithms and parameters to protect the transmitted data, followed by a finished message indicating the finish of the whole process.

3.2 Disadvantage of Using TLS Protocol in 802.1X

Although TLS is a well designed protocol for authentication and key-distribution, it has its own disadvantage when we apply it into WLAN.

3.2.1 The complicacy of TLS for WLAN

As mentioned earlier, TLS is originally designed for transport layer, it is compatible for most of the system. For instance, in the phase 1 of TLS handshake protocol, the client and server will negotiate a certain version of SSL in order to finish the following conversation. This is because different system may support different SSL version. TLS is the third version of SSL. When a system supporting TSL is willing to communicate with another system supporting only SSLv1, it must adjust itself to SSLv1 in order to understand every message sent by that system. But in WLAN, because we only combine TLS into 802.1X, there is only one version of SSL used for authentication and key-distribution which is TLS. Therefore, negotiating SSL version is not necessary. Also, the reason both parties negotiate a *cipher_suite* is that each system support their own key-exchange algorithm, cipher algorithm for data privacy and other parameters. During the handshake, they must find a certain bunch of algorithms and parameters which both of them support, so that after handshake, they can continue to exchange data. But again, in our case, we still use WEP as the cipher algorithm for data privacy, and for some efficiency concern which will be explained in the next chapter, we will use only one key-exchange algorithm to distribute the key. Thus, negotiating these algorithms is not necessary. Basically, in the first phase, we only need both parties to setup a session ID and send a couple of random numbers due to the prevention of replay attack.

In the second phase of TLS handshake, certificates must be sent for authentication. But *server_key_exchange* is useless. *Server_key_exchange* is only necessary for a couple of certain key exchange algorithms, such as Diffie-Hellman key exchange, which requires both parties to exchange some parameters before distribute the secret key. But in our case, we will use RSA key exchange algorithm to distribute the key due to the efficiency concern. Since RSA key exchange algorithm doesn't require any previous parameters shared by both parties before distributing the key, *server_key_exchange* can be deleted in this phase. Also, mutual authentication is a requirement in WLAN in order to avoid man-in-the-middle attack, the server doesn't need to send the *certificate_request* to ask client for a certificate.

In the third phase, because the same reason in phase 2, *client_key_exchange* can be eliminated.

During the last phase, the purpose of *change_cipher_spec* is to indicate each other that right after the authentication and key-distribution, the new negotiated algorithm and parameters will be used for data privacy. But as mentioned before, we still use WEP as the cipher algorithm, thus after handshake, both parties know which algorithm they are going to use, which means sending a message to indicate which algorithm to use is redundant.

After the analysis above, we can see that many steps in TLS are designed for traffic in transport layer. For the sake of compatibility, TLS protocol includes almost everything needed by all kinds of systems, which is why the whole process have 4 phases and many sub-phases. In our case, however, since we only deal with wireless systems which requires only a certain bunch of algorithms and parameters under a specific circumstance, we can combine the whole 4-phase TLS handshake protocol into several simple steps for WLAN. We will see this new protocol in the next chapter.

Table 3.1: Unbalanced time consumption by using different processors for RSA operation

Microprocessors	**Time Required (ms)**	
	RSA(1024) Encryption	RSA(1024) Signature Verification
Using a 20MHz microprocessor in a PDA	622	598
Using a Pentiym4 2.1GHz microprocessor	0.18	0.19

3.2.2 Unbalanced Computation Ability

In TLS, certificate is used for authentication (The concept of certificate will be described in the next chapter). Basically, the security of certificate depends on the use of digital signature technology, which is based on public key algorithms. For key-distribution in TLS, public key algorithms such as Deffie_Hellman_key_exchange and RSA_key_exchange are the best choices. However, for WLAN, it is very normal that the clients' devices for communication are always laptops, PDAs, or even cellphones. The processors' computation abilities in these devices are restricted compared to those used by servers. Figure 3.1 gives us some examples on how different the time consumptions are due to different processors. For example, We can see that by using a 20 MHz microprocessor in a PDA, the time required for doing an RSA encryption with a 1024-bit modulus is about 622 ms; to do the same operation, if use a Pentium4 2.1 GHz processor, the time required is only 0.18 ms, which is about 3456 times shorter than that in the first case. To do an RSA signature verification with a 1024-bit modulus by using both processors, we can nearly get the same result.

It is easy to understand that the time consumption for the whole authentication and key-distribution depends only on the one whoever's computation ability is weaker. In other words, no matter how fast the server's processor can execute the operations, authentication and key-distribution can only be finished by the time that the client's processor finishes its job, the spending time of which may be much longer than that of the server.

This is a very serious problem in reality. Disregarding other factors which may delay the time, when a user tries to roam from one access point to another in large infrastructure deployments, the time used for a full reauthentication and key-distribution by TLS handshake protocol used in 802.1X, such as 600ms, is too slow to support real-time applications such as audios and videos. Therefore, in order to reduce the spending time for authentication and key-distribution, we should find out how to reduce the time for client, even somehow the server's time may be increased a bit. An algorithm called "unbalanced RSA" will be used to achieve this goal and will be introduced in the next chapter.

Chapter 4

The Proposed Protocol with Unbalanced RSA

In this chapter, the proposed protocol will be introduced. We will also apply unbalanced RSA into this protocol to reduce the time consumption. Regarding the security flaws in original 802.11 standard and the disadvantage of using TLS in 802.1X for authentication and key-distribution, the ability of addressing all those problems in this proposed protocol with unbalanced RSA will be described in detail.

4.1 The Proposed Protocol

Figure 4.1 is the proposed protocol for authentication and key-distribution in WLAN. The protocol consists of several messages exchanged by client and server. Each message has three fields:

- Type: Indicates one of the messages in Table 4.1.

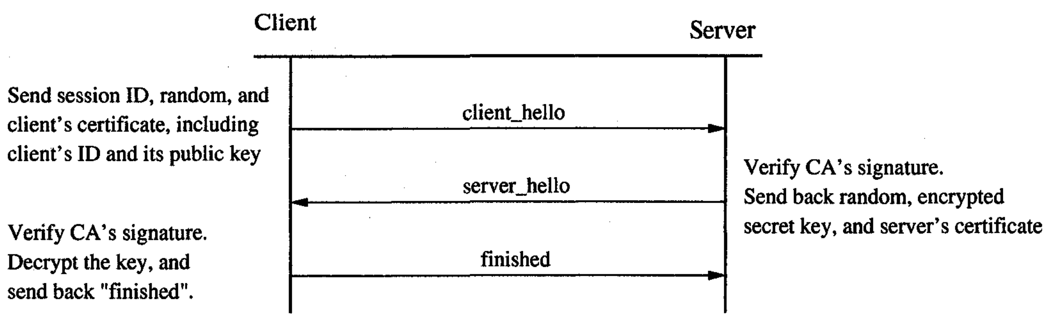
Client
Server
Send session ID, random, and client's certificate, including client's ID and its public key
client_hello
server_hello
Verify CA's signature. Send back random, encrypted secret key, and server's certificate
Verify CA's signature. Decrypt the key, and send back "finished".
finished

Figure 4.1: The proposed protocol for authentication and key-distribution

- Length: The length of the message in bytes.
- Content: The parameters associated with this message.

