BASIC KEY EXCHANGE PROTOCOLS FOR SECRET KEY CRYPTOSYSTEMS UNDER CRYMPIX LIBRARY

A Thesis Submitted to
The Graduate School of Engineering and Sciences of
İzmir Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

In Computer Software

by
Sevgi Uslu

August 2007
İZMİR

We approve the thesis of **Sevgi USLU**

	Date of Signature
.... Assoc. Prof. Dr. Ahmet KOLTUKSUZ Supervisor Department of Computer Engineering İzmir Institute of Technology	27 August 2007
.... Prof. Dr. Şaban EREN Department of Computer Engineering Maltepe University	27 August 2007
.... Dr. Serap ATAY Department of Computer Engineering İzmir Institute of Technology	27 August 2007
.... Prof. Dr. Sıtkı AYTAÇ Head of Department İzmir Institute of Technology	27 August 2007

....
Prof. Dr. M.Barış ÖZERDEM
Head of the Graduate School

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Assoc. Prof. Dr. Ahmet Koltuksuz, for his guidance, patience and encouragement. He was the one who supported me when I was in trouble with critical decisions. His valuable support and confidence have been the driving force of this thesis work.

Furthermore, I had the pleasure of working with Dr. Serap Atay who helped me in understanding the elliptic curve arithmetic and documentation of the thesis.

I would also like to thank Hüseyin Hışıl, Evren Akalp, Ali Mersin, Mutlu Beyazıt and Selma Tekir who cooperated with me in many studies.

Finally, I should thank to my parents who always supported me throughout my education as well as in my graduate study.

ABSTRACT

BASIC KEY EXCHANGE PROTOCOLS FOR SECRET KEY CRYPTOSYSTEMS UNDER CRYMPIX LIBRARY

Key exchange protocols are developed in order to overcome the key distribution problem of symmetrical cryptosystems. These protocols which are based on various algebraic domains are different implementations of public-key cryptography. In this thesis, the basic key exchange protocols are reviewed and CRYMPIX[1] implementations of them are provided. CRYMPIX has a portable structure that provides platform independence for generated code. Hence, the implemented key exchange mechanisms are suitable to be used on different hardware and software platforms.

[1] CRYMIX is a multiprecision cryptographic library and available at http://crympix.iyte.edu.tr.

ÖZET

CRYMPIX KÜTÜPHANESİ ALTINDA GİZLİ ANAHTAR KRİPTOSİSTEMLERİ İÇİN TEMEL ANAHTAR DEĞİŞİM PROTOKOLLERİ

Anahtar değişim protokolleri simetrik kripto sistemlerdeki anahtar dağıtım problemini çözmek amacıyla geliştirilmiştir. Çeşitli matematiksel temeller üzerine geliştirilen bu protokoller, açık anahtar kripto sisteminin farklı uygulamalarıdır. Bu tezde, temel anahtar değişim protokolleri incelenmiş ve CRYMPIX[2] uygulamaları verilmiştir. Taşınabilir bir yapıya sahip olan CRYMPIX kütüphanesinin kullanımı platform bağımsızlığı sağlamıştır. Böylece oluşturulan anahtar değişimi mekanizmaları farklı donanım ve yazılımlarla bir arada çalışmaya uygun hale gelmiştir.

[2] CRYMPIX çok-basamaklı (multiprecision) bir kütüphanesidir ve http://crympix.iyte.edu.tr adresinden ulaşılabilinir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Security of information has been a critical issue of mankind for centuries. Especially secrecy and authentication are the most common problems of security. Different methods have been developed to overcome these two problems. Various encoding and decoding schemes have been used to provide secrecy; Caesar's cipher (substitution cipher) is one of the earliest and well known methods in this regard. As for authentication signatures have common utilization.

The science that deals with the security of information, is called "Cryptology", that consist of the words, cryptos (= hidden) and logos (=word). Cryptology is the study of two different concepts; cryptography and cryptanalysis. Both of them are actually the study of mathematics. Cryptography deals with creating a secure system that is based on mathematical problems. Cryptanalysis is the process of breaking this system using mathematics, statistics, etc.

The improvement of communication technologies, especially during World War II and past twenty years, led to a revolution in cryptology. Besides, mathematical fundamentals which have been studied for centuries accelerated this revolution. Two different forms of cryptography; symmetrical and asymmetrical cryptosystems, were developed during this revolution.

1.1. Symmetrical Cryptosystems

Symmetrical cryptosystems, also called conventional cryptosystems, encrypt and decrypt data using the same key, as shown in Figure 1.1. Sender encrypts the plain text P using the key and generates the cipher text C. The cipher text is sent through an insecure channel. Receiver, who has the same key with sender, gets the cipher text C and decrypts it using the key to obtain plain text P. The key must be shared before the protocol using a secure channel.

The most common algorithms used for symmetrical cryptography are stream ciphers and block ciphers. **Stream ciphers**, such as RC4, operates on plain text or cipher text as a stream of digits (bits, bytes or sometimes 32-bit words). A pseudorandom key stream is generated using the key as a seed and the key stream combined with the plain text, one digit at a time, typically using an exclusive-or. Same plain text digit generates a different cipher text digit every time it is encrypted. **Block ciphers**, such as DES, AES, operate on fixed-length bit blocks. Generally 64-bit or 128-bit blocks of plain text are encrypted into same sized blocks of cipher text. Unlike stream ciphers, same plain text block generates the same cipher text block every time it is encrypted using the same key. Consequently, there exist data patterns which lower the confidentiality. Block ciphers use techniques known as *modes of operation* to avoid this problem.

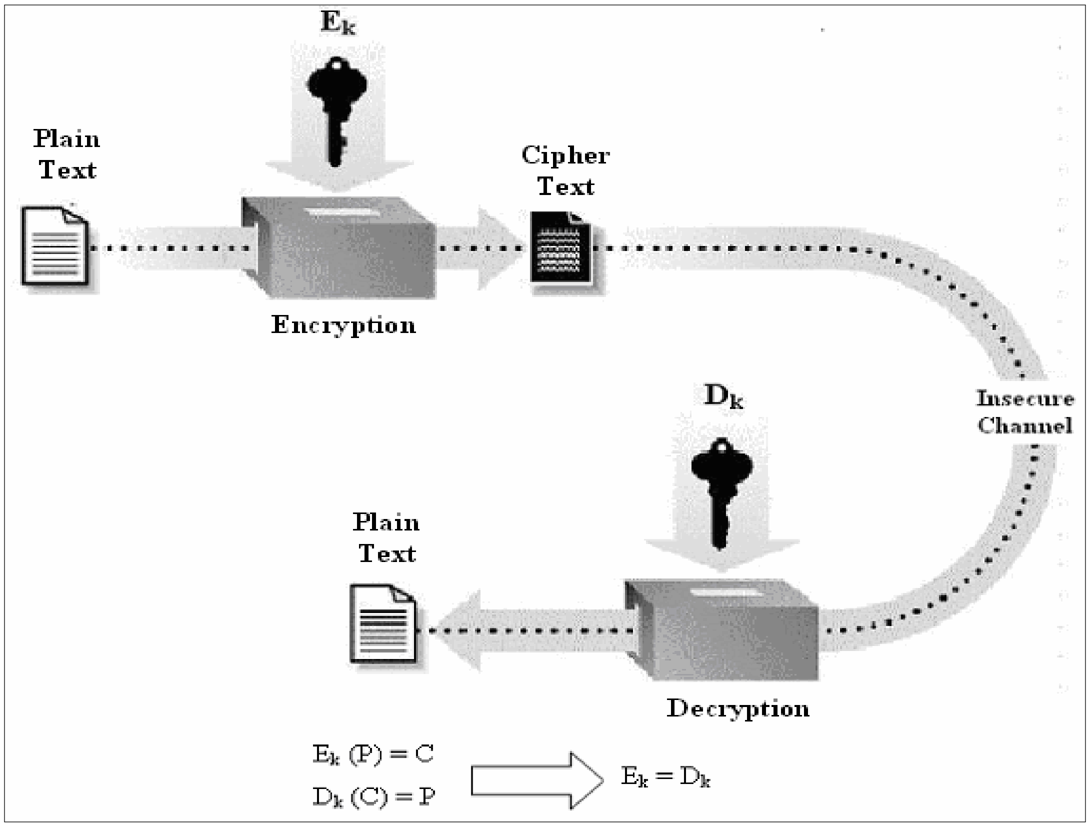

Figure 1.1. Symmetrical Cryptosystems

The security of a symmetric cryptosystem is a function of two parameters: the length of the key and the strength of the algorithm (Schneier 1996). Therefore the selected key length must be large enough to resist brute-force attacks.

1.2. Asymmetrical Cryptosystems

Asymmetrical cryptosystems, also known as public key cryptosystems, uses two different but mathematically related keys, a private key and a public key, as shown in Figure 1.2. Only the owner knows private key, whereas his/her public key is available for everyone. Therefore anyone can encrypt data using the public key, but just the owner executes the decryption process which requires the private key. Computing the public key using the private key must be an easy mathematical process. But calculating the private key using the public key must be computationally expensive.

Sender encrypts the plain text P using the receiver's public key and generates the cipher text C. The cipher text is sent through an insecure channel. Receiver gets the cipher text C and decrypts it using his/her private key to obtain plain text P.

The most common cryptosystems using public key are:

- Diffie-Hellman Key Exchange Protocol, DHKE, invented by Whitfield Diffie and Martin Hellman in 1976 (Diffie and Hellman 1976).
- RSA, invented by Ronald Rivest, Adi Shamir, and Len Adleman in 1978 (Rivest et al. 1978).
- ElGamal algorithm, described by Taher Elgamal in 1984 is based on Diffie-Hellman key agreement.
- Elliptic curve cryptography (ECC) based on the algebraic structure of elliptic curves over finite fields, suggested by Neal Koblitz and Victor S. Miller in 1985.

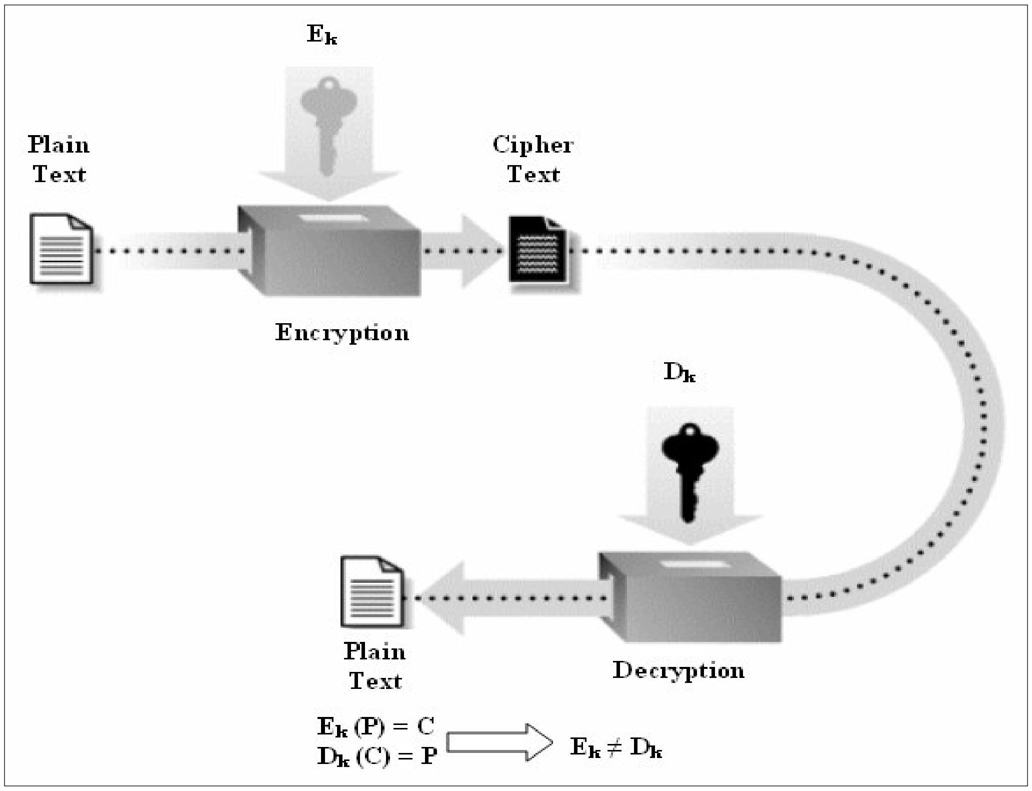

Figure 1.2. Asymmetrical Cryptosystems

1.3. Key Management

Design and implementation of a secure cryptosystem is not an easy procedure. It requires extensive mathematical and technical knowledge. But the security of the key (or key pair), which operates on the developed system, is more critical and severe as the security of the key (or key pair) indicates the security of the cryptosystem, either symmetrical or asymmetrical. Therefore, the key operations must be managed separately.

Key management copes with different operations on key, such as generating, destroying, storing, exchanging and publishing (Menezes et al. 1996).

- Generating, destroying and storing keys are common processes for both symmetrical and asymmetrical cryptosystems.
- Publishing (the public key) is an issue for only asymmetrical cryptosystems.
- Exchanging key is only required by symmetrical cryptosystems.

