SECURE SMS USING SIMPLIFIED PGP

GÖKSEL CAN

Submitted to the Graduate School of Science of Engineering
In partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Engineering

IŞIK UNIVERSITY

2008

SECURE SMS USING SIMPLIFIED PGP

APPROVED BY:

Assoc. Prof. Ercan Solak ______________________

(Thesis Supervisor)

Assist. Prof. Olcay Taner Yıldız ______________________

Assist. Prof. Hasan Fehmi Ateş ______________________

APPROVAL DATE:

SECURE SMS USING SIMPLIFIED PGP

Abstract

While communicating with your partner, the ability to communicate in a secure way is a fundamental and important requirement for information security. Security of communication requires identification and verification of partner's id and decrypting partner's message. These and other kinds of controls need to be verified between two parties of communication. The need to have a scalable and secure binding between public keys and names gave rise to Pretty Good Privacy (PGP). PGP is a public key infrastructure, which allows verification of the binding between a message and a public key. In PGP people can use public keys issued by their friends in order to verify the binding. The primary goal of PGP is to provide cryptographic privacy and authentication for messages. The goal of the master's thesis was to implement a simplified PGP encryption application for mobile devices such as cellular phones using java programming language over java 2 micro edition platform.

According to our knowledge, there was no public implementation of PGP for mobile devices. In order to fill this gap, we simplified the general PGP structure. We did not implement web of trust. We used a simple exchange of public keys instead. We constructed a new simplified version of PGP that uses 160-bit SHA-1 as hash algorithm, 1024-bit RSA algorithm as asymmetric cipher and 128-bit AES as symmetric cipher, a CLDC configured mobile devices such as mobiles phones.

BASİTLEŞTİRİLMİŞ PGP İLE GÜVENLİ SMS

Özet

Karşınızdaki kişi ile güvenli bir şekilde ileitişim kurmak bilgi güvenliğinin önemli bir unsurudur. İletişimin güvenliği karşımızdaki kişinin tanımlanmasını ve kimliğini doğrulanmasını ve mesajın şifrelenmesini gerektirir. Bunlar gibi gereksinimler karşılıklı olarak iletişimin tarafları tarafından sağlanmalıdır. Bütün bu ihtiyaçların sonucu herkese açık şifrelere dayanan PGP ortaya çıkmıştır. PGP açık şifreleri kullanarak mesaj ve kimlik doğrulamasını sağlayan şifreleme yapısıdır. Bu tezin amacı taşınabilir cihazlar için PGP nin basitleştirilmiş bir versiyonunun J2ME kullanılarak gerçekleştirilmesidir.

Bildiğimiz kadarıyla şu ana kadartaşınabilir cihazlar için gerçekleştirilmiş bir PGP versiyonu mevcut değil. Bu boşluğu doldurmak için, PGP yapısını basitleştirdik ve bu amaçla güvenlik ağı yapısını kullanmadık. Basit olarak açık şifrelerin karşılıklı taraflar tarafından değişilmesine dayanan, hash oluşturak için 160-bit SHA-1 algoritması kullanan, asimetrik şifreleme için 1024-bit RSA algoritmasını kullanan, simetrik şifreleme için 128-bit AES kullanan yeni bir PGP versiyonu oluşturduk.

Acknowledgements

There are many people who helped to make my years at the graduate school most valuable. First, I thank my advisor Assoc. Prof. Ercan Solak. Having the opportunity to work with him over the years was intellectually rewarding and fulfilling. I also thank everyone who contributed to the development of this research starting from the early stages of my dissertation work. Cem Kaya provided valuable contributions to the development of this project. I thank him for his insightful suggestions.

Many thanks to professors of the department of computer engineering, who patiently answered my questions and problems. I would also like to thank to my graduate student colleagues who helped me all through the years full of class work and exams. My special thanks go to Murat Kaya whose friendship I deeply value.

The last words of thanks go to my family. I thank my parents Ahmet and Atike Can and my brother Yuksel Can for their patience and encouragement. Lastly, I thank my friends, for their endless support through this long journey.

To my parents Mr. Ahmet and Mrs. Atike Can.