Table 4.1: Message Types

Message Type	**Parameters**
client_hello	session id, random, chain of X.509v3 certificates
server_hello	session id, random, chain of X.509v3 certificates, encrypted secret key
finished	null

The exchange includes three steps.

Step 1 In this step, the client will send a *client_hello* message with the following parameters:

- **Random:** A client-generated random structure, consisting of a timestamp and a sequence of random number. These values serve as nonces and are used during key exchange to prevent replay attacks.

- **Session ID:** A variable-length session identifier. A nonzero value indicates that the client wishes to update the parameters of an existing connection or create a new connection on this session. A zero value indicates that the client wishes to establish a new connection on a new session.

- **Certificate(s):** one or a chain of X.509 certificates. It will be sent for the sake of being authenticated by the server.

After receiving the *client_hello* message, the server will verify the CA's (Certificate Authority) digital signature in client's certificate. If the certificate is approved, the server will generate a secret key, and encrypt this secret key using client's RSA public key which is retrieved from client's certificate. The encryption result, ciphertext, will be sent back to the client.

Step 2 In this step, the server sends the *server_hello* message with his own random and a session ID. The random is generated by the server and is independent of the client's random. If the session ID of the client was nonzero, the same value is used by the server; otherwise the server's session ID contains the value for a new session. Following the session ID and the random, the server sends his own certificate(s), in order to be authenticated by the client. At last, the ciphertext which contains the encrypted secret key will be sent.

After getting the message from the server, the client will first verify the server's certificate. If approved, the client then decrypts the ciphertext to get the secret key. So far, both client and server have authenticated each other; and the secret key has been distributed to both parties.

Step 3 The client finally sends a *finished* message to indicate that the whole authentication and key-distribution process is done, and from now on, they can use the shared secret key to transmit data by using WEP algorithm.

4.1.1 Security Analysis

The proposed protocol can prevent the system from all the general attacks.

Interception By using well-designed RSA algorithm for key-distribution and authentication, it is extremely difficult to gain any information of the distributed key and the content of the authentication by an unauthorized party. RSA guarantees the confidentiality of the data.

Modification With the use of certificate (it will be explained in this section shortly), an unauthorized party can modify the data only if he can break the hash function, which is almost impossible.

Fabrication The public key algorithm used in the certificate and key-exchange method prevents the messages from being fabricated by an unauthorized party.

Replay As mentioned earlier, random is used to defeat replay attack during the authentication and key-distribution.

In the following of this section, we will take a closer look at the certificate and the RSA key-exchange method.

Certificate for Authentication

In our proposed protocol, in order to achieve a reliable mutual authentication, we use the well designed public-key certificate as our algorithm.

The heart of certificate is the digital signature technology. Digital signature is an application of public key algorithm. It is used to ensure that the signed message is really from the one who signed it. For instance, Bob wants to send a message to Alice and, although it is not important that the message be kept secret, he wants Alice to be certain that the message is indeed from him. Then Bob can use his own private key to encrypt the message. When Alice receives the ciphertext, she finds that she can decrypt it with Bob's public key, thus proving that the message must have been encrypted by Bob. No one else has Bob's private key and therefore no one else could have created a ciphertext that could be decrypted with Bob's public key. Therefore the entire encrypted message serves as a digital signature. In addition, it is impossible to alter the message without access to Bob's private key, so the message is authenticated in terms of both source and data integrity.

In essence, a certificate consists of a public key plus a User ID of the key owner, with the whole block signed by a trusted third party. Typically, the third party is a CA (certificate authority) that is trusted by the user community, such as a government agency or a financial institution. A user can present his or her public key to the authority in a secure manner and obtain a certificate. The user can then publish the certificate. Anyone needing this user's public key can obtain the certificate and verify that it is valid by way of the attached trusted signautre. Figure 4.2 illustrates the process.

One scheme has become universally accepted for formatting public-key certificates: the X.509 standard. X.509 certificates are used in most network security applications, as well as used in my proposed protocol. The detail of X.509 standard is in RFC 2459.

Certificate is a well designed algorithm for authentication. Because the security of certificate depends on the digital signature which is one of the public key algorithms, it can be compromised only if the public key algorithm is defeated. Public key algorithm such as RSA has been stayed secure for more than three decades. With a proper modulus, RSA will stay secure in the foreseeable future [12]. Thus, using certificate for authentication in our protocol can ensure that the man-in-the-middle attack which happened in the original 802.11 standard will be defeated.

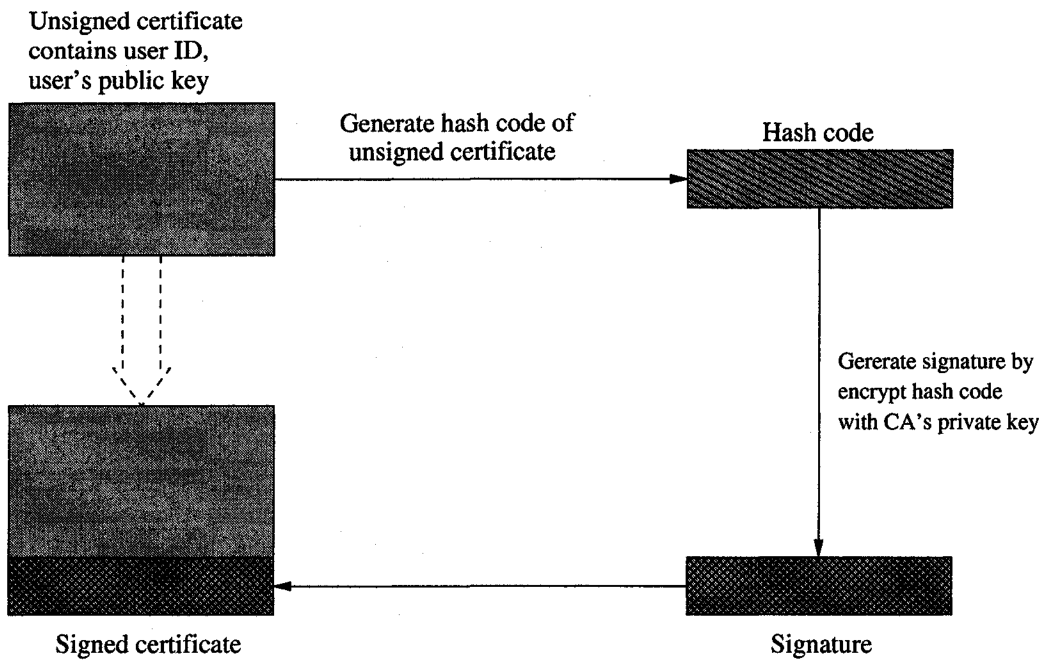
Unsigned certificate contains user ID, user's public key
Generate hash code of unsigned certificate
Hash code
Gererate signature by encrypt hash code with CA's private key
Signed certificate
Signature

Figure 4.2: Certificate

RSA Key Exchange algorithm for Key-Distribution

Normally, the two communication parties encrypt and decrypt a message using a shared secret key. The keys are decided upon in advance and somehow this information is sent securely from one to the other. There are some obvious limitations and drawbacks to pre-distribution. First of all, it requires two parties, Alice and Bob, to have met or to have established a secure channel between them in the first place. Secondly, once Alice and Bob have met and exchanged information, there is nothing they can do, other than meeting again, to change the key information in case it gets compromised.