Publishing key does not cause a secrecy problem, because the published key is available for everyone. But it causes authentication problems. The publisher must be the person who he/she claims to be. On the other hand exchanging key causes both secrecy and authentication problems. The key must be calculable by two sides of the protocol but nobody else. And the same as the publisher, people who exchange keys must be the ones who they claim to be.

1.4. Key Exchange Problem for Symmetrical Cryptosystems

As mentioned earlier, symmetrical cryptosystems use the same secret key for encryption and decryption processes. Therefore the key must be distributed to each participant in the protocol. A secure method must be specified for key distribution because even if the message is encrypted using a hardly breakable algorithm, when the distribution method breaks down, encryption process makes no sense. In other words the security of a symmetrical cryptosystem depends on the security of the key.

Cryptanalytic attacks often work on key management as a result of this critical role of the key in symmetrical cryptosystems. Recovering the key from a storage mechanism or during a key exchange procedure is much easier than breaking a cryptographic algorithm. For this reason all cryptosystems, either symmetrical or asymmetrical, should have to recreate and distribute the key (or key pair) as frequent as possible. Not only the distribution frequency but also the number of the participants in a symmetrical protocol is a critical issue for key distribution. In symmetrical cryptosystems, if there are n participants in the protocol, the key exchange process is performed k times (where $k = (n.(n-1))/2$). When today’s network communication techniques is considered, it is possible to communicate hundreds of thousands of nodes simultaneously which requires billions of key exchange.

In early times of cryptography couriers are used as a secure method to distribute keys. But using a courier is not an effective choice today in the scope of the parameters; frequency and number of nodes. Couriers are not capable of distributing keys among such a large number of nodes as frequent as necessary.

Despite the fact that symmetrical cryptosystems are faster than asymmetrical cryptosystems, the problem of exchanging key makes them less popular.

1.5. The Proposed Solution

Distributing keys through the same communication channel (an insecure channel) as messages is more effective than using a courier. But the mentioned channel is not the secure one, which enforces us to encrypt our messages. Therefore the key must be exchanged through the channel using some cryptographic methods that provide security.

The key distribution problem of the symmetrical cryptosystems was firstly solved by Whitfield Diffie and Martin Hellman in 1976 using public key cryptography (Diffie and Hellman 1976). The solution method is called Diffie-Hellman Key Exchange Algorithm. It is based on a public-private key pair and their mathematical relation. The mathematical issue under Diffie-Hellman key exchange algorithm is the discrete logarithm problem.

After Diffie-Hellman had proposed using public key cryptosystems for exchanging keys, different key exchange protocols were developed similarly. Pretty Good Privacy (PGP) Encryption, which was originally created by Philip Zimmermann in 1991, is one of those protocols (Zimmermann 1995). The key is encrypted using an asymmetrical cryptosystem (using receiver’s public key). So only the receiver can decrypt the encrypted key. But using such a protocol requires not only another encryption-decryption process, but also the certification of the public key. This protocol is not in the scope of this thesis.

As newer cryptosystems had been developed, Diffie-Hellman Key Exchange Algorithm mutated over different mathematical problems. Elliptic Curve Diffie-Hellman and Elliptic-Curve MQV, which are based on elliptic curve cryptography, are some other algorithms for key exchange. These three algorithms are reviewed in the rest of this thesis.

CHAPTER 2

KEY EXCHANGE PROTOCOLS

As mentioned in Chapter 1, key exchange protocols are applied to solve the key distribution problem of symmetrical cryptosystems. The objective of a key exchange protocol is that, only the participants at the two ends of the protocol only have the possession of the secret key, but nobody else.

Similar to asymmetrical cryptosystems, key exchange protocols generate and use public-private key pairs. Also like all other cryptosystems they are based on mathematical problems. The mathematical background of these protocols makes it easy to compute a public key with given domain parameters and a private key. But computing the private key using the public key must be hard and infeasible.

Unlike the asymmetrical cryptosystems, key pairs are not used for encryption and decryption processes in key exchange protocols. They are used for the calculation of the shared key which is the secret key of the chosen symmetrical cryptosystem. The shared key must be calculable for any two key pairs which were generated using the same domain parameters. For this reason, designing a mathematical protocol for key exchange is harder than designing one for other public-private cryptosystems.

Shared key, or secret key, is also known as session key, because it is generally used for only one particular communication session. A session key does not exist at the end of the communication. The reason of generating a new key for each communication session is preventing cryptanalytic attacks. Because when more material that is encrypted with the same key is available, several attacks are made easily.

Even if each key exchange protocol has a different mathematical background, and consequently different parameters to generate keys, all of them have the same fundamental structure. This structure consists of domain parameters, public-private key pairs of both participants and a shared key. On top of this structure a generic algorithm is processed for each key exchange protocol, as illustrated in Figure 2.1.

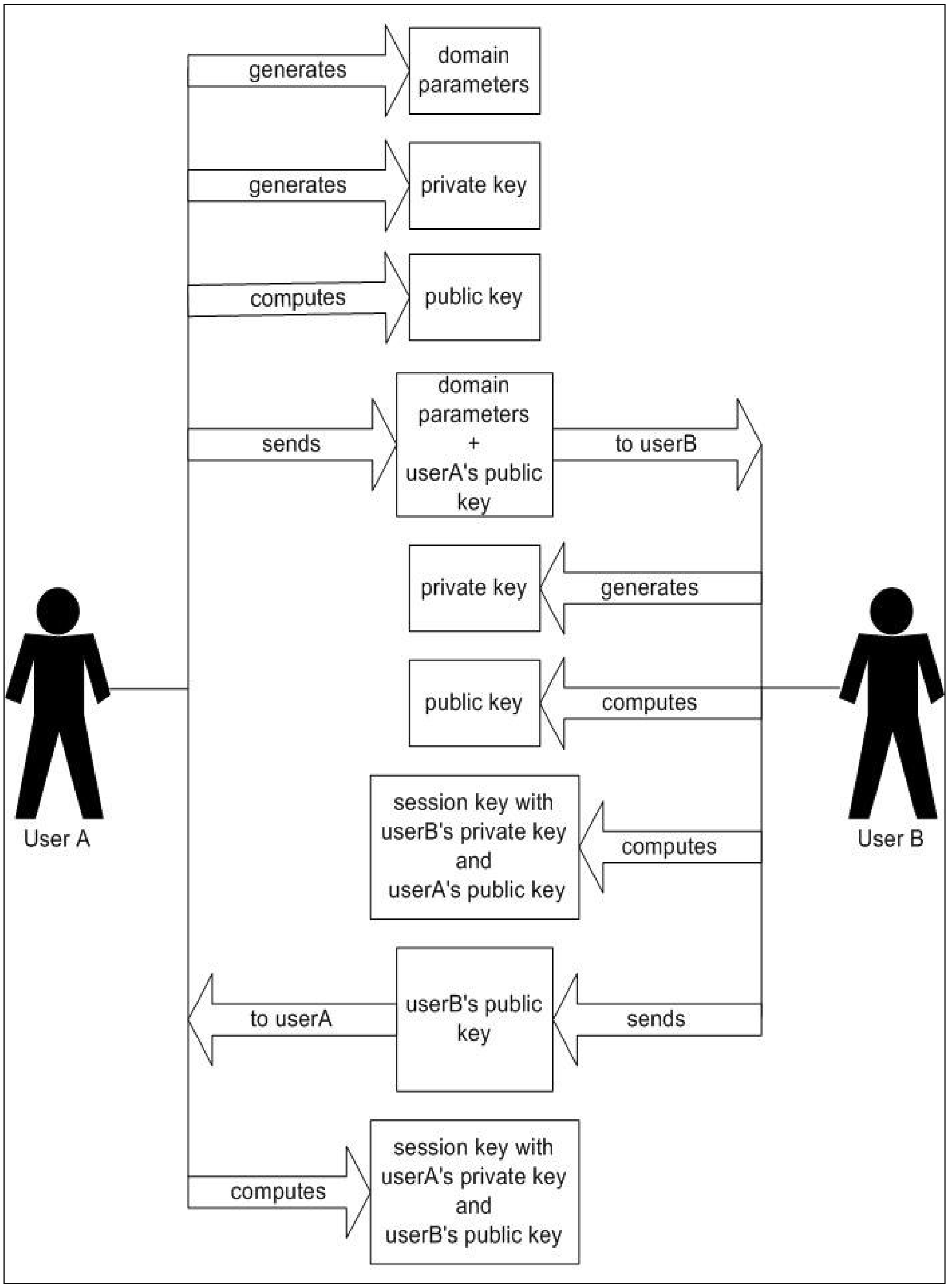

Figure 2.1. Fundamental Key Exchange Protocol

When users agree on starting to communicate and specify which key exchange protocol to use, the algorithm starts stepping as the following:

Step 1: User A generates domain parameters for specified key exchange protocol.

Step 2: User A generates a random private key using domain parameters.

Step 3: User A calculates public key using domain parameters and private key.

Step 4: User A sends domain parameters and its own public key to User B.

Step 5: User B generates a random private key using domain parameters.

Step 6: User B calculates public key using domain parameters and private key.

Step 7: User B calculates session key using its own private key and User A's public key.

Step 8: User B sends its own public key to User A.

Step 9: User A calculates session key using its own private key and User B's public key.

At the end of this algorithm, it is expected that both participants have the same session key. Otherwise, the encrypted material by one participant can not be decrypted by the other. This indicates that an error occurred during the key exchange procedure or an adversary attacked to the system.

There are numbers of protocols and their variants for exchanging keys. The mathematical improvements lead different types of protocols to be developed. Also additional solutions for the problems other than the key exchange, for example authentication, create variants of these protocols. Three of these protocols are explained in the rest of this chapter.

2.1. Diffie-Hellman Key Exchange

Diffie-Hellman Key Exchange Algorithm was developed by Whitfield Diffie and Martin Hellman in 1976 and published in “New Directions in Cryptography”. Actually it had been discovered by Malcolm J. Williamson within GCHQ[3] a few years earlier than Diffie-Hellman, but GCHQ didn’t make it public until 1997. Ralph Merkle’s work on public-key cryptography influenced the studies of Diffie and Hellman (Merkle 1978). Hence, Martin Hellman proposed calling the algorithm as Diffie-Hellman-Merkle Key Exchange Algorithm in 2002.

2.1.1. Mathematical Background

Diffie-Hellman Key Exchange Algorithm is based on discrete logarithm problem. In other words, security of the protocol depends on the difficulty of calculating discrete logarithms in finite fields.

Discrete logarithm can be considered as group theoretical version of ordinary logarithm. Ordinary logarithm is the inverse function of exponential function. Similarly, discrete logarithm is the inverse function of discrete exponential function, as shown in Figure 2.2.

[3] The Government Communications Headquarters (GCHQ) is a British intelligence agency responsible for providing signals intelligence (SIGINT) and information assurance to the UK government and armed forces as required, under the guidance of the Joint Intelligence Committee.

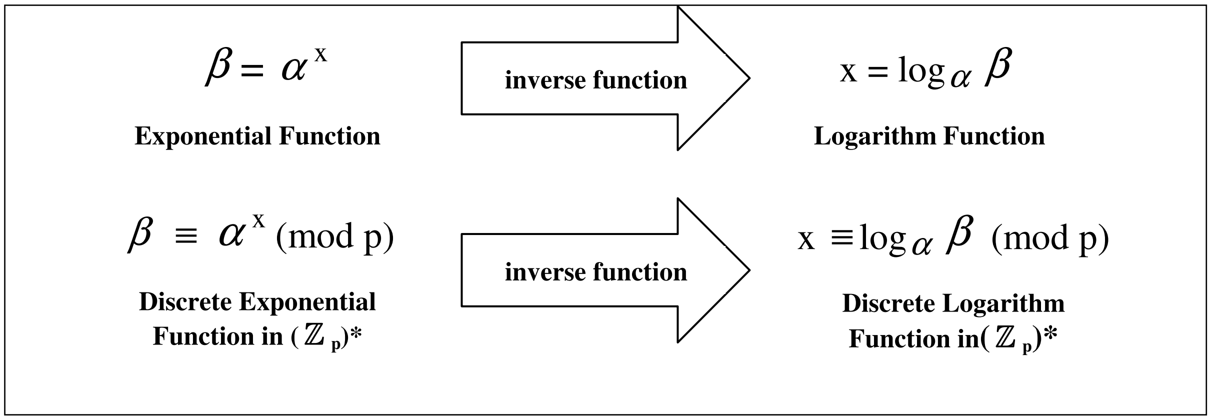
$\beta = \alpha^x$
Exponential Function
inverse function
$x = \log_\alpha \beta$
Logarithm Function
$\beta \equiv \alpha^x \pmod p$
Discrete Exponential Function in $(\mathbb{Z}_p)^*$
inverse function
$x \equiv \log_\alpha \beta \pmod p$
Discrete Logarithm Function in$(\mathbb{Z}_p)^*$

Figure 2.2. Discrete Logarithm

Using a formal definition discrete logarithm can be explained as:

"Let G is a finite cyclic group of order n. Let α be a generator of G, and let $\beta \in$ G. The discrete logarithm of β to the base α, which is denoted as $\log_\alpha \beta$, is the unique integer x, $0 \leq x \leq n-1$, such that $\beta = \alpha^x$ (Menezes et al.1996)."