The solution for those problems is to use public key exchange algorithm, such as RSA and Diffie-Hellman key exchange. The advantage of public key algorithm is that there is no secret information necessary to be shared before the secret key is distributed. For efficiency concern which will be discussed later in this chapter, in

Key Generation

Select p, q	p and q both prime
Calculate n=p*q	
Calculate (p−1)(q−1)	
Select integer e	gcd((p−1)(q−1), e)=1; 1<e<(p−1)(q−1)
Calculate d	d = e^(−1) (mod (p−1)(q−1))
Public key	KU = {e, n}
Private key	KR = {d, n}

Encryption

Plaintext:	M < n
Ciphertext:	C = M^e (mod n)

Decryption

Plaintext:	C
Ciphertext:	M = C^d (mod n)

Figure 4.3: RSA algorithm

our protocol, we will choose RSA key exchange algorithm to distribute the secret key.

RSA Algorithm for Key-Exchange In order to understand the RSA key exchange algorithm and for the sake of efficiency concern discussed in the next section, let's take a brief review on the RSA algorithm. Figure 4.3 shows the basic idea of RSA.

For key exchange, we treat the secret key to distribute as a plaintext and encrypt it with RSA public key. For recipient who holds the relevant RSA private key, he can simply decrypt the ciphertext and get the secret key. As mentioned before, there is no secret information required before the key is distributed.

4.1.2 Summary

As far, the proposed protocol has been described. The protocol used to address the flaws in authentication and key-distribution in 802.11 standard has been explained. As mention before, one serious problem for using TLS in 802.1X is that the client processor's computation ability is the bottleneck during the authentication and key-distribution. The solution for that is either to use more advanced processor with stronger computation ability, which depends on the development of the processor technology and will cost more money, or, to make the current algorithm more efficient. Both of their purposes are to reduce the time consumption for the whole process. In the next section, a new algorithm called "unbalanced RSA" will be introduced. Let's see how it can make the current RSA algorithm more efficient in the case that the computation ability is unbalanced, and finally reduce the whole time for authentication and key-distribution.

4.2 Unbalanced RSA

One of the most important decisions in practical implementations of the RSA cryptosystems is the choice of modulus size. It is clear that 512-bit modulus no longer provides adequate protection, and 1024 bits is a common choice in current RSA systems. With the progress of factoring algorithm, the size of modulus will increase in the foreseeable future. However, the time complexity of modulus exponentiation grows rapidly with the size of the modulus (actually, the time complexity of RSA computations grows cubically with the size of the modulus [12]), and thus it is difficult to choose a size which combines efficient operation with long term security.

"Unbalanced RSA" was first mentioned by Adi Shamir, it is based on Chinese Remainder Theory (CRT). For instance, we use a modulus n with a size of 1024 bits, and the size of its prime factor p is 256 bits. Then, the size of q is about 768 bits.

The different size of p and q reflects the name of "unbalanced".

Since RSA encryption is typically used only in order to exchange session keys for fast secret key cryptosystems, the cleartexts are usually quite short: even three independent keys for triple DES require only 168 bits, which means it is very likely that the cleartext is always smaller than p. We can thus assume that the cleartext is in the range $[0, p)$.

For RSA decryption, we need to consider the equation $m = c^d \ (mod\ n)$. If we use the Chinese Remainder Theorem, we can compute $m_1 = c^d \ (mod\ p)$ via a 256 bit exponentiation, by reducing c modulo p and d modulo $p-1$ at the beginning of the computation. However, we then have to carry out the 756 bits exponentiation $m_2 = c^d (mod\ q)$, which is $\left(\frac{768}{256}\right)^3 = 9$ times slower.

As a matter of fact, there is no need to carry out this expensive computation at all. By definition, m_1 is equal to $m \ (mod\ p)$. However, the cleartext m is known to be smaller than p, and thus $m \ (mod\ p)$ is simply m itself. By combining these observations, we conclude that m_1 is the original cleartext m, and thus it is just a waste of time to carry out the computation of m_2 modulo q, which will yield the same result.

If we compare the size of p which we use to calculate the cleartext to the modulus n which we should use without using "unbalanced RSA", the time we need is about $\left(\frac{1024}{256}\right)^3 = 64$ times shorter, which is a significant enhancement.

4.2.1 Security Analysis

From the analysis above, we can see that using "unbalanced RSA" can reduce the time for decryption which is great for client in WLAN. In this section, let's take a look at the security affection by using this algorithm.

It is known that the security of RSA relies on the difficulty of factoring the modulus n. So far, all the known factoring algorithms can be divided into two broad types:

algorithms whose running time depends on the size of the factors, and algorithms whose running time depends only on the size of the factored number n. The oldest factoring algorithms typically searched for the smallest factor p of n, and were thus of the first type. However, modern algorithms tend to use indirect approaches which require the same time to find a single digit or a fifty digit prime factor of n.

The fastest factoring algorithm of the first type is currently the elliptic curve method. Its asymptotic running time is $exp(O((ln(p))^{\frac{1}{2}} \cdot (lnln(P))^{\frac{1}{2}}))$, but its basic operations are very slow. The largest factor ever found in practice with this algorithm was about 183 bits long, and it is very unlikely that this algorithm will be able to find the 256 bit factors of 512 bit RSA keys in the next few years.

Factoring algorithms of the second type are much faster, since they can use a wider array of mathematical techniques. The best algorithm of this type is currently the general number field sieve. It has an asymptotic complexity of $exp(O((ln(n))^{\frac{1}{3}} \cdot (lnln(n))^{\frac{2}{3}}))$, and is believed to be capable of factoring a 512 bit modulus in 10000 to 15000 MIPS-years [12].

Since the inception of the RSA cryptosystem, all the record breaking factorizations of RSA keys were based on algorithms of the second type, and it is reasonable to assume that this trend will continue in the foreseeable future. If we assume this is the case, then using 256-bit p, with remaining modulus n to be 1024 bits, comparing to the regular 1024-bit n with 512-bit p, we keep the security of our unbalanced RSA to be on the same security level. Of course, this is under the condition that the prime number p is larger than 183 bits, otherwise it can be found by the first factoring algorithm, although this algorithm is much slower. In order to provide some security margin, we will use 256-bit p in our protocol, while keeping n to be 1024 bits.

4.3 Unbalanced Exponents

A well known way of speeding up the RSA encryption process $c = m^e (mod\ n)$ is to use a small encryption exponent e. Recall that during the handshake process, the client uses RSA encryption equation to verify the certificate. Therefore, we could use a small encryption exponent e to reduce the time for client during the certificate verification.

There are two common options for the public exponent e. One is to use $e = 3$, which is the smallest value can be used. By this option, the time consumption for RSA encryption on client can be reduced to the smallest. However, there are couple of drawbacks: What if the cleartext m is less than $n^{\frac{1}{e}}$, or, $m^e \leq n$? If so, recalling the RSA encryption equation $c = m^e\ (mod\ n)$, there is no modulation operation involved in that equation. Actually, we can find m directly by calculating $m = c^{\frac{1}{e}}$. When e is as small as 3, it is really not difficult to find m [18]. Also, Hastad [19] shows that small public exponents can be dangerous when the same plaintext is sent to many different recipients.