Also discrete logarithm problem can be explained as:

"Given a prime p, a generator α of $\mathbb{Z}_p^*$ and an element $\beta \in \mathbb{Z}_p^*$, find the integer x, $0 \leq x \leq p-2$, such that $\alpha^x \equiv \beta \pmod p$ (Menezes et al. 1996)."

Calculation of discrete exponentiation is soluble in polynomial time. Firstly, x^{th} power of α is calculated then α^x is divided by the prime p and the remainder of the division is the result of the function. On the other hand, calculation of the discrete logarithm function is not soluble in polynomial time, which means it is an NP complete problem[4]. This attribute of the problem makes it a suitable candidate for use in the cryptographic protocols.

[4] NP (non-deterministic polynomial time) complete problems can not be computed in a polynomial time by a deterministic machine. For this reason these problems are insoluble.

2.1.2. Domain Parameters

The domain parameters include (ANSI X9.42):

- *p: A* prime defining the Galois Field[5] *GF(p)*, which is used as a modulus in the operations of *GF(p)*, where $2^{(L-1)} < p < 2^{L}$, for $L \geq 1024$, and L is a multiple of 256.
- *q: A* prime factor of p-1 such that $p = jq+1$ and $q > 2^{m-1}$. $GF(p)^*$ has a cyclic subgroup of order q.
- *g: A* generator of the q-order cyclic subgroup of $GF(p)^*$, that is, an element of order q in the multiplicative group of *GF(p)*.

2.1.3. Keys

The key pair includes:

- x_A*: A* private key which is selected as $1 \leq x \leq (q-1)$.
- y_A*: A* public key which is calculated as $y_A = g^{x_A} \pmod{p}$.

The shared key is calculated as:

$$K = (y_B)^{x_A} (\text{mod}\, p) = (g^{x_B})^{x_A} (\text{mod}\, p) = g^{x_A . x_B} (\text{mod}\, p) \qquad (2.1)$$

$$K = (y_A)^{x_B} (\text{mod}\, p) = (g^{x_A})^{x_B} (\text{mod}\, p) = g^{x_A . x_B} (\text{mod}\, p) \qquad (2.2)$$

[5] In abstract algebra, a finite field or Galois field (so named in honor of Évariste Galois) is a field that contains only finitely many elements. Finite fields are important in number theory, algebraic geometry, Galois theory, cryptography, and coding theory.

2.1.4. Algorithm

Diffie-Hellman Key Exchange Algorithm steps as the following:

Step 1: User A generates domain parameters p, q and g.

Step 2: User A generates a random private key x_A.

Step 3: User A calculates public key as $y_A = g^{x_A} \pmod{p}$.

Step 4: User A sends (p, g, y_A) to user B.

Step 5: User B generates a random private key x_B.

Step 6: User B calculates public key $y_B = g^{x_B} \pmod{p}$.

Step 7: User B calculates session key as

$$K = (y_A)^{x_B} (\text{mod } p) = (g^{x_A})^{x_B} (\text{mod } p) = g^{x_A \cdot x_B} (\text{mod } p)$$

Step 8: User B sends y_B to user A.

Step 9: User A calculates session key as

$$K = (y_B)^{x_A} (\text{mod } p) = (g^{x_B})^{x_A} (\text{mod } p) = g^{x_A \cdot x_B} (\text{mod } p)$$

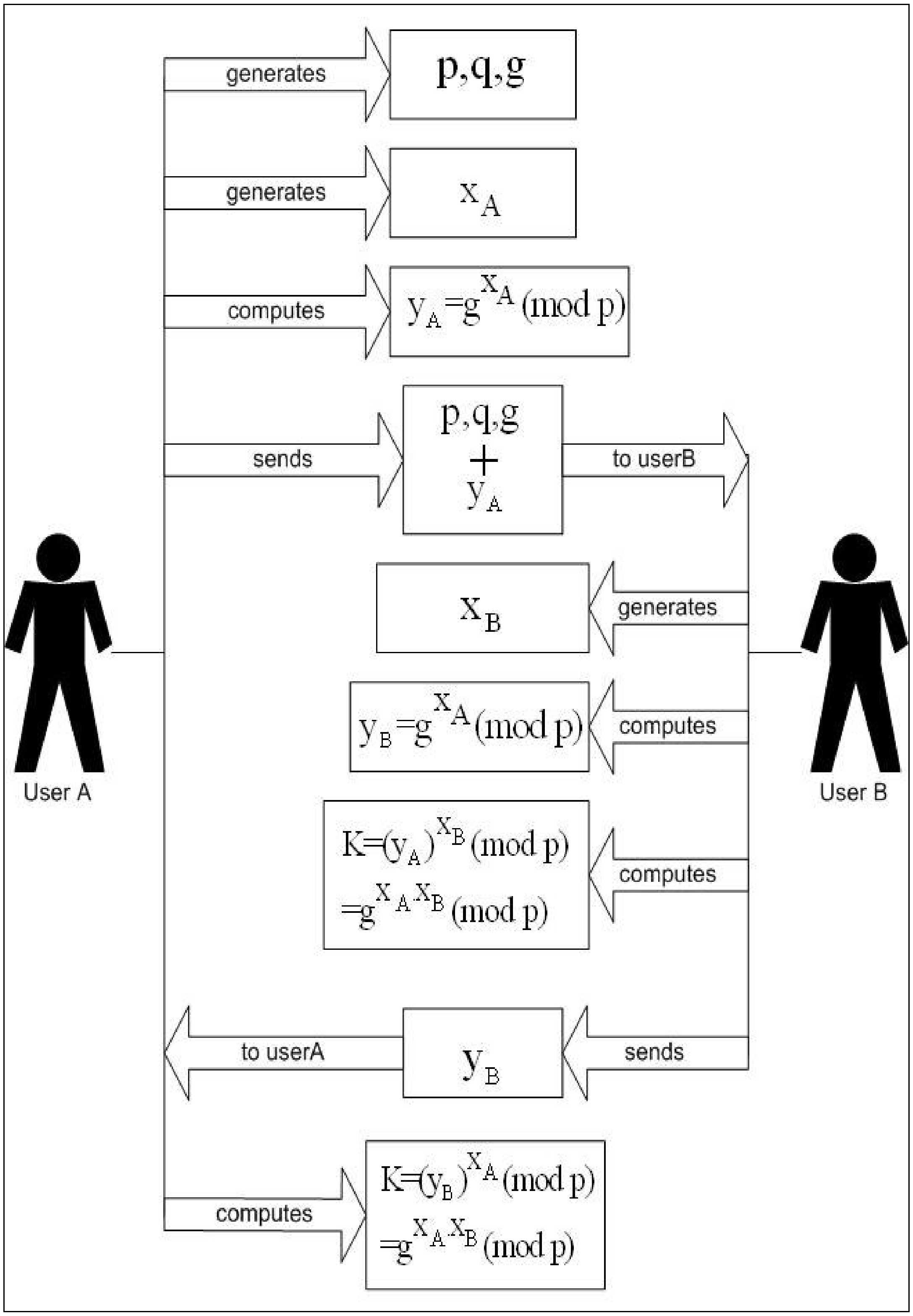

Figure 2.3. Diffie-Hellman Key Exchange Protocol

2.2. Elliptic Curve Diffie-Hellman

Elliptic Curve Diffie-Hellman Key Exchange Algorithm which uses elliptic curve cryptography is a variant of Diffie-Hellman Key Exchange Algorithm. Elliptic curve cryptography which is denoted as ECC is a new approach to public-key cryptography (Hankerson et al. 2004). Using elliptic curves in cryptography was suggested by two professors of mathematics, Neal Koblitz and Victor S. Miller, in 1985 separately. ECC has been applied in different schemes of cryptology, such as;

- Elliptic Curve Diffie-Hellman Key Exchange Algorithm (ECDH).
- Elliptic Curve MQV (ECMQV), for key agreement.
- Elliptic Curve Digital Signature Algorithm (ECDSA).

The characteristics of elliptic curves, such as being defined on a finite cyclic group, and having operations as addition and doubling for the points on the elliptic curve makes these curves suitable for cryptographic protocols. Because when a scalar multiplication, which is actually a point addition, of an integer number (n) and a point (P) on curve is performed, even if the start and stop points and all other parameters about curve are known, it is hard to find the integer n.

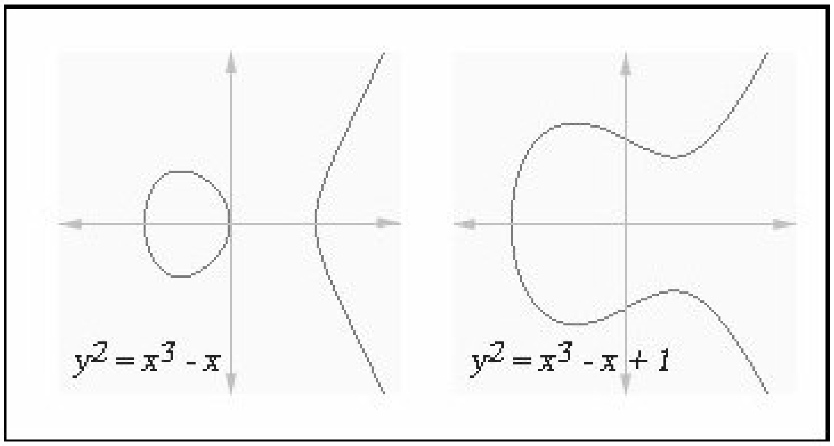

Figure 2.4. Examples for Elliptic Curves
(Source: Atay, 2006)

2.2.1. Mathematical Background

Elliptic Curves: An elliptic curve over a finite field $\mathbb{F}$ is defined by the equation,

$$E: y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{2.3}$$

This equation is known as Weierstrass Equation (Hankerson 2004). Actually, elliptic curves are defined over three types of finite fields and Weierstrass Equation is simplified differently due to these types of field $\mathbb{F}_q$.

- If the chosen field has a odd characteristic, which is denoted as $\mathbb{F}_p$ ($p>3$ is a large prime) the equation is simplified as,

  $$E: y^2 = x^3 + ax + b \tag{2.4}$$

  where a, b $\in \mathbb{F}_p$ and the discriminant of the curve is $\Delta = 16(4a^3 + 27b^2)$.

- If the chosen field has a characteristic of two, which is denoted as $\mathbb{F}_{2^m}$ the equation is simplified as,

  $$E: y^2 + xy = x^3 + ax^2 + b \tag{2.5}$$

  where a, b $\in \mathbb{F}_{2^m}$ and the discriminant of the curve is $\Delta = b$. Such curves are called *non-supersingular*. If in the equation 2.3 a1 = 0, then equation is simplified as,

  $$E: y^2 + cy = x^3 + ax + b \tag{2.6}$$

  where a, b, c $\in \mathbb{F}_{2^m}$ and the discriminant of the curve is $\Delta = c^4$. Such curves are called *supersingular.*

- If the chosen field has a characteristic of three, which is denoted as $\mathbb{F}_3$ the equation is simplified as,

  $$E: y^2 = x^3 + ax^2 + b \tag{2.7}$$

  where a, b $\in \mathbb{F}_3$ and the discriminant of the curve is $\Delta = -a^3b$. Such curves are called *non-supersingular*. If in equation 2.3 $a_1^2 = -a_2$, then equation is simplified as,

$$E : y^2 = x^3 + ax + b \qquad (2.8)$$

where a, b $\in$ $\mathbb{F}_3$ and the discriminant of the curve is $\Delta = -a^3$. Such curves are called *supersingular.*

The chosen elliptic curve must satisfy the rules of being a finite cyclic group. This necessity leads some rules and operations to occur.

Chord-Tangent Rule: A line which passes through the two different points on an elliptic curve must intercept the same curve at a third point.

Point at Infinity: An identity element is necessary for elliptic curve group operations. Actually Weierstrass Equation, equation 2.3, is defined over a three dimensional field such as $\mathbb{F}^3[x : y : z]$. But in order to obtain the identity element, it is assumed that z coordinates of the field is 0 and the point (0, 1, 0) is chosen as the identity element. This point is called as point at infinity and denoted as ∞. The rules that requires identity element can be defined for the points P, Q $\in$ E($\mathbb{F}_q$) as such;

- if $Q = \infty$ then $P + \infty = P$
- if $Q = -P$ then $P + Q = \infty$

Point Multiplication: Point multiplication is required for generating key pairs, decryption and encryption processes. This operation is known as scalar multiplication, because it is multiplication of an integer n by a point P. Scalar multiplication can be considered as an n times point addition, such as,

$$n.P = \underbrace{P + P + ...+ P}_{n}$$

Point Addition: In order to add two points, P and Q, on an elliptic curve the chord-tangent rule is applied. If P and Q are not equal, then the line which passes through the points intercepts the curve at a third point –R, as shown in Figure 2.5. If P and Q are equal, then a tangent line to the elliptic curve at point P (or Q) intercepts the curve at a second point. The result of addition is the symmetrical point of the -R (the third interception point in chord-tangent rule) according to x-axis. Addition operation can be denoted as,

$$P + Q = -(P.Q)$$

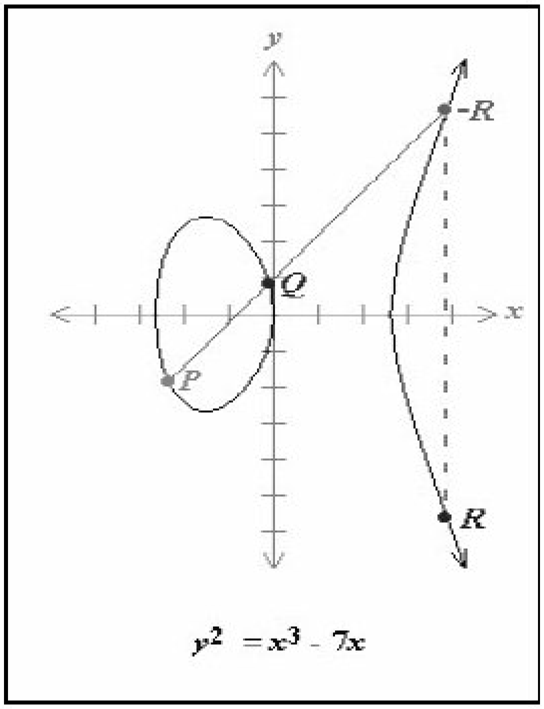

Figure 2.5. Point Addition on an Elliptic Curve
(Source: Atay, 2006)

2.2.2. Domain Parameters

The domain parameters include (ANSI X9.63):

- *p: A* prime defining order of the finite field $\mathbb{F}_q$, where q = p and p > 3
- *a, b:* Two field elements in $\mathbb{F}_p$ which define the equation of the elliptic curve

$$E : y^2 = x^3 + ax + b (\text{mod } p)$$

- x_G, y_G*:* Two field elements in $\mathbb{F}_p$ which define a point $G(x_G, y_G)$ of prime order on E (note that E $\neq$ ∞)
- *n:* The order of the point G
- *h:* The cofactor defined as $h = \# E(Fp) / n$

2.2.3. Keys

The key pair includes:

- x_A: A private key which is selected in the interval [1, n-1].
- Q_A: A public key which is a point $Q(x_Q, y_Q)$ on elliptic curve and calculated using point multiplication $Q = x_A.G$

The shared key is calculated as:

$$K = x_A.Q_B = x_A.x_B.G \text{ by User A} \tag{2.9}$$

$$K = x_B.Q_A = x_A.x_B.G \text{ by User B} \tag{2.10}$$

2.2.4. Algorithm

Elliptic Curve Diffie-Hellman Key Exchange Algorithm that is shown in Figure 2.6 steps as the following:

Step 1: User A generates domain parameters p, a, b, x_G , y_G , n and h.

Step 2: User A generates a random private key x_A .

Step 3: User A calculates public key as $Q_A = x_A.G$.

Step 4: User A sends (p, a, b, x_G **,** y_G **,** n, h, Q_A) to User B.

Step 5: User B generates a random private key x_B .

Step 6: User B calculates public key as $Q_B = x_B.G$.

Step 7: User B calculates session key $K = x_B.Q_A = x_B.x_A.G$.

Step 8: User B sends Q_B to User A.

Step 9: User A calculates session key $K = x_A.Q_B = x_A.x_B.G$

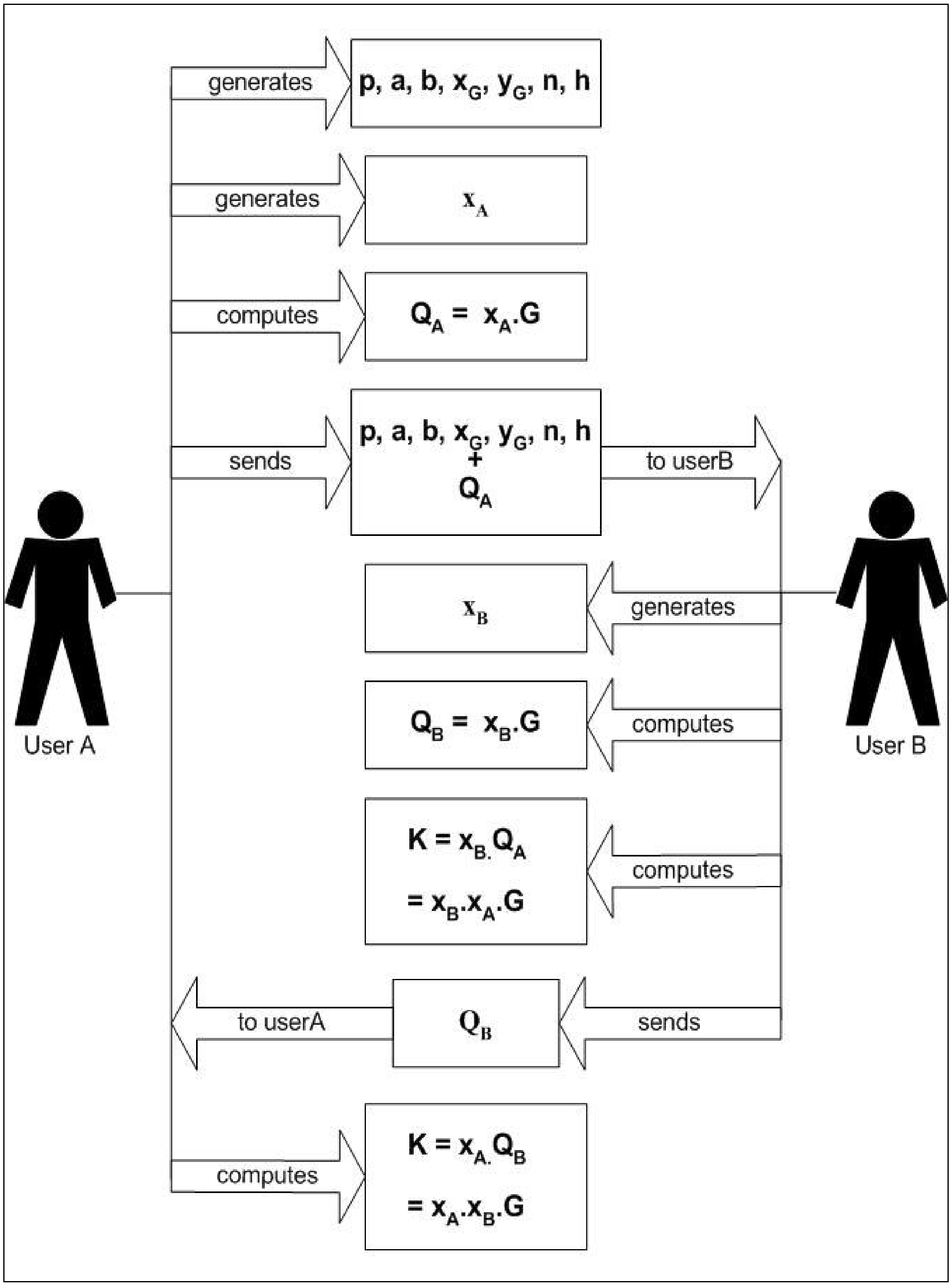

Figure 2.6. Elliptic Curve Diffie-Hellman Key Exchange Protocol

2.3. Elliptic Curve MQV

MQV which stands for Menezes-Qu-Vanstone is based on the Diffie-Hellman scheme. It was proposed by Alfred Menezes, Minghua Qu and Scott Vanstone in 1995 (Hankerson et al. 2004). It is an authenticated key agreement protocol which is developed for protection against active attacks[6], such as man-in-the-middle attack.

The fundamental distinction between MQV and other key exchange protocols is that MQV uses two different key pairs. One of these key pairs is called static or long-term key pair which is bound to the entity for a certain period of time, typically through the use of certificates. The other key pair is called ephemeral or short-term key pair which is generated for each run of the protocol (Law et al. 1998).

MQV protocols vary according to the finite group on which the protocol works. Elliptic Curve MQV which works on elliptic curve groups is one of these variants and it is denoted as ECMQV. There are also different forms of MQV algorithms, which can be applied to any variant. These forms can be summarized as fallows:

- *One Pass Authenticated Key Agreement Protocol* can be considered as store and forward form. Because it does not require interactive communication. It is used when a party sends an encrypted message and the other party decrypts the message any time, but does not respond (i.e., can be applied while sending e-mail). Hence, just the initiator generates an ephemeral key pair and passes the public key to the receiver. Because the transmission operation through the channel is performed only once, the protocol is called One Pass Authenticated Key Agreement Protocol.

- *Two Pass Authenticated Key Agreement Protocol* can be considered as an interactive form. Because, the each party in the protocol sends and receives encrypted messages and each performs decryption processes simultaneously (i.e., can be applied during instant messaging). For this reason, both initiator and receiver have to generate an ephemeral key pair and send the public key to the other.

[6] An active attack is one in which the attacker can modify or delete transmitting messages, or transmit new messages.

Because the transmission operation is performed twice, one for each party, the protocol is called Two Pass Authenticated Key Agreement Protocol.

- *Three Pass Authenticated Key Agreement Protocol* is also known as Authenticated Key Agreement with Key Confirmation. It provides not only secrecy for the secret key but also assurance that each party in the protocol has the possession of the same secret key. The protocol uses a message authentication code (MAC) algorithm for the key confirmation. MAC is a key dependent one-way hash function (Schneier 1996). Each party applies the MAC algorithm using domain parameters and obtains a secret key. Initially the initiator generates and sends its ephemeral public key, and then the receiver generates and sends its ephemeral public key and computes a MAC result. Finally the initiator sends the computed MAC result. Because the transmission operation is performed three times, the protocol is called Three Pass Authenticated Key Agreement Protocol. This protocol also requires a hash algorithm as a key derivation function (KDF) to derive one or more keys from the secret key. When such a derivation function is applied, even if attackers obtain the derived key, they can not learn any useful information about either the input secret value or any of the other derived keys. Also deriving keys eliminates the weak keys. Typically hash functions, such as SHA-1, are used as key derivation functions.

2.3.1. Mathematical Background

Similar to ECDH protocol, ECMQV is also based on elliptic curve arithmetic. For this reason, the operations and structures that are defined in section 2.2.1 are also in use for ECMQV. On the other hand ECMQV requires an additional calculation to derive a point on a specified elliptic curve, such as:

"f denotes the bit length of n, which is the prime order of the base point P; i.e $f = \lfloor \log_2 n \rfloor + 1$. If Q is point over the elliptic curve, then $\overline{Q}$ is defined as follows. Let x be the x-coordinate of Q, and let $\overline{x}$ be the integer obtained from binary representation of x. (The value of $\overline{x}$ will depend on the representation chosen for the elements of the field $\mathbb{F}_q$.)

Then $\overline{Q}$ is defined to be the integer $\left(\bar{x} \bmod 2^{\lceil f/2 \rceil}\right) + 2^{\lceil f/2 \rceil}$. Observe that $(\overline{Q} \bmod n) \neq \infty$ (Law et al. 1998)."

2.3.2. Domain Parameters

The domain parameters include (ANSI X9.63):

- *p: A* prime defining the order of the finite field $\mathbb{F}_p$, where p > 3
- *a, b:* Two field elements in $\mathbb{F}_p$ which define the equation constants of the elliptic curve

$$E : y^2 = x^3 + ax + b (\text{mod } p)$$

- x_G, y_G**:** Two field elements in $\mathbb{F}_p$ which define a point $G(x_G, y_G)$ of prime order on E (note the E $\neq$ ∞)
- *n:* The order of the points over E in $\mathbb{F}_p$; $\# E(F_p) = n$
- *h:* The cofactor defined as $h = \# E(Fp) / n$

These parameters are specified by assuming that the elliptic curve E is defined on a finite field that has odd characteristic ($\mathbb{F}_p$).

2.3.3. Keys

The static (or long-term) key pair includes:

- w_A : A static private key.
- W_A: A static public key.