Regarding to those drawbacks, the other option of the public exponent e is to use $2^{16} + 1 = 65537$, which is chosen due to some implementation concern. But in our case, since we only use low public exponent during the certificate verification, which actually depends on digital signature scheme. The cleartext to be signed is not something we want to protect from recovering, but rather the message which we want the recipient to believe is from its original signer. In other words, the drawbacks mentioned just now which threat the exposure of the cleartext is not a problem in our case. Therefore, we can use $e = 3$ without any worry.

4.4 Summary

In this chapter, the proposed protocol for authentication and key-distribution in WLAN is introduced. Certificate and RSA key exchange algorithm are also reviewed briefly, which is good for the reader in order to understand how this protocol can address the flaws in 802.11 standard. Then, the concept of "unbalanced RSA" was presented. We can see that using "unbalanced RSA" can reduce the RSA decryption time to 64 times shorter under the given condition, which is great for client during the key-distribution process. More importantly, the security level is kept as high as the regular RSA even we use a short prime number p. For certificate verification on client, we can use a low public exponent $e = 3$ in order to reduce the time consumption without causing any flaws.

In the next chapter, some simulation work will be introduced according to the proposed protocol and the "unbalanced RSA" used in WLAN.

Chapter 5

Implementation

In this research, I used Java to implement my proposed protocol with the "unbalanced RSA" on application layer, the purpose of which is to prove the feasibility of my idea. Then in the next step, we can adapt it to data link layer used in WLAN.

5.1 Implementation Detail

5.1.1 Certificate Generation Class

In order to use certificate as the method for authentication, we need to generate certificate for CA. In reality, users can get a certificate or a chain of certificates from CA. For simplification, we will generate a self-signed certificate instead of a certificates chain. As mentioned before, we use the wide-applied X.509 standard certificate in our protocol. Noting that X.509 standard is not included in java.sun package, we need to download a package from other supplier which has this standard, such as BouncyCastle.

In the certificate generation class, we need to import the following packages, as in the codes:

```
import java.math.*;
import java.util.*;
import org.bouncycastle.jce.*;
import java.security.*;
import java.security.spec.*;
import org.bouncycastle.asn1.x509.*;
import java.security.cert.*;
import java.io.*;
```

There is a class in Java called BigInteger, which is designed for very large integers such as those used in cryptography. We generate the prime number p and q, using the BigInteger's constructor BigInteger(size, randomSeed). Then choose the public exponent to be 3, and calculate the private exponent.

After getting the public key $\{e, n\}$ and private key $\{d, n\}$, we convert them into the public key object and secret key object: *pubkey* and *prikey*, respectively. Then use the methods and constructors provided by BouncyCastle to generate the self-signed certificate:

```
X509V1CertificateGenerator certificateGenerator =
    new X509V1CertificateGenerator();
certificateGenerator.setSerialNumber(BigInteger.ONE);
String x509Name = "CN=SomeName";
certificateGenerator.setIssuerDN(new X509Name(x509Name));
certificateGenerator.setSubjectDN(new X509Name(x509Name));
Calendar nextYearDate = Calendar.getInstance();
nextYearDate.add(Calendar.YEAR, 1);
certificateGenerator.setNotAfter(nextYearDate.getTime());
certificateGenerator.setNotBefore
    (Calendar.getInstance().getTime());
certificateGenerator.setSignatureAlgorithm("SHA1withRSA");
certificateGenerator.setPublicKey(pubkey);
```

```
String alias = Long.toHexString
    (SecureRandom.getInstance("SHA1PRNG").nextLong());
X509Certificate certificate =
    certificateGenerator.generateX509Certificate(prikey);
```

Then, we can store the certificate, private exponent d, prime number p into different files for client to use later.

(The complete code of certificate generation class is in Appendix C)

5.1.2 Class for Client

First of all, we need to setup a socket for client and server to communicate.

```
try {
    clientSocket = new Socket("localhost", 4444);
    out = clientSocket.getOutputStream();
    in = new
          BufferedInputStream(clientSocket.getInputStream());
} catch (UnknownHostException e) {
    System.err.println("Don't know about host: localhost.");
    System.exit(1);
} catch (IOException e) {
    System.err.println("Couldn't get
                        I/O for the connection to: localhost.");
    System.exit(1);
}
```

Then, as the first step in the proposed protocol which was introduced in the last chapter, client sends its *hello_ message* to the server, which includes session ID, random and client's certificate. The client can do this by using FileInputStream or ObjectInput/OutputStream. (The detail of this part of code is in Appendix A.)

In the second step for client, he expects a message from server which includes server's certificate and the encrypted secret key. The client then retrieve CA's public key from CA's own self-signed certificate:

```
FileInputStream CAFis = new FileInputStream("CAcertificate.cer");
BufferedInputStream CABis = new BufferedInputStream(CAFis);
CertificateFactory CACf = CertificateFactory.getInstance("X.509");
java.security.cert.Certificate CACert = CACf.generateCertificate(CABis);
PublicKey CApublickey = CACert.getPublicKey();
```

The purpose to get CA's public key is to verify server's certificate. This is done by using the verification method in certificate class:

```
serverCert.verify(CApublickey);
```

When we generate the server's certificate, we have already set the public exponent e to be 3, as analysed before, to reduce the time consumption of certificate verification for client.

If the certificate is approved, the client can decrypt the ciphertext to get the secret key by doing:

```
plaintext = ciphertext.modPow(d, p);
```

In this line of code, modPow(exponent, modulus) is the method in BigInteger which operate the following equation:

$$P = C^{exponent} (mod\ modulus)$$

Actually, here we apply "unbalanced RSA" to use p instead of n to be the modulus for RSA decryption on client, in order to reduce the time consumption. Afterward, the client saves the secret key in a file or somewhere else for later use of transferring data using secret key algorithm.

At this point, the authentication and key-distribution have been successfully done with the use of "unbalanced RSA" to speed up the whole process for client. The client then sends back a message indicating the success of the handshake.

(The complete code is in Appendix A.)

Table 5.1: Time consumption for key-distribution

	Regular RSA	RSA for paranoids
Modulus n (bits)	1024	1024
p (bits)	512	256
q (bits)	512	768
Time for RSA decryption	$m = c^d (mod\ n)$	$m = c^d (mod\ p)$
on client (ms)	65.19	1.71

5.1.3 Class for Server

The code for the server is almost identical to the client's. The difference is that there is no "unbalanced RSA" involved. The only step server needs to do is receiving client's *hello_message*, verifying client's certificate with CA's public key, generating a secret key, and encrypting this key with client RSA public key retrieved from client certificate. The complete code can be found in Appendix B.

5.2 Simulation Result

The Java program was tested on a laptop with an Intel Celeron 1.33GHz CPU and 240 MB RAM. The operation system is Windows XP Home Edition. There was no other process running while executing the codes. We calculated the time consumption by taking an average of 1000 times handshakes in order to get a more accurate result.