It is assumed that static key pairs are exchanged via public key certificates. These certificates provide storage, security and authenticity for public keys and they are verified by a certification authority (CA). Certification authority is a trusted third party who vouches for the authenticity of the public key. VeriSign (WEB_1 2007), GeoTrust (WEB_2 2007) and Comondo (WEB_3 2007) are the first three companies in the certification authority business. The data partition of the certificate includes not only the static public key but also the domain parameters. (Menezes et al. 1996)

The ephemeral (or short-term) key pair includes:

- r_A: An ephemeral private key which is selected as a statistically unique and unpredictable (random) integer in the interval [1, n-1].
- R_A: An ephemeral public key which is a point $R(x_R, y_R)$ on elliptic curve and calculated using point multiplication $R_A = r_A.G$

Static key pairs are shared over a certification authority, so they provide authentication. An i*mplicit signature* is calculated in order to append the static key pair to the shared key calculation. The implicit signature is symbolized as s_A (or s_B) and calculated as:

$$s_A = (r_A + \overline{R}_A.w_A) \bmod n \quad \text{by User A} \tag{2.11}$$

$$s_B = (r_B + \overline{R}_B.w_B) \bmod n \quad \text{by User B} \tag{2.12}$$

Then the shared key is calculated as:

$$\begin{aligned} K &= h.s_A.(R_B + \overline{R}_B.W_B) \\ &= h.((r_A + \overline{R}_A.w_A) \bmod n).(R_B + \overline{R}_B.W_B) \\ &= h.(((r_A + \overline{R}_A.w_A) \bmod n).R_B + ((r_A + \overline{R}_A.w_A) \bmod n).\overline{R}_B.W_B) \\ &= h.(r_A.R_B + \overline{R}_A.w_A.R_B + r_A.\overline{R}_B.W_B + \overline{R}_A.w_A.\overline{R}_B.W_B) \\ &= h.(r_A.(r_B.G) + \overline{R}_A.w_A.(r_B.G) + r_A.\overline{R}_B.(w_B.G) + \overline{R}_A.w_A.\overline{R}_B.(w_B.G)) \\ &= h.G.(r_A.r_B + r_B.w_A.\overline{R}_A + r_A.w_B.\overline{R}_B + w_A.w_B.\overline{R}_A.\overline{R}_B) \qquad \text{by User A} \end{aligned} \tag{2.13}$$

$$\begin{aligned} K &= h.s_B.(R_A + \overline{R}_A.W_A) \\ &= h.((r_B + \overline{R}_B.w_B) \bmod n).(R_A + \overline{R}_A.W_A) \\ &= h.(((r_B + \overline{R}_B.w_B) \bmod n).R_A + ((r_B + \overline{R}_B.w_B) \bmod n).\overline{R}_A.W_A) \\ &= h.(r_B.R_A + \overline{R}_B.w_B.R_A + r_B.\overline{R}_A.W_A + \overline{R}_B.w_B.\overline{R}_A.W_A) \\ &= h.(r_B.(r_A.G) + \overline{R}_B.w_B.(r_A.G) + r_B.\overline{R}_A.(w_A.G) + \overline{R}_B.w_B.\overline{R}_A.(w_A.G)) \\ &= h.G.(r_A.r_B + r_A.w_B.\overline{R}_B + r_B.w_A.\overline{R}_A + w_A.w_B.\overline{R}_A.\overline{R}_B) \qquad \text{by User B} \end{aligned} \tag{2.14}$$

2.3.4. Algorithm

Domain parameters and static keys have been already generated and shared through a certification authority when the communication is started. So these operations are not included in the following Key Exchange Algorithms.

One Pass Authenticated Key Agreement Algorithm: Algorithm that is shown in Figure 2.7 steps as the following:

Step 1: User A generates a random private key r_A, computes public key as $R_A = r_A.G$ and sends R_A to User B.

Step 2: User A computes the implicit signature $s_A = (r_A + \overline{R}_A.w_A) \bmod n$ and the shared key $K = h.s_A.(W_B + \overline{W}_B.W_B)$. If K is equal point at infinity ($K = \infty$) then User A terminates the protocol run with failure.

Step 3: User B does a key validation of R_A. If validation fails then User B terminates the protocol run with failure. Otherwise User B computes the implicit signature $s_B = (w_B + \overline{W}_B.w_B) \bmod n$ and the shared key $K = h.s_B.(R_A + \overline{R}_A.W_A)$. If K is equal point at infinity ($K = \infty$) then User B terminates the protocol run with failure.

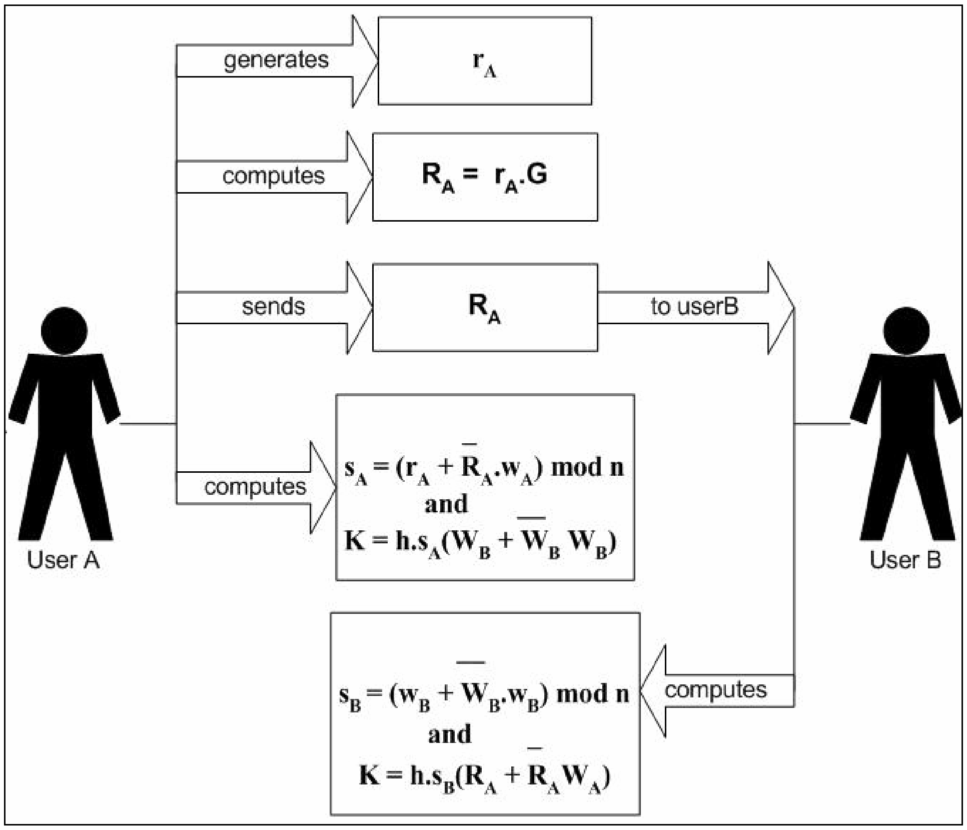
generates
r_A
computes
$R_A = r_A.G$
sends
R_A
to userB
computes
$s_A = (r_A + \overline{R}_A.w_A) \bmod n$ and $K = h.s_A(W_B + \overline{W}_B W_B)$
User A
User B
$s_B = (w_B + \overline{W}_B.w_B) \bmod n$ and $K = h.s_B(R_A + \overline{R}_A W_A)$
computes

Figure 2.7. One Pass Authenticated Key Agreement Algorithm

Two Pass Authenticated Key Agreement Algorithm: Algorithm that is shown in Figure 2.8 steps as the following:

Step 1: User A generates a random private key r_A, computes public key as $R_A = r_A.G$ and sends R_A to User B.

Step 2: User B generates a random private key r_B, computes public key as $R_B = r_B.G$ and sends R_B to User A.

Step 3: User A does a key validation of R_B. If validation fails then User A terminates the protocol run with failure. Otherwise User A computes the implicit signature $s_A = (r_A + \overline{R}_A.w_A) \bmod n$ and the shared key $K = h.s_A.(R_B + \overline{R}_B.W_B)$. If K is equal point at infinity ($K = \infty$) then User A terminates the protocol run with failure.

Step 4: User B does a key validation of R_A. If validation fails then User B terminates the protocol run with failure. Otherwise User B computes the implicit signature $s_B = (r_B + \overline{R}_B.w_B) \bmod n$ and the shared key $K = h.s_B.(R_A + \overline{R}_A.W_A)$. If K is equal point at infinity ($K = \infty$) then User A terminates the protocol run with failure.

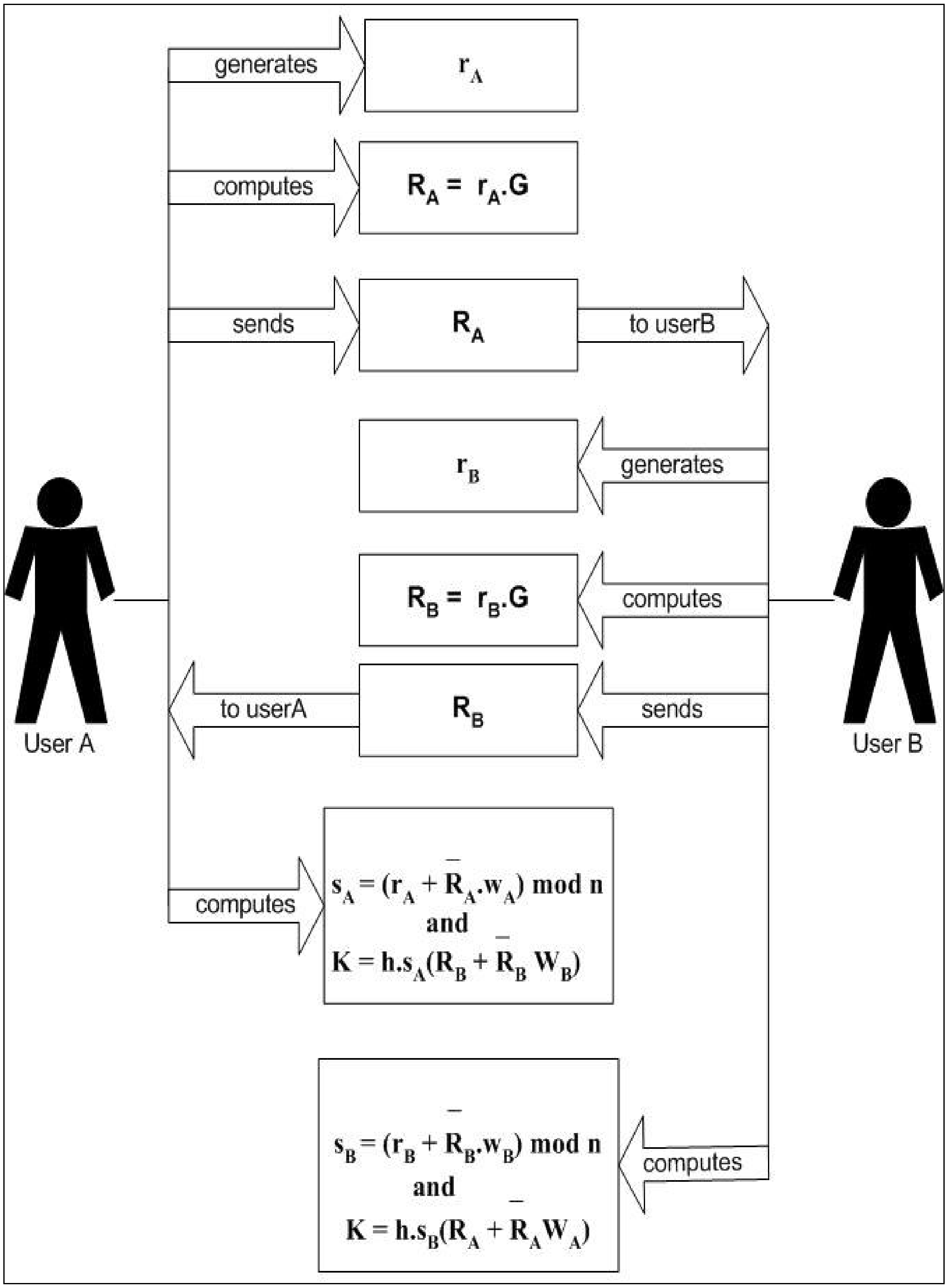

Figure 2.8. Two Pass Authenticated Key Agreement Algorithm

Three Pass Authenticated Key Agreement Algorithm: Algorithm that is shown in Figure 2.9 steps as the following:

Step 1: User A generates a random private key r_A, computes public key as $R_A = r_A.G$ and sends R_A to User B.

Step 2: User B does a key validation of R_A. If validation fails then User B terminates the protocol run with failure. Otherwise User B generates a random private key r_B, computes public key as $R_B = r_B.G$.

Step 3: User B computes the implicit signature $s_B = (r_B + \overline{R}_B.w_B) \bmod n$ and the shared key $K = h.s_B.(R_A + \overline{R}_A.W_A)$. If K is equal point at infinity ($K = \infty$) then User B terminates the protocol run with failure.

Step 4: User B derives the key K and computes a $\mathrm{MAC_K(B, A, R_B, R_A)}$ value for the derived key. Then send the MAC value and R_B to User A.

Step 5: User A does a key validation of R_B. If validation fails then User A terminates the protocol run with failure. Otherwise User A computes the implicit signature $s_A = (r_A + \overline{R}_A.w_A) \bmod n$ and the shared key $K = h.s_A.(R_B + \overline{R}_B.W_B)$. If K is equal point at infinity ($K = \infty$) then User A terminates the protocol run with failure.

Step 6: User A derives the key K and computes a $\mathrm{MAC_K(B, A, R_B, R_A)}$ value for the derived key and verifies that the computed value equals to the value that was sent by User B.

Step 7: User A computes a $\mathrm{MAC_K(A, B, R_A, R_B)}$ value for the derived key. Then send the MAC value to User B.

Step 8: User B computes a $\mathrm{MAC_K(A, B, R_A, R_B)}$ value for the derived key and verifies that the computed value equals to the value that was sent by User A.