Table 5.1 is the simulation result of time consumption for key-distribution (RSA decryption). As we can see in the table, the time consumption for RSA decryption on client by using "unbalanced RSA" is about 38 times shorter than that of using regular RSA. It is obviously a significant improvement for handshake time consumption, in the case of unbalanced computation abilities for server and client.

Table 5.2: Time consumption for certificate verification

	Time for executing $c = m^e (mod\ n)$ during the certificate verification (ms)
$e = 3$	0.075
$e = 65537$	0.341

Table 5.2 is the simulation result of time consumption for certificate verification (RSA encryption). This table gets the result that verifying certificate with a public exponent $e = 3$ can reduce the time consumption by a factor of $\frac{0.341}{0.075} = 4.54$, comparing to $e = 65537$.

5.2.1 Result Analysis

As mentioned in chapter 4, under the fact that "time complexity of RSA computations grows cubically with the size of modulus" [12], the time result by using "unbalanced RSA" should be $\left(\frac{1024}{256}\right)^3 = 64$ times shorter than that of the regular one. But in my simulation, we only got about 38 times better than original one instead of 64. In order to find out the reason, we need to take a look at the number of operations when Java executes the RSA encryption and decryption.

As a matter of fact, the RSA encryption and decryption operations which involves modulus exponentiation operations is executed by $modPow(exponent, modulus)$, which is a method in the class BigInteger in Java. If we take a look at the source code of Java, we see that the core of this method is the "window algorithm" combined with Montgomery exponentiation algorithm. According to [20], the "window algorithm" is somehow difficult to calculate the number of operations, because of the indeterminacy of the window size. However, we can calculate the number of multiplications of Montgomery exponentiation, which is a similar method to the "window algorithm" except no window size needed.

The reference [20] proves that the expected average number of multiplications to compute $x^e(mod\ m)$ by algorithm “Montogomery exponentiation” is

$$3l(l+1)(t+1),$$

where l is the length of modulus and t is the length of exponent.

Note that this expression is an average result, which is under one condition that the number of ones in the binary form of exponent equals to the number of zeros. For the worst case, where all the digital number of binary exponent are ones, the expected number of multiplications to compute $x^e(mod\ m)$ by algorithm “Montogomery exponentiation” is

$$4l(l+1)(t+\frac{3}{4}).$$

And for the best case where all the digital number of binary exponent are zero except the most left one, the expected number of multiplications is

$$2l(l+1)(t+\frac{5}{2}).$$

Based on the analysis above, we can calculate the number of operations by doing a regular RSA decryption with 1024-bit modulus and by using “unbalanced RSA” in my protocol, as shown in Table 5.3.

We can see that the best time ratio between regular RSA and “unbalanced RSA” is

$$\frac{Regular\ RSA_{(worst\ case)}}{Unbalanced\ RSA_{(best\ case)}} = 128,$$

and the worst ratio is

$$\frac{Regular\ RSA_{(best\ case)}}{Unbalanced\ RSA_{(worst\ case)}} = 32.$$

Therefore, due to the different values of private exponent d used during RSA decryption, the ratio between 128 and 32 are all acceptable.

Table 5.3: Number of operations in RSA for data decryption

	General RSA algorithm			RSA for Paranoids		
	best case	**average**	**worst case**	**best case**	**average**	**worst case**
Length of modulus n (L_n bits)	1024	1024	1024	1024	1024	1024
Length of p (bits)	512	512	512	256	256	256
Length of q (bits)	512	512	512	768	768	768
Length of public exponent e, L_e (bits)	2 ($e = 3$)	2	2	2	2	2
length of private exponent d, L_d (bits)	1022	1022	1022	1022	1022	1022
Number of multiplications in $m = c^d(mod\ x)$	$2 * 1024$ $*(1024 + 1)$ $*(1022 + \frac{5}{2})$	$3 * 1024$ $*(1024 + 1)$ $*(1022 + 1)$	$4 * 1024$ $*(1024 + 1)$ $*(1022 + \frac{3}{4})$	$2 * 256$ $*(256 + 1)$ $*(256 + \frac{5}{2})$	$3 * 256$ $3 * 256 * (256 + 1)$ $*(256 + 1)$	$4 * 256$ $*(256 + 1)$ $(256 + \frac{3}{4})$

Table 5.4: Number of operations in RSA for certificate verification

	RSA encryption ($e = 65537$)	**RSA encryption** ($e = 3$)
Number of multiplications in $c = m^e (mod\ n)$	$1024 * (1024 + 1) * (2 * 17 + 7)$	$1024 * (1024 + 1) * (2 * 2 + 7)$

For certificate verification (RSA encryption) in our protocol, we can apply the same approach to find out the reason of getting a ratio of 4.54 instead of $\frac{17}{2} = 8.5$, which is based on the assumption that "time complexity of RSA computations is linear with the length of exponent" [12].

Based on Table 5.4, since the binary public exponent 65537 has two ones, as same as that of the binary public exponent 3, we can calculate the number of operations of executing RSA encryption, and the time ratio is $\frac{2*17+7}{2*2+7} = 3.73$. The point shown here is that the ratio of number of operations could be much different as the average result according to different number of ones in exponent.

5.3 Summary

In this chapter, the simulation of my proposed protocol has been described, as well as the simulation result. The proposed protocol has been proved practicable. The time consumption by using "unbalanced RSA" is about 38 times shorter than that of a regular RSA according to my simulation result. By applying $e = 3$ in certificate verification, the calculation speed is about 5 times faster than using $e = 65527$, as we expected in chapter 4.

Chapter 6

Conclusion

In this thesis, the problems in IEEE 802.11 standard in terms of authentication and key-distribution have been reviewed in detail. The lack of proper key-distribution method and reliable authentication algorithm is critical in 802.11. As we can see in chapter 3, by applying TLS into 802.1X, the flaws can be addressed. However, since TLS is not designed specifically for WLAN, there are some redundant steps during the TLS handshake if used for 802.11. For wireless environment, it is common that the computation abilities between clients and servers are severely different, which makes the client's device a bottleneck during the authentication and key-distribution process.

In order to eliminate the redundant steps in TLS for WLAN and balance the time consumption between clients and servers, a new protocol for authentication and key-distribution is proposed. Comparing to TLS handshake, this new protocol is much simpler and more suitable for 802.11. Furthermore, in order to reduce the time consumption for client during the handshake, "unbalanced RSA" is applied in this protocol. With the use of "unbalanced RSA", the time required can be reduced by a factor of 38 for key distribution; using low public exponent can reduce the time by a factor of 4.5 for certificate verification, according to my simulation, respectively. The use of "unbalanced RSA" makes the protocol much more efficient than before.

In summary, my contribution in this research can be concluded as following:

- Propose a new protocol for authentication and key-distribution for 802.11
  - Address the flaws in terms of key-distribution and authentication found in original 802.11 standard
  - Much simpler than the TLS handshake protocol
- Apply "unbalanced RSA" into the proposed protocol
  - Reduce the time consumption for key-distribution

Although we cut down the size of prime factor p during the use of "unbalanced RSA", the security is kept as strong as the original one. This is because we remain the size of modulus n to be 1024 bits as regular, which is the only factor that can affect the most efficient factoring algorithm. On the other hand, using short public exponent $e = 3$ will not affect the security level of the RSA algorithm, especially in the case of digital signature.

In terms of the trade-off of this proposed protocol, the lack of compatibility is definitely one. Since this protocol is designed specifically intending to address the flaws in 802.11 and balance the time consumption, it can not be used anywhere else. If we want to make it more compatible, we have to negotiate those algorithms and parameters as TLS handshake did, which makes it more complicated.

Since the secret keys shared by clients and servers need to be distributed frequently, and every pair of client and server has its own secret key, an AP needs to remember all the secret keys being used. The ability of remember how many different keys depends only on the memory of the AP. The more memory an AP has, the more clients it can handle.

The whole protocol is under the assumption that both client and server have already obtained the certificates from a CA. In reality, we have to think about how to distribute these certificates quickly and safely. And, how to authenticate the client and the server by a CA and how to obtain the information which need to be certificated are another two problems. Further more, we need to consider how to generate secrets keys on the server side, which surely needs some kind of secret key generator. All those problems can be treated as open problems and future work after this thesis.

Appendix A

Class for Client

```
import java.io.*;
import java.net.*;
import java.security.*;
import java.security.cert.*;
import javax.crypto.*;
import java.security.spec.*;
import java.math.*;
import java.security.interfaces.*;
import java.util.*;

public class client8021x {
    public static void main (String[] args) throws IOException,
CertificateException, NoSuchAlgorithmException,InvalidKeyException,
IllegalBlockSizeException,NoSuchProviderException, BadPaddingException,
NoSuchPaddingException, KeyStoreException, UnrecoverableKeyException,
InvalidKeySpecException, SignatureException, FileNotFoundException {

/*create a connection and do following things*/
        Socket clientSocket = null;
        BufferedInputStream in = null;

        byte[] buf = new byte[1024];
        int r=0;
        OutputStream out = null;

        try {
            clientSocket = new Socket("localhost", 4444);
            out = clientSocket.getOutputStream();
            in = new BufferedInputStream(clientSocket.getInputStream());
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: localhost.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to:
                                                        localhost.");
            System.exit(1);
        }

        FileInputStream clientCertFis = new FileInputStream("certi.cer");
        BufferedInputStream clientCertBis = new BufferedInputStream(clientCertFis);

        while((r = clientCertBis.read(buf, 0, buf.length)) != -1) {
            out.write(buf, 0, r);
        }

        System.out.println("\nStep 1: Send client's certificate to
                                        the server: Done!\n");
        System.out.println("----------------------------------------------\n");
////////////////////////////////////////////////////////////////////
        System.out.println("\nStep 2: (1)Receive server's certificate;\n
                                  (2)Verify it;\n
                                  (3)If the certificate is valid,
                                     then receive the encrypted secret key
                                     and decrypt it;\n
                                  (4)Send back a message indicating
                                     success.\n");
//receive server's certificate
        CertificateFactory serverCf = CertificateFactory.getInstance("X.509");
        java.security.cert.Certificate serverCert
                                   = serverCf.generateCertificate(in);

        System.out.println("(1)The received server's certificate is:\n");

        System.out.println(serverCert.toString());

///get CA's public key
        FileInputStream CAFis = new FileInputStream("certi.cer");
        BufferedInputStream CABis = new BufferedInputStream(CAFis);
        CertificateFactory CACf = CertificateFactory.getInstance("X.509");
        java.security.cert.Certificate CACert =
                                 CACf.generateCertificate(CABis);

        PublicKey CApublickey = CACert.getPublicKey();

///verify server's cert

        Date now1 = new Date();
        long time1 = now1.getTime();
        try {
            serverCert.verify(CApublickey);
        } catch (SignatureException e) {
            System.err.println("the certificate is fake");
            System.exit(1);
        } catch (CertificateException e) {
            System.err.println("CertificateException");
        } catch (NoSuchAlgorithmException e) {
            System.err.println("NoSuchAlgorithmException");
            System.exit(1);
        } catch (InvalidKeyException e) {
            System.err.println("InvalidKeyException");
            System.exit(1);
        } catch (NoSuchProviderException e) {
            System.err.println("NoSuchProviderException");
            System.exit(1);
        }

        Date now2 = new Date();
        long time2 = now2.getTime();

        long veriperiod = time2 - time1;
///the end of verify

        System.out.println("\n\n(2)The server's certificate is valid!\n");

/*receive the encrypted ciphertext*/

        //receive the length first

        DataInputStream in2 = null;
        try {
            in2 = new DataInputStream(clientSocket.getInputStream());
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to:
                                                          localhost.");
            System.exit(1);
        }
        int ciphertextlength = 0;
        ciphertextlength = in2.readInt();

        //System.out.println("the ciphertextbytearraysize is: " +
                                            ciphertextlength);

        //receive the cipher
        byte[] ciphertextbytearray = new byte[ciphertextlength];
        in.read(ciphertextbytearray,0,ciphertextbytearray.length);
        /*System.out.println("the received ciphertext is: " +
                                            (new String(ciphertextbytearray)));*/
        System.out.println("(3)The received ciphertext is: \n\n" +
                                                   (new BigInteger(ciphertextbytearray)));

/*Decrypt the ciphertext*/
        BigInteger ciphertext = new BigInteger(ciphertextbytearray);
        //RSA decryption
          //get p from p.file
        FileInputStream fisp = new FileInputStream("p");
        DataInputStream disp = new DataInputStream(fisp);
        int plength = disp.readInt();
        //disp.close();
        byte[] pbytearray = new byte[plength];
        fisp.read(pbytearray);
        BigInteger p = new BigInteger(pbytearray);

        //disp.close();
        fisp.close();
        //BigInteger p = cgcobj.getp();

          //get d from d.file
        FileInputStream fisd = new FileInputStream("d");
        DataInputStream disd = new DataInputStream(fisd);
        int dlength = disd.readInt();
        //disd.close();
        byte[] dbytearray = new byte[dlength];
        fisd.read(dbytearray);
        BigInteger d = new BigInteger(dbytearray);
        fisd.close();

          //get n from certificate
        FileInputStream nFis = new FileInputStream("certi.cer");
        BufferedInputStream nBis = new BufferedInputStream(nFis);
        CertificateFactory nCf = CertificateFactory.getInstance("X.509");
        java.security.cert.Certificate nCert = nCf.generateCertificate(nBis);

        java.security.interfaces.RSAPublicKey nPub =
             (java.security.interfaces.RSAPublicKey)nCert.getPublicKey();
        //PublicKey CApublickey = CACert.getPublicKey();

        BigInteger n = nPub.getModulus();

     /*decryption using RSA and calculate the time*/
        Date current1 = new Date();
        long tbefore = current1.getTime();

        BigInteger plaintext = new BigInteger("0");
        for (int i=1; i<=100; i++)
            plaintext = ciphertext.modPow(d, p);
        Date current2 = new Date();
        long tafter = current2.getTime();

        long period = tafter - tbefore;

        System.out.println("   The decrypted plaintext is: \n\n" + plaintext);

/*write the plaintext into the file receivedSecretkey.txt*/

        FileWriter fw = new FileWriter("receivedSecretkey.txt");
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write(plaintext.toString(),0,(plaintext.toString()).length());
        bw.flush();
        System.out.println("The secret key has been written
                            into the file 'receivedSecretkey.txt'\n");

///////////////////////////////////////////////////////////////

/*send back the string: "The authentication and key distribution
                         are successfully done!"*/
        PrintWriter out2 = null;
        try {
            out2 = new PrintWriter(clientSocket.getOutputStream(),true);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to:
                                localhost.");
            System.exit(1);
        }
        out2.println("The authentication and key distribution
                      are successfully done!");

        out.flush();
        out.close();
        in.close();
        in2.close();
        out2.flush();
        out2.close();
        clientSocket.close();
        System.out.println("(4)A success message has been sent
                             to the server!");
        System.out.println("-------------------------------------------");
        System.out.println("Time record:");
        System.out.println("The time used for decryption using RSA is (ms): "
                            + ((float)period)/100);
        System.out.println("The time used for RSA digital signature verification
                            is (ms): " + veriperiod);
    }
}
```