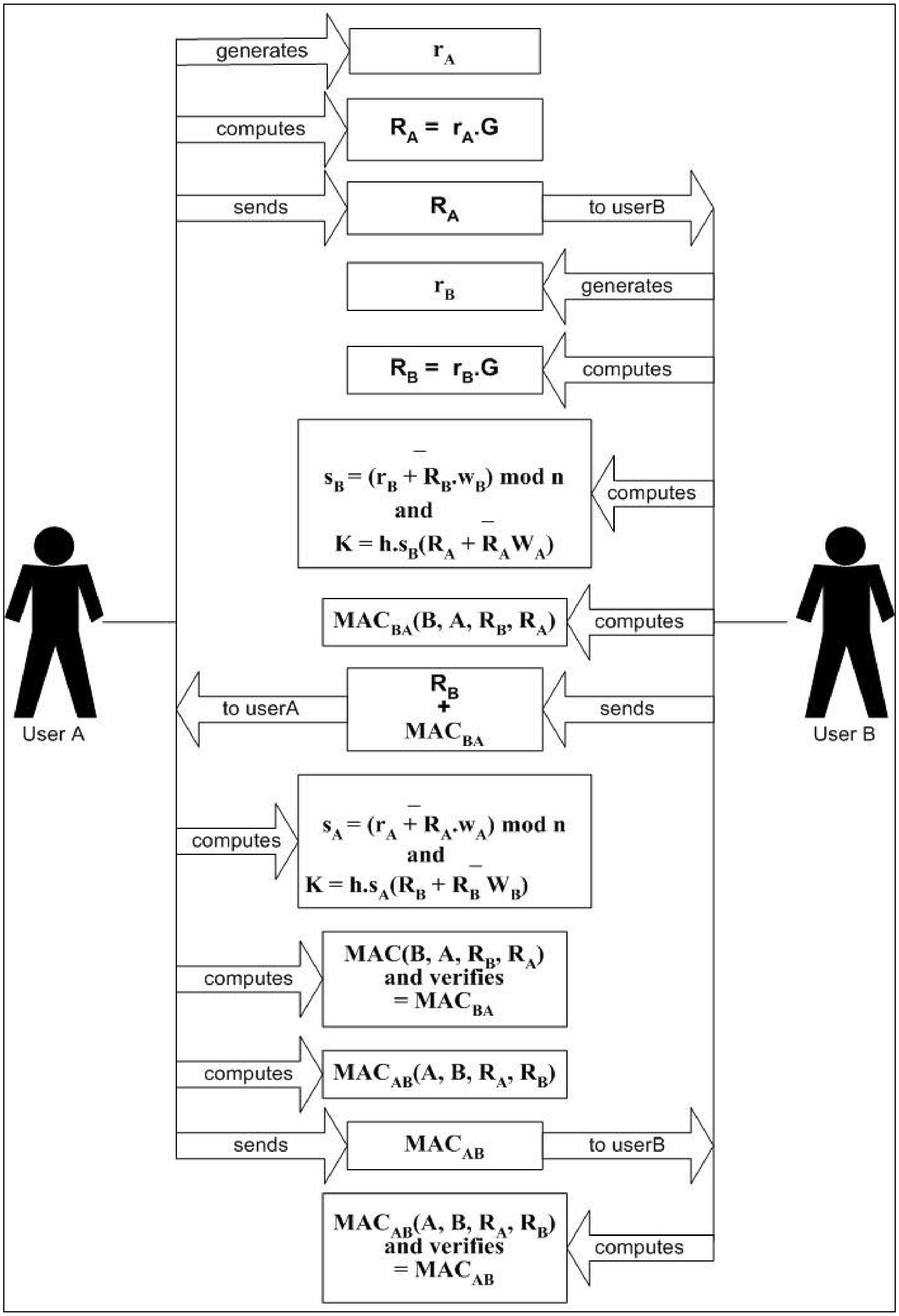
generates
r_A
computes
$R_A = r_A.G$
sends
R_A
to userB
r_B
generates
$R_B = r_B.G$
computes
$s_B = (r_B + \bar{R}_B.w_B) \bmod n$ and $K = h.s_B(R_A + \bar{R}_A W_A)$
computes
$MAC_{BA}(B, A, R_B, R_A)$
computes
User A
User B
to userA
R_B + MAC_{BA}
sends
computes
$s_A = (r_A + \bar{R}_A.w_A) \bmod n$ and $K = h.s_A(R_B + \bar{R}_B W_B)$
computes
$MAC(B, A, R_B, R_A)$ and verifies $= MAC_{BA}$
computes
$MAC_{AB}(A, B, R_A, R_B)$
sends
MAC_{AB}
to userB
$MAC_{AB}(A, B, R_A, R_B)$ and verifies $= MAC_{AB}$
computes

Figure 2.9. Three Pass Authenticated Key Agreement Algorithm

2.4. Key Length of Key Exchange Protocols

As mentioned in section 1.4 cryptanalytic attacks often work on the keys. For this reason key must be recreated as much as possible. Hence, adversaries can not have much data which is encrypted using the same key to analyze. But it is still possible to break the cryptographic algorithm using any sniffed data. Therefore some parameters must be specified to make key more resistant against cryptanalytic attacks. The bit length of the key is the one of these parameters. It is easier to break a cryptosystem which uses smaller key lengths. As the key length rises up, the system becomes more resistant against attacks.

NIST[7] describes some standards to identify security levels for key lengths of each cryptographic protocol. There are several publications including FIPS[8] and special publications 800 series. Special publication 800-57 Recommendation for Key Management includes the key lengths of key exchange protocols with corresponding security levels, Table 2.1 (Barker et al. 2007).

Table 2.1. Key Lengths with Corresponding Security Levels

		FFC[9]	
Bits of security	**Symmetric Key Algorithms**	**DLP[10] (DHKE)** (N:bit length of private key L: bit length of public key)	**ECC[11](ECDHKE, ECMQV)** (f: bit length of the order of base point G)
80	2TDEA[12]	L = 1024, N = 160	f = 160 - 223
112	3 TDEA[13]	L =2048, N = 224	f = 224 - 255
128	AES-128	L = 3072, N =256	f = 256 – 383
192	AES-192	L = 7680, N =384	f = 384 – 511
256	AES-256	L = 15360, N =512	f = 512 +

7 National Institute of Standards and Technology, http://www.nist.gov/
8 Federal Information Processing Standarts, http://www.itl.nist.gov/fipspubs/
9 Finite Field Cryptography
10 Discrete Logarithm Problem
11 Elliptic Curve Cryptography
12 Two Key Triple Data Encryption Algorithm: TDEA uses three 56-bit keys; K_1, K_2 and K_3. If $K_1 = K_3 \neq K_2$ then TDEA is said as Two Key Triple Data Encryption Algorithm.
13 Three Key Triple Data Encryption Algorithm: TDEA uses three 56-bit keys; K_1, K_2 and K_3. If $K_1 \neq K_2 \neq K_3$ then TDEA is said Three Key Triple Data Encryption Algorithm.

The first column indicates the security strength of a cryptographic algorithm. “Bits of security” is a number that associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm. The security strength of an algorithm for a given key size is traditionally described in terms of the amount of work it takes to try all keys for a symmetric algorithm that has no short cut attacks (i.e., the most efficient attack is to try all possible keys). In this case, the best attack is said to be the exhaustion attack. An algorithm that has a "*Y*" bit key, but whose strength is comparable to an "*X*" bit key of such a symmetric algorithm is said have a “security strength of *X* bits” or to provide “*X* bits of security” (Barker et al. 2007). The larger “bits of security” indicates the higher key lengths and the more difficultly broken algorithm.

The second column identifies the symmetric key algorithms that provide the indicated level of security (at a minimum) (Barker et al. 2007).

The third column identifies minimum size of the parameters (L and N) for algorithms (e.g., Digital Signature Algorithm (DSA), Diffie-Hellman Key Exchange Algorithm (DHKE)) which are based on discrete logarithm problem (DLP). L indicates the length of the public key and N indicates the length of the private key.

The fourth column identifies the range of f (the bit size of n, where n is the order of the base point G) for algorithms (Elliptic Curve Digital Signature Algorithm (ECDSA), Elliptic Curve Diffie-Hellman Key Exchange Algorithm (ECDHKE)) which are based on elliptic curve discrete logarithm problem (ECDLP).

These key lengths are used to test implemented key exchange protocols in CRYMPIX. Execution times of key generation functions for different key lengths are placed in Chapter 3. Hence it is possible to compare effectiveness of the protocols with respect to execution times.

CHAPTER 3

PROTOCOL IMPLEMENTATION

3.1. A Cryptographic Library CRYMPIX

Computers which have fixed-sized processor architecture are not suitable for cryptographic computations, because they are designed for single-precision operations. An n-bit processor is able to use numbers up to 2^n. On the other hand numbers that are used in cryptographic applications can grow up to thousands of bits. This problem can not be handled by a hardware solution. Instead, it should be solved in the software layer.

The large numbers of cryptographic applications are split into small pieces. Each of these pieces is called a *word,* and the length of each word is n-bit for an n-bit processor. Implementing large numbers in such words is called *multiprecision*.

CRYMPIX is a multiprecision cryptographic library. It was designed and developed by Hüseyin Hışıl, as the implementation part of the graduate thesis "A Distributed Multiprecision Cryptographic Library Design" (Hışıl 2005). Development language of CRYMPIX is ANSI C which provides portability to the library. So it is suitable for different hardware platforms.

CRYMPIX uses 32-bit words to implement large numbers. It includes various structures that are designed for not only mutiprecision integers but also different types of mathematical forms that are used in cryptography, such as finite fields, prime numbers and elliptic curves.

A layered structure is applied during the development of the library (Figure 3.1). Basic arithmetic operations, such as addition, subtraction, multiplication, division, exponentiation, modulus and modular exponentiation, are implemented in the *multiprecision layer* which is placed on top of the *kernel layer*. Input-output (io) operations and some base functions that are used by the multiprecision layer are placed in the kernel layer. *Finite layer* is structured on the multiprecision layer and implements the finite field mathematical forms and operations. Prime numbers and elliptic curves are included in this layer. Finally the *cryptography layer* is placed on the top. The cryptographic protocols such as encryption-decryption algorithms are placed in this layer.

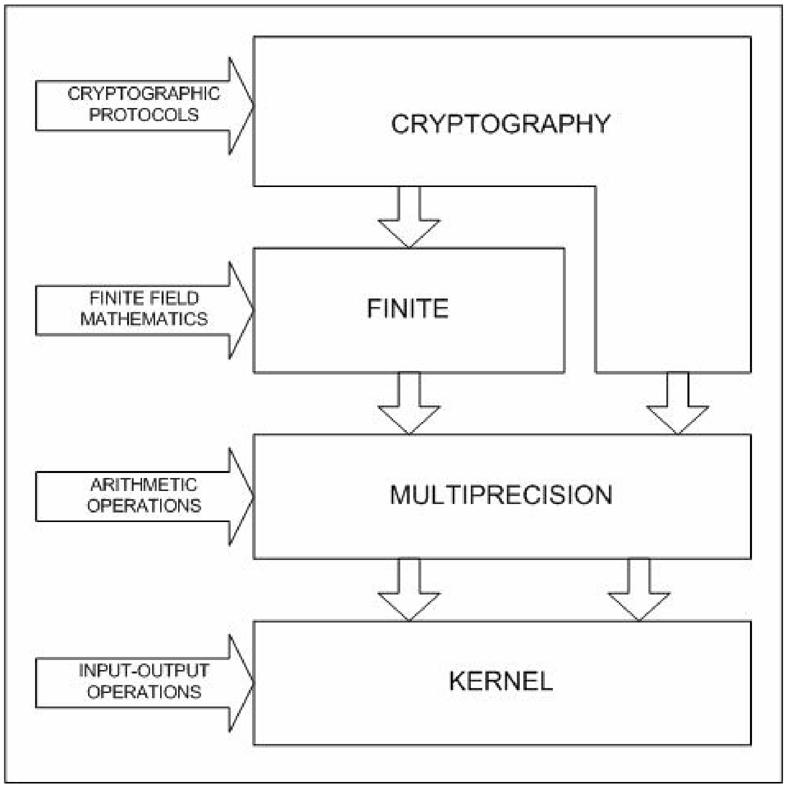

Figure 3.1. Layered Structure of CRYMPIX

Each layer calls the functions of a lower layer, except the cryptography. It calls both the functions of finite and multiprecision layers. Also none of them knows about the upper layers.

As a cryptographic protocol, each key exchange protocol is placed in the cryptography layer. Each protocol was implemented independently from the others. Key structures are defined for each protocol with their initializer and finalizer functions. Also both structure and function names are specified according to coding conventions of CRYMPIX.

3.2. Protocols

As shown in Figure 3.2, there is a common structure for all implemented key exchange protocols. Each of them includes the owner's public and private keys, the other party's public key and the shared key. The other parameters depend on the selected protocol. These structures include both generated and computed values. Despite the fact that computed values can be obtained from the generated values anytime, in order to avoid the delay that is caused by calculation time in each computed value call, they are also placed in the key structures (i.e. owner's public key).

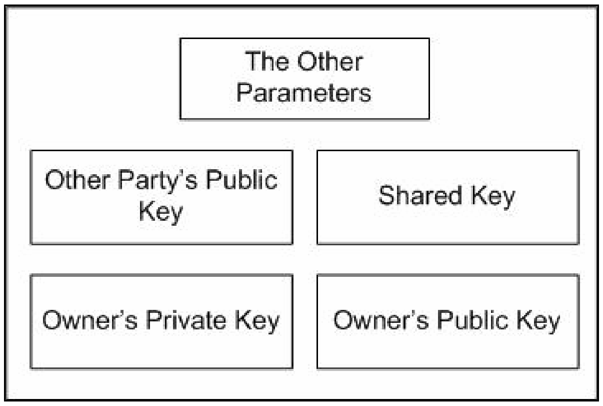

Figure 3.2. Generic Key Structure

3.2.1. Diffie-Hellman Key Exchange Protocol

Domain parameters and key pairs of Diffie-Hellman Key Exchange Protocol which are described in sections 2.1.2 and 2.1.3 are provided with their corresponding conventions in CRYMPIX in Table 3.1.