Appendix B

Class for Server

```
import java.net.*;
import java.io.*;
import java.security.cert.*;
import java.security.*;
import javax.crypto.*;
import java.security.interfaces.*;
import java.security.spec.*;
import java.math.*;

public class server8021x {
    public static void main(String[] args) throws IOException,
CertificateException, NoSuchAlgorithmException,InvalidKeyException,
IllegalBlockSizeException, NoSuchProviderException, BadPaddingException,
NoSuchPaddingException {

        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Could not listen on prot: 4444.");
            System.exit(1);
        }

        Socket clientSocket = null;
        try {
            clientSocket = serverSocket.accept();
        } catch (IOException e) {
            System.err.println("Accept failed.");
            System.exit(1);
        }
```

```
//get client's cert from client
        BufferedInputStream in = new BufferedInputStream(
                                     clientSocket.getInputStream());

        CertificateFactory clientCf
                                    = CertificateFactory.getInstance("X.509");
        java.security.cert.Certificate clientCert
                                                = clientCf.generateCertificate(in);

        System.out.println("Step 1:");
        System.out.println("(1)Receive client's certificate;\n");
        System.out.println("   Client's certificate is:\n\n");

        System.out.println(clientCert.toString());
/*get client's publickey from its cert*/
        java.security.interfaces.RSAPublicKey clientPub
            = (java.security.interfaces.RSAPublicKey)clientCert.getPublicKey();

///get CA's public key

        FileInputStream CAFis = new FileInputStream("certi.cer");
        BufferedInputStream CABis = new BufferedInputStream(CAFis);
        CertificateFactory CACf = CertificateFactory.getInstance("X.509");
        java.security.cert.Certificate CACert
                                                = CACf.generateCertificate(CABis);

        PublicKey CApublickey = CACert.getPublicKey();

///verify client's cert
        try {
            clientCert.verify(CApublickey);
        } catch (SignatureException e) {
            System.err.println("the certificate is fake");
            System.exit(1);
        } catch (CertificateException e) {
            System.err.println("CertificateException");
            System.exit(1);
        } catch (NoSuchAlgorithmException e) {
            System.err.println("NoSuchAlgorithmException");
            System.exit(1);
        } catch (InvalidKeyException e) {
            System.err.println("InvalidKeyException");
            System.exit(1);
        } catch (NoSuchProviderException e) {
            System.err.println("NoSuchProviderException");
            System.exit(1);
        }
///the end of verify
```

```
        System.out.println("\n\n(2)Verify client's certificate:\n
                                   The client's certificate is valid\n");
/*encrypt the secret key using client's publickey*/

/*Create the cipher*/
        Cipher rsaCipher = Cipher.getInstance("RSA", "BC");
  /*Initialize the cipher for encryption*/
        rsaCipher.init(Cipher.ENCRYPT_MODE, clientPub);
  /*Cleartext*/

        FileReader cleartextFr = new FileReader("secretkey.txt");
        BufferedReader cleartextBr = new BufferedReader(cleartextFr);
        StringBuffer cleartextSb = new StringBuffer();
        String line = " ";
        while ((line = cleartextBr.readLine()) != null)
            cleartextSb.append(line);
        cleartextBr.close();
        String cleartextString = cleartextSb.toString();
        BigInteger plaintext = new BigInteger(cleartextString);


  /*Encrypt the cleartext*/
        BigInteger e, n;
        BigInteger ciphertext;
        e = clientPub.getPublicExponent();
        n = clientPub.getModulus();
        //RSA encryption
        ciphertext = plaintext.modPow(e, n);


        byte[] cipherByteArray = ciphertext.toByteArray();


////////////////////////////////////////////////////////////
///send back server's certificate

        byte[] buf = new byte[1024];
        int r=0;
        OutputStream out = null;

        try {
            out = clientSocket.getOutputStream();
        } catch (UnknownHostException ee) {
            System.err.println("Don't know about host: localhost.");
            System.exit(1);
        }

        FileInputStream serverCertFis = new FileInputStream("certi.cer");
        BufferedInputStream serverCertBis = new BufferedInputStream(serverCertFis);

        while((r = serverCertBis.read(buf, 0, buf.length)) != -1) {
            out.write(buf, 0, r);
        }

        System.out.println("(3)Send back server's own certificate: Done!\n");

    /*Send the encrypted ciphertext to the client*/

        //System.out.println("the ciphertext length is:" + cipherByteArray.length);

        //send the ciphertext length first
        DataOutputStream out2 = null;
        try {
            out2 = new DataOutputStream(clientSocket.getOutputStream());
        } catch (UnknownHostException ee) {
            System.err.println("Don't know about host: localhost.");
            System.exit(1);
        }
        out2.writeInt(cipherByteArray.length);
        //end of sending length

        //send cipher
        out.write(cipherByteArray);

        System.out.println("(4)Encrypt the secret key and
                              send the cipher to the client: Done!\n");
        System.out.println("The original cleartext or secretkey is: \n"
                           + plaintext);
        System.out.println("The ciphertext is: \n" + ciphertext);
////////////////////////////////////////////////////////////////////////////

        //receive the success sign
        BufferedReader in2 = null;
        try {
            in2 = new BufferedReader(
                          new InputStreamReader(clientSocket.getInputStream()));
        } catch (IOException ee) {
            System.err.println("Couldn't get I/O for the connection to: taranis.");
            System.exit(1);
        }
        String fromClient;
        fromClient = in2.readLine();
        if (fromClient.equals("The authentication and key distribution
                              are successfully done!")) {
            //System.out.println("DONE!");
            //System.out.println(fromClient);
            System.out.println("\nStep 2: Receive client's success message: Done!");
        }


        out2.close();
        out.close();
        in.close();
        in2.close();
        clientSocket.close();
        serverSocket.close();
    }
}
```