Table 3.1. Diffie-Hellman Key Structure Comparison

Protocol Description	**CRYMPIX Implementation**
p	Fp
g	g
x_A	x
y_A	X
y_B	Y
K	key

The implemented key structure of the protocol is shown in *Figure 3.3*. The name of the structure is defined as DH_FP; DH for Diffie-Hellman and FP for the prime finite field arithmetic.

```
typedef struct{
      FP_t fp;
      FP_t q;
      MI_t g;
      MI_t x;
      MI_t X;
      MI_t Y;
      MI_t key;
}DH_FP_t[1], *DH_FP;
```

Figure 3.3. Implemented Key Structure for Diffie-Hellman KE

MI_t types are multiprecision integer values. *FP_t fp* is a finite field representation that indicates prime p. The domain parameter q is not placed in the structure because it is required for only generation of g.

Generation of these domain parameters requires prime number generation, primality testing and the generation of generator number. Prime number generation and primality testing functions are included in CRYMPIX. However, just these functions are not enough to generate the domain parameters of this protocol. There is a prerequisite for the generation of p and q primes such that q must be a prime factor of p-1. Such q prime is required for the generation of the generator g. But factorization of p-1 is an NP-complete problem that this protocol is based on. Consequently, instead of calculating a prime p and trying to factorize it, q is generated using CRYMPIX's prime generation function then 2q+1 is tested with prime testing functions (Menezes 1996). Until (2q+1) is a valid prime p, these two steps are repeated. "2" is selected as the other factor of p in order to make the calculation of generator g easy.

Additionally, modular exponentiation and pseudorandom number generator functions of the library are used during domain parameter generation.

Similar to key structure, functions are also named according to the naming convention. Hence, each function starts with *dh_fp* prefix, as shown in Table 3.2.

Table 3.2. Functions of Diffie-Hellman Key Exchange Protocol

Function Name	Input	Process
dh_fp_init	DH_FP new, uni_t rtype	to initialize the Diffie-Hellman key structure “new”
dh_fp_kill	DH_FP dh	to free the Diffie-Hellman key structure “dh”
dh_fp_generate_domain_parameters	MI p, MI g, uni_t len, uni_t cert, MI seed	to generate domain parameters p and g with given length and certainty using the seed
dh_fp_generate_key_pair	DH_FP dh, MI seed	to generate a private key and calculate a public key using given seed and to store these keys in the Diffie-Hellman key structure “dh”
dh_fp_calculate_session_key	DH_FP dh	to calculate session key using public and private keys in the Diffie-Hellman key structure “dh” and to store it in the same key structure

3.2.2. Elliptic Curve Diffie-Hellman Key Exchange Protocol

CRYMPIX contains the fundamental structures for elliptic curve cryptography, such as an elliptic curve structure which is denoted as EC_FP_t and a point structure which is denoted as ECP_FP_t. Also required functions, like point multiplication, are included in the library. On the other hand, there is no generation function for an elliptic curve in the library because it is an expensive process to create a suitable elliptic curve for the cryptographic usage. Instead of generating elliptic curves, pre-defined NIST (WEB_4 2007, WEB_5 2007) curves are used in implementation. Such predefinition does not cause a security problem because elliptic curve structures are public parameters of elliptic curve cryptography. Also elliptic curve functions are designed as parametric functions. Hence, the function caller does not have to use pre-defined curves in the library. It is possible to define a curve outside of the library and pass it through the elliptic-curve functions. In order to obey the coding conventions of CRYMPIX (WEB_6 2007), the parametric functions were designed to implement this protocol.

Domain parameters and key pairs of Elliptic Curve Diffie-Hellman Key Exchange Protocol which are described in sections 2.2.2 and 2.2.3 are shown with their corresponding conventions in CRYMPIX in Table 3.3.

Table 3.3. Elliptic Curve Diffie-Helman Key Structure Comparison

Protocol Description	**CRYMPIX Implementation**
p	ec→fp
a	ec→a
b	ec→b
x_G	g→x
y_G	g→y
n	ec→n
x_A	x
x_{QA}	X→x
y_{QA}	X→y
x_{QB}	Y→x
y_{QB}	X→y
x_K	key→x
y_K	key→y

The implemented key structure of the protocol is shown in Figure 3.4. The name of the structure is defined as DH_EC_FP; DH for Diffie-Hellman, EC for elliptic curve and FP for finite field arithmetic.

```
typedef struct{
      EC_FP_t ec;
      ECP_FP_t g;
      MI_t x;
      ECP_FP_t X;
      ECP_FP_t Y;
      ECP_FP_t key;
}DH_EC_FP_t[1], *DH_EC_FP;
```

Figure 3.4. Implemented Key Structure for Elliptic Curve Diffie-Hellman KE

Similar to key structure, functions are also named according to the naming convention. Hence, each function starts with *dh_ec_fp* prefix, as shown in Table 3.4.

Table 3.4. Functions of Elliptic Curve Diffie-Hellman Key Exchange Protocol

Function Name	Input	Process
dh_ec_fp_init	DH_EC_FP new	to initialize the Elliptic Curve Diffie-Hellman key structure "new"
dh_ec_fp_kill	DH_EC_FP ecdh	to free the Elliptic Curve Diffie-Hellman key structure "ecdh"
dh_ec_fp_generate_key_pair	DH_EC_FP ecdh, MI seed	to generate a private key and calculate a public key using given seed and to store these keys in the Elliptic Curve Diffie-Hellman key structure "ecdh"
dh_ec_fp_calculate_session_key	DH_EC_FP ecdh	to calculate session key using public and private keys in the Elliptic Curve Diffie-Hellman key structure "ecdh" and to store it in the same key structure

3.2.3. Elliptic Curve MQV

As described in section 2.3, Elliptic Curve MQV protocol has three variants as one-, two- and three-pass authenticated key exchange protocols. The one-pass authenticated key exchange protocol is not suitable for interactive communications. The three-pass authenticated key exchange protocol requires hash and MAC algorithms, which are not in the scope of this thesis and are not included in CRYMPIX. Therefore in this thesis, it is aimed to implement two-pass authenticated key exchange protocol.

As mentioned earlier, different from the other two protocols, MQV uses an implicit signature which is computed to provide authentication. Also an additional long-term key pair is used. It is assumed that the long term key pair is shared through a certification authority between two parties of the protocol. Actually, there is no structural difference between the implementations of Elliptic Curve Diffie-Hellman and Elliptic curve MQV, except an additional key pair and an implicit key. On the other hand computations are totally different. Elliptic Curve MQV uses not only the point multiplication but also the point addition function of the library. Also two extra functions, one for implicit signature calculation and one for point derivation (Section 2.3.1) are implemented.

Domain parameters and key pairs of Elliptic Curve MQV Key Exchange Protocol which are described in sections 2.3.2 and 2.3.3 are shown with their corresponding conventions in CRYMPIX in Table 3.5.

Table 3.5. Elliptic Curve MQV Key Structure Comparison

Protocol Description	**CRYMPIX Implementation**
p	ec→fp
a	ec→a
b	ec→b
n	ec→n
x_G	g→x
y_G	g→y
r_A	x
w_A	a
R_A	X
W_A	A
R_B	B
K	key
h	h
S_A	S

The implemented key structure of the protocol is shown in Figure 3.5. The name of the structure defined as MQV_EC_FP; MQV for the name of protocol, EC for elliptic curve and FP for finite field arithmetic.

```
typedef struct{
      EC_FP_t ec;
      ECP_FP_t g;
      MI_t x;
      MI_t a;
      ECP_FP_t X;
      ECP_FP_t Y;
      ECP_FP_t A;
      ECP_FP_t B;
      ECP_FP_t key;
      MI_t h;
      MI_t S;
}MQV_EC_FP_t[1], *MQV_EC_FP;
```

Figure 3.5. Implemented Key Structure for Elliptic Curve KE

MI_t types are multiprecision integer values. *EC_FP_t* indicates an elliptic curve structure and *ECP_FP_t* indicates a point on an elliptic curve.

Similar to key structure, functions are also named according to the naming convention. So each function starts with *mqv_ec_fp* prefix, as shown in Table 3.6.

Table 3.6. Functions of Elliptic Curve MQV Key Exchange Protocol

Function Name	Input	Process
mqv_ec_fp_init	MQV_EC_FP new	to initialize the Elliptic Curve MQV key structure “new”
mqv_ec_fp _kill	MQV_EC_FP ecmqv	to free the Elliptic Curve MQV key structure “ecmqv”
mqv_ec_fp _generate_key_pair	MQV_EC_FP ecmqv, MI seed	to generate a private key and calculate a public key using given seed and to store these keys in the Elliptic MQV key structure “ecmqv”
mqv_ec_fp _calculate_session_key	DH_EC_FP ecmqv	to calculate session key using public and private keys in the Elliptic Curve MQV key structure “ecmqv” and to store it in the same key structure
mqv_ec_fp _calculate_implicit_signiture	DH_EC_FP ecmqv	to calculate implicit signature S in the Elliptic Curve MQV key structure “ecmqv”
mqv_ec_fp _derive_point	MI z, ECP_FP p, MI n	to derive elliptic curve point “p” using the given order n and store the multiprecision integer result in “z”

3.3. Comparison of Protocols

At the end of the implementation of three key exchange protocols, the algorithms are compared to understand which one is the most effective. But the effectiveness can be defined with respect to different parameters. In the scope of this thesis, the execution times of the key pair generation and session key calculation are selected as the parameter of effectiveness.

In order to make an objective comparison, the generated keys must be providing the same level of the security. As mentioned in section 2.3, the *bits of security*, in other words the number of operations that is required to break a cryptographic algorithm, indicates the security level. Hence, the key lengths must be chosen in the intervals of the same bits of security level for the comparison.

The execution times of the key pair generations and shared key calculations are measured for comparison. A test code for each protocol is implemented in CRYMPIX to measure execution times. The configuration of the machine on which the tests run is as the following:

- CPU, Intel Pentium IV 2.99GHz
- RAM, 2 GB
- Operating System, Windows XP Professional

Table 3.7 shows the time measurement results of Diffie-Hellman Key Exchange algorithm. The first two columns identify bits of security and corresponding key lengths in terms of bits. The third column identifies chosen bit lengths for the public (N) and private (L) keys. The minimum bit lengths are chosen for keys; hence the second and the third columns have the same values. The forth column indicates the time measurement for generating domain parameters, p, q and g. The fifth column indicates public-private key pair generation time and the last one indicates shared key calculation time.

Table 3.7. Time Measurements of Diffie-Hellman Key Exchange

Bits of security	**DHKE** (N: bit length of private key L: bit length of public key)	**Chosen Length (bits)**	**Domain Parameter (p, q, g) Generation Time (milliseconds)**	**Key Pair Generation Time (milliseconds)**	**Shared Key Calculation Time (milliseconds)**
80	L = 1024 N = 160	L = 1024 N = 160	340877,0	11,95	11,86
112	L =2048 N = 224	L =2048 N = 224	248445,0	58,64	58,42
128	L = 3072 N =256	L = 3072 N =256	----	----	----
192	L = 7680 N =384	L = 7680 N =384	----	----	----
256	L = 15360 N =512	L = 15360 N =512	----	----	----

As shown in 4th column of Table 3.7, generating domain parameters takes to much time in order to generate suitable primes as mentioned in 2.1.2. Two prime numbers, q and p, are generated as $p = q * 2^k$. Bit length of q must be at least equal to bit length of private key. Also bit length of p must be at least equal to bit length of public key. Generating such parameters for larger bit lengths (bit length ≥ 3072 bits) take hours for CRYMPIX implementation of Diffie-Hellman Key Exchange. For this reason, in Table 3.7, the last three rows of time measurements are empty. Consequently, corresponding key pair generation and shared key generation time cells are empty. Results of time measurements for generating domain parameters are not acceptable for cryptographic usage. They must be reduced which can be achieved by reducing generation time of larger primes in the library. Existing prime generators can be refactored or more effective algorithms can be implemented. On the other hand, because these parameters are public and implemented functions have parametric structure, it is possible to use predefined domain parameters.

Table 3.8. The Time Measurements of Elliptic Curve Diffie-Hellman Key Exchange

Bits of security	**ECDHKE** (f: bit length of the order of base point G)	**Chosen Length (bits)**	**Key Pair Generation Time (milliseconds)**	**Shared Key Calculation Time (milliseconds)**
80	f = 160 – 223	192	33,1	15,07
112	f = 224 - 255	224	45,57	21,27
128	f = 256 – 383	256	58,94	27,96
192	f = 384 – 511	384	163,57	75,6
256	f = 512 +	521	369,11	169,93

Table 3.8 and Table 3.9 show the time measurement results of Elliptic Curve Diffie-Hellman Key Exchange and Elliptic Curve MQV Key Exchange algorithms. Both of them include the same columns which indicate the same parameters. The first two columns identify bits of security and corresponding base point order (n) lengths (f: the bit length of n) in terms of bits. The third column identifies the chosen bit lengths for the base point order n. Actually, not bit lengths of order n, but NIST curves that have orders in the corresponding intervals in the second column, are chosen. There is no time measurement for domain parameter generation because predefined curves are used. The forth column

identifies public-private key pair generation time and the last one indicates shared key calculation time.

Table 3.9. The Time Measurements of Elliptic Curve MQV Key Exchange

Bits of security	**ECMQV** (f: bit length of the order of base point G)	**Chosen Length (bits)**	**Key Pair Generation Time (milliseconds)**	**Shared Key Calculation Time (milliseconds)**
80	f = 160 - 223	192	33,57	23,28
112	f = 224 - 255	224	45,09	31,66
128	f = 256 – 383	256	59,21	42,37
192	f = 384 – 511	384	161,52	111,99
256	f = 512 +	521	371,68	255,44

The results of execution time measurements show larger key lengths causes longer execution times for key pair generation and shared key calculation. The execution time comparisons of implemented algorithms are shown in the graphics in Figure 3.6 and Figure 3.7.

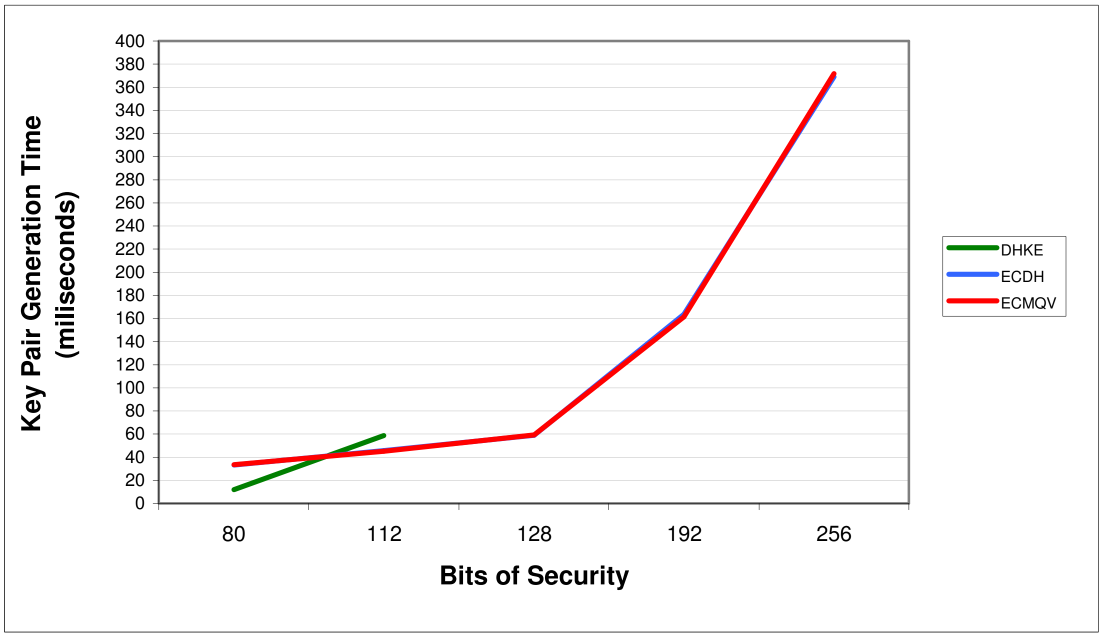

Figure 3.6. Key Pair Generation Time Comparison Table

ECC protocols have not only similar algorithms and implementation but also similar results for the same security levels. Key pair generation times are nearly same because the same steps are executed in both protocol; a random number generation and a point multiplication. Hence, graphics of generation times are on top of each other, Figure 3.6. On the other hand the session key calculation times are longer for Elliptic Curve MQV than Elliptic Diffie-Hellman because an implicit signature calculation is included, Figure 3.7.

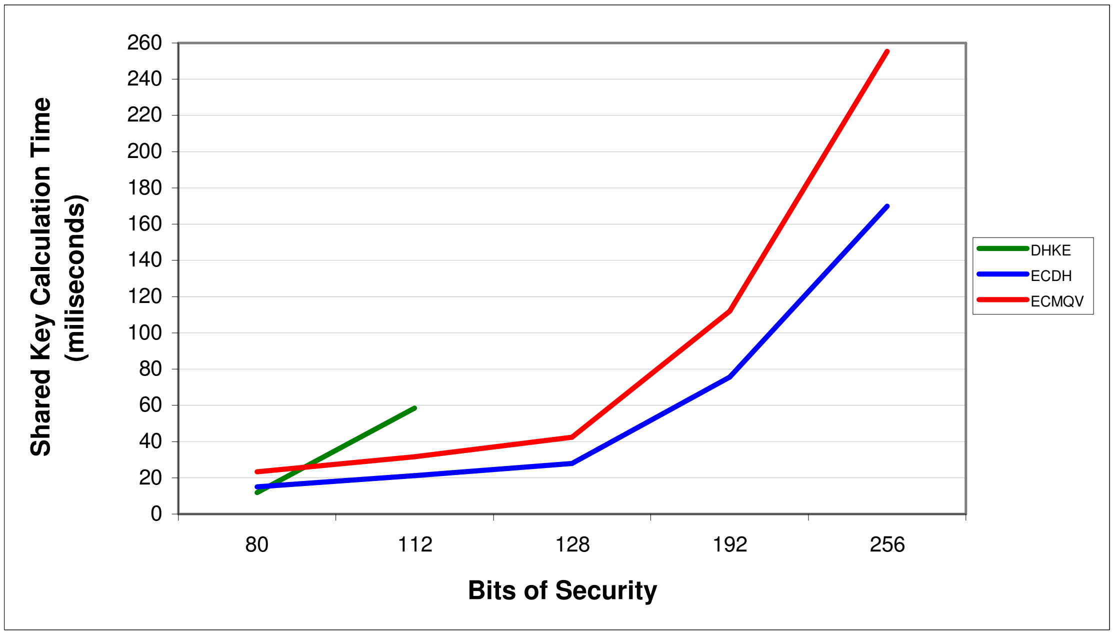

Figure 3.7 Session Key Calculation Time Comparison Table

CHAPTER 4

CONCLUSION AND THE FUTURE WORK

Development of the public-key cryptosystems not only became a new perspective in encryption and decryption processes but also solved the key distribution problem of the symmetrical cryptosystems. Key exchange protocols which are based on public-key cryptology make it possible to distribute key over an insecure channel. The aim of this study is to review the key exchange protocols and implement them using a multiprecision library.

During this study Diffie-Helman Key Exchange, Elliptic Curve Diffie-Hellman Key Exchange and Elliptic Curve MQV Key Exchange protocols were analyzed in detail. These protocols are based on different mathematical one way trap functions which are known as the discrete logarithm problem and the elliptic curve discrete logarithm problem. Hence, these mathematical aspects were also examined.

The protocol implementations are done at the cryptographic protocol layer of the library. In this study the parametric functions are designed and predefined curves are passed as parameters in tests.

Not only the protocols but also the test codes for general utilization plus key execution timer codes were implemented for the library. Some bugs in the existing library functions were discovered during testing software development. The first problem is the increased execution time of the Diffie-Hellman domain parameter generation function which was up to hours. The latency observed here is caused by the slowness of prime generation function for higher bit lengths. This problem can be solved by implementing the faster prime generators.

The second problem is the pseudorandom number generator which is called by key generation functions of all protocols. Test codes call the key generator functions so many times in order to obtain more accurate execution time value. But there is only one pseudorandom number generator in the library. When the initializer function of the generator is called too many times, it causes too many loops. This problem is solved by passing initialized pseudo-random number generators to key generation functions instead of multiprecision integer seeds. But it is understood that pseudo-random number generator does not work correctly.

In conclusion, these protocols are included in the last release of CRYMPIX. Also the documentation of the functions and structures are provided for the benefit of users.

4.1. Future Work

First of all prime number generator should be refactored to get a usable Diffe-Hellman Key Exchange Protocol. Also pseudo-random generator function must be reviewed and after correction of the generator, parameters of key generation functions should be refactored.

As mentioned earlier the Elliptic Curve MQV Key Exchange has three different variations. In scope of this study the two-pass authenticated key exchange protocol is implemented. Three-pass authenticated key exchange protocol which includes key confirmation is left as the future study. It is necessary to implement a message authentication code (MAC) algorithm and a hash algorithm such as one in SHA-2 family, for the three-pass authenticated protocol.

REFERENCES

ANSI (American National Standards Institute) X9.42, 1998. Public Key Cryptography for the Financial Services Industry: *Agreement of Symmetric Keys Using Discrete Logarithm Cryptography.*

ANSI (American National Standards Institute) X9.63, 1999. Public Key Cryptography for the Financial Services Industry: *Key Agreement and Key Transport Using Elliptic Curve Cryptography.*

Atay, S., 2006. “Performance issues of elliptic curve cryptographic implementations”, *Unpublished Ph.D. Dissertation Thesis, Ege University, Graduate School of Natural and Applied Sciences.*

Barker, E., Barker, W., Burr, W., Polk, W. and Smid, M., 2007, NIST Special Publication 800-57: *Recommendation for Key Management Part1 - General*, NIST (National Institute of Standards and Technology).

Diffie, W. and Hellman, M.E, 1976. “New Directions in Cryptography”, *IEEE Transactions on Information Theory*, Vol.22, pp.644-654.

Hankerson, D., Menezes, A.J. and Vanstone, S., 2004. *A Guide to Elliptic Curve Cryptography* (Springer).

Hışıl, H., 2005. “A distributed multiprecision cryptographic library design”, *Unpublished MS. Thesis, İzmir Institute of Technology, The Graduate School of Engineering and Science.*

Law, L., Menezes, A., Qu, M., Solinas, S. and Vanstone, S., 1998. “Efficient Protocol for Authenticated Key Agreement”, *Designs, Codes and Cryptography,* V.28, pp.119-134.

Menezes, A.J., Oorschot, P.C.V. and Vanstone, S.A., 1996. *Handbook of Applied Cryptography* (CRC Press).

Merkle, R.C, 1978. "*Secure* Communications over Insecure Channels", *Communications of the ACM*, V.21, pp.294-299.

Rivest, R., Shamir. A. and Adleman, L., 1978. "A method for obtaining Digital Signatures and Public Key Cryptosystems", *Communications of the ACM, ACM Press*, Vol.21, pp. 120-126.

Schneier, B., 1996. *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C* (Wiley).

WEB_1 2007, VeriSign Official Web Site, http://www.verisign.com

WEB_2 2007, GeoTrust Official Web Site, http://www.geotrust.com/

WEB_3 2007, Commondo Official Web Site, http://www.comodo.com/

WEB_4 2007, National Institute of Standards and Technology, http://nist.gov/

WEB_5 2007, Computer Security Research Center, http://csrc.nist.gov/

WEB_6 2007, CRYMPIX Home Page, http://crympix.iyte.edu.tr

Zimmermann, P., 1995, *PGP Source Code and Internals* (MIT Press)

APPENDIX A

BASIC STRUCTURES AND TYPE DEFINITIONS IN CRYMPIX

1. The type definition *of uni_t*:

```c
/**
 * Type definition for a single precision variable.
 **/
typedef unsigned long uni_t;
```

2. The type definition *of uni*:

```c
/**
 * Type definition for a pointer to a single precision variable.
 **/
typedef uni_t *uni;
```

3. Structure for *MA_t:*

```c
/**
 * Type definition for an array.This struct is used for low level
 * integer and polynomial arithmetic.
 **/
typedef struct {
      uni_t l; /* Number of digits */
      uni n; /* Starting address of digits */
} MA_t[1], *MA;
```

4. Structure for *MI_t:*

```c
/**
 * Type definition for an integer. A vector is encapsulated with
 * the sign.
 **/
typedef struct {
      SIGN s; /* Sign of the integer */
      MA_t v; /* Vector part of the integer */
} MI_t[1], *MI;
```

5. Structure for *FP_t:*

```
typedef struct{
      MI_t ch; /* Field characteristic */
      MI_t nd2; /* n^-1 in montgomery's nresidue system */
      uni_t nd; /* n[0]^-1 in montgomery's nresidue system due to
dusse and kaliski */
      uni_t rtype; /* Representation type */
}FP_t[1], *FP;
```

6. Structure for *EC_FP_t:*

```
/**
 * Type definition for an elliptic curve, E(Fp)
 * Tentatively designed for now. To be modified in the future.
 **/
typedef struct{
      FP_t fp;
      MI_t a;
      MI_t b;
      MI_t n; /* #E. */
      MI_t t0, t1, t2, t3, t4, t5; /* Temp variables. */
      BOOL is_a_3; /* TRUE if a = -3. */
}EC_FP_t[1], *EC_FP;
```

7. Structure for ECP_*FP_t:*

```
/**
 * Type definition for an elliptic curve point, P in E(Fp).
 * The elliptic curve must be defined over Fp.
 **/
typedef struct{
      MI_t x;
      MI_t y;
      MI_t z;
      MI_t z2;
      MI_t z3;
      MI_t az4;
      BOOL inf; /* TRUE if point at infinity, FALSE otherwise. */
      uni_t ct; /* Coordinate type */
}ECP_FP_t[1], *ECP_FP;
```