Appendix C

Certificate Generation Class

```
import java.math.*;
import java.util.*;
import org.bouncycastle.jce.*;
import java.security.*;
import java.security.spec.*;
import org.bouncycastle.asn1.x509.*;
import java.security.cert.*;
import java.io.*;


public class cergen {

    static final BigInteger TWO = new BigInteger("2");
    static final BigInteger THREE = new BigInteger("3");
    static final BigInteger SECOPT = new BigInteger("65537");

    public static BigInteger nextPrime(BigInteger x) {
        if ((x.remainder(TWO)).equals(BigInteger.ZERO))
           x = x.add(BigInteger.ONE);
        while(true) {
           BigInteger xM1 = x.subtract(BigInteger.ONE);
           if (!(xM1.remainder(THREE)).equals(BigInteger.ZERO))
              if (x.isProbablePrime(10)) break;
           x = x.add(TWO);
        }
        return x;
    }

    public static void main(String[] args) throws NoSuchAlgorithmException,
 InvalidKeySpecException, SignatureException, InvalidKeyException,
FileNotFoundException, IOException, CertificateEncodingException{

        BigInteger n;
        BigInteger q;
        BigInteger d;
        BigInteger p;

        Random rnd = new Random();

        int size1 = 256;
        int size2 = 768;

        BigInteger p1  = new BigInteger(size1, rnd);//random int
                   p   = nextPrime(p1);
        BigInteger pM1 = p.subtract(BigInteger.ONE);//p minus 1
        BigInteger q1  = new BigInteger(size2, rnd);
                   q   = nextPrime(q1);
        BigInteger qM1 = q.subtract(BigInteger.ONE);
                   n   = p.multiply(q);
        BigInteger phiN= pM1.multiply(qM1);//(p-1)*(q-1)
        BigInteger e = THREE;
                   d = e.modInverse(phiN);
/*
        // printout n, e, d, p, q
        System.out.println("p is: \n" + p + "\n");
        System.out.println("q is: \n" + q + "\n");
        System.out.println("n is: \n" + n + "\n");
        System.out.println("e is: \n" + e + "\n");
        System.out.println("d is: \n" + d + "\n");
*/
        //generate the pubkey and prikey object
        RSAPublicKeySpec pubKeySpec = new RSAPublicKeySpec(n, e);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        PublicKey pubkey = kf.generatePublic(pubKeySpec);

        RSAPrivateKeySpec priKeySpec = new RSAPrivateKeySpec(n, d);
        PrivateKey prikey = kf.generatePrivate(priKeySpec);



        //generate the self-signed certificate

        X509V1CertificateGenerator certificateGenerator
                                    = new X509V1CertificateGenerator();
        certificateGenerator.setSerialNumber(BigInteger.ONE); // set serial number
        String x509Name = "CN=SomeName";
        certificateGenerator.setIssuerDN(new X509Name(x509Name));
        certificateGenerator.setSubjectDN(new X509Name(x509Name));
        Calendar nextYearDate = Calendar.getInstance();
        nextYearDate.add(Calendar.YEAR, 1); // Valid for 1 year
        certificateGenerator.setNotAfter(nextYearDate.getTime());
        certificateGenerator.setNotBefore(Calendar.getInstance().getTime());
        certificateGenerator.setSignatureAlgorithm("SHA1withRSA");
        certificateGenerator.setPublicKey(pubkey);
        String alias = Long.toHexString(
                           SecureRandom.getInstance("SHA1PRNG").nextLong());
        X509Certificate certificate =
                         certificateGenerator.generateX509Certificate(prikey);


        //System.out.println(certificate.toString());

        //write the certificate into .cer
        FileOutputStream fos = new FileOutputStream("certi.cer");
        fos.write(certificate.getEncoded());
        fos.close();

        //write p, d into file p.file and d.file
        int plength = p.toByteArray().length;
        FileOutputStream fosp = new FileOutputStream("p");
        DataOutputStream dosp = new DataOutputStream(fosp);
        dosp.writeInt(plength);
        dosp.flush();
        fosp.write(p.toByteArray());
        //dosp.close();
        fosp.close();

        int dlength = d.toByteArray().length;
        FileOutputStream fosd = new FileOutputStream("d");
        DataOutputStream dosd = new DataOutputStream(fosd);
        dosd.writeInt(dlength);
        dosd.flush();
        fosd.write(d.toByteArray());
        //dosd.close();
        fosd.close();

    }
}
```

References

[1] M. Gast, "802.11 Wireless Networks: The Definitive Guide", O'REILLY, April 2002.

[2] "IEEE Standard 802.11, Standards for Local and Metropolitan Area Networks: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification", 1999.

[3] N. Borisov, I. Goldberg, D. Wagner, "Intercepting Mobile Communications: The Insecurity of 802.11", in Proc.International Conference on Mobile Computing and Networking, ACM, pp 180-189, July 2001.

[4] S. Fluhrer, I. Mantin, A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4", in Proc.8th Annual Workshop on Selected Areas of Cryptography, pp 1-24, August 2001.

[5] W. Arbaugh, N. Shankar, Y. Wan, "Your 802.11 Wireless Network has No Clothes", IEEE Wireless Communications, Volume 9, Issue 6, pp 44-51, Dec.2001.

[6] J. Walker, "802.11 Security Series, Part 2: The Temporal Key Integrity Protocol (TKIP)", 2002. (http://cache-www.intel.com/cd/00/00/01/77/17769_80211_part2.pdf)

[7] J. Walker, "802.11 Security Series, Part 3: AES-based Encapsulations of 802.11 Data", 2002. (http://www.ida.liu.se/ TDDC03/literature/wireless/intel-aes.pdf)

[8] "IEEE Standard for Local and metropolitan area networks – Port-Based Network Access control", June 2001.

[9] W. Stallings, "Cryptography and Network Security, Principles and Practice", Second Edition, Prentice-Hall, Inc. 1999.

[10] N. Daswani, "Cryptographic Execution Time for WTLS Handshakes on Palm OS Devices", Certicom Public Key Solutions, San Jose,

CA, September 2000. (http://www-db.stanford.edu/ daswani/papers/WTLSPerformancePresentation.ppt)

[11] http://www.eskimo.com/ weidai/benchmarks.html

[12] A. Shamir, "RSA for Paranoids", RSA laboratories' CryptoBytes, Volume 1, Number 3, Autumn 1995.

[13] S. Singh, "The code book: the evolution of secrecy from Mary, Queen of Scots, to quantum cryptography.", Doubleday, New York, NY, USA, 1999.

[14] W. Tutte, "FISH and I", A transcript of Tutte's lecture on June 19, 1998 at the University of Waterloo.

[15] N. Ferguson, "Michael: an improved MIC for 802.11 WEP", IEEE 802.11 doc 02-020r0, January 17, 2002.

[16] D. Stanley, "IV Sequencing Requirements Summary", IEEE 802.11 doc 02-006r2, January 18, 2002.

[17] R. Housely, D. Whiting, "Temporal key Hash", IEEE 802.11 doc 01-550r1, October 31, 2001.

[18] R. Rivest, "RSA Problem", December 10, 2003. (To appear in Encyclopedia of Cryptography and Security (Kluwer).) (http://theory.lcs.mit.edu/ rivest/RivestKaliski-RSAProblem.pdf)

[19] J. Hastad, "Solving simultaneous modular equations of low degree", SIAM J. Computing, 17:336-341, 1988.

[20] A. Menezes, P. Oorschot, S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1996. (www.cacr.math.uwaterloo.ca/hac)

VITA AUCTORIS

Zhong Zheng was born in 1980 in P.R. China. He received his Bachelor's Degree from Electrical and Computer Engineering Department in Tsinghua University in 2002. He is currently a candidate for the Master of Applied Science Degree in the Department of Electrical and Computer Engineering at University of Windsor and hopes to graduate in Summer 2004.