Table of Contents

Abstract iii

Özet iv

Acknowledgements v

Table of Contents vii

List of Figures viii

Abbreviations ix

1. Introduction 1

2. Encryption Primitives 3

2.1 Cryptography Terminology. 4

2.2 Symmetric Encryption 4

2.3 Block Cipher 6

2.4 Advanced Encryption Standard (AES) 6

2.5 Rivest-Shamir-Adleman (RSA). . . . . 10.

2.6 Hash Function 12

3. Pretty Good Privacy (PGP) 14

4. Java 2 Platform Micro Edition (J2ME) 18

4.1 Connected Device Configuration (CDC) 19

4.2 Connected Limited Device Configuration (CLDC) 19

4.3 Bouncy Castle API 19

5. Implementation 23

5.1 Key Generation 26

5.2 Key Exchange 31

5.3 Encryption Process 40

5.4 Decryption Process 43

6. Conclusions 47

References 48

List of Figures

Figure 2.1 SubBytes 8
Figure 2.2 ShiftRows. 8
Figure 2.3 MixColumns 9
Figure 2.4 AddRoundKey 9
Figure 3.1 PGP Architecture 16
Figure 4.1 Class diagram key pair generator 20
Figure 4.2 Class diagrams of key parameters 20
Figure 4.3 Class diagram of SHA-1 hash algorithm 21
Figure 4.4 Class diagram of RSA algorithm 21
Figure 4.5 Class diagrams of AES 22
Figure 5.1 Encryption process 24
Figure 5.2 Decryption process. 25
Figure 5.3 Menu of the application. 26
Figure 5.4 Generate Key Midlet 28
Figure 5.5 Generator Code Segment 28
Figure 5.6 Generate Key Midlet 29
Figure 5.7 Display Ownkey Midlet 30
Figure 5.8 Code segment Stores Own key 31
Figure 5.9 List Received Keys Midlet 32
Figure 5.10 Send Your Key Midlet and SMS Receive Midlet 33
Figure 5.11 Display Own Key Midlet and Display Other's Key Midlet 34
Figure 5.12 SMS Send Midlet and SMS Receive Midlet 36
Figure 5.13 SMS Send Midlet and SMS Receive Midlet Cont'd 37
Figure 5.14 SMS Send Midlet and SMS Receive Midlet Cont'd 38
Figure 5.15 SMS Send Midlet and SMS Receive Midlet Cont'd 39
Figure 5.16 SHA-1 hash generator code 40
Figure 5.17 RSA encryption and Decryption code 41
Figure 5.18 AES CFB Mode Block Cipher Encryption code 42
Figure 5.19 AES CFB Mode Block Cipher Decryption code 46

Abbreviations

AES	Advanced Encryption Standard
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
DES	Data Encryption Standard
J2ME	Java 2 Micro Edition
NIST	National Institute of Standards
PGP	Pretty Good Privacy
RSA	Rivest-Shamir-Adleman Algorithm
SHA-1	Secure Hash Algorithm 1

Chapter 1
Introduction

Ways of communication between people, organizations and businesses has changed dramatically with the introduction of the Internet and its applications. People are buying and selling, making payments, exchanging e-mails, publishing information and browsing for information and doing many other things over the Internet. Using the Internet, a person can communicate digitally with a much larger number of people than before. This increased number of contacts and communication opens new opportunities, but also increases security and privacy problems.

Security of communication requires identification and verification of communication partner's id and decrypting partner's message. These and other kinds of controls need to be verified between two parties of communication. The need to have a scalable and secure binding between public keys and names gave rise to Pretty Good Privacy (PGP). PGP is a public key infrastructure, which allows verification of the binding between a message and a public key [1]. In PGP people can use public keys issued by their friends in order to verify the binding. The primary goal of PGP is to provide cryptographic privacy and authentication for messages.

The work presented here has been conducted as part of a master's thesis. The goal of the master's thesis was to implement an instance of a simple PGP application for mobile devices such as cellular phones using java programming language over java 2 micro edition platform.

According to our knowledge, there was no public implementation of PGP for mobile devices around. As a result of this fact, to implement PGP for mobile device with memory and computation limitations and limited capabilities, we had to simplify it. So, web of trust was not implemented.

This thesis is organized as follows: Chapter 2 and 3 presents some background information on the thesis topic. Chapter 4 explains the sort of technology and platform used for implementation. The main part of this report is chapter 5, where the architectural design and implementation is described. The last chapter is conclusion and future work.

Chapter 2
Encryption Primitives

Cryptography is the study of means of converting information from its normal, comprehensible form into an incomprehensible format, by changing it to an unreadable format without secret knowledge [2]. Cryptography helps and ensures secrecy in important communications, such as those of spies, military leaders, and diplomats. In recent decades, the field of cryptography has expanded its remit in two ways. Firstly, it provides mechanisms for more than just keeping secrets by constructing schemes like digital signatures and digital money. Secondly, cryptography has had widespread use by many civilians who do not have extraordinary needs for secrecy, although typically it is transparently built into the infrastructure for computing and telecommunications, and users are not aware of it.

For thousands of years, humans were using cryptography. But until recent decades, it has not been much developed. This is called classical cryptography. Classical cryptography methods use pen and paper or simple mechanical machines. The inventions of complex mechanical machines and computers, such as enigma motors, lead to more sophisticated and efficient algorithms and systems of encryption. Until the 1970s, secure cryptography was largely the concern of governments. Two events have since brought it squarely into the public domain: the creation of a public encryption standard and the invention of public-key cryptography. Nowadays, to understand and to use cryptography, someone should know and understand some encryption primitives' .These primitives include encryption process, decryption process, plaintext, cipher text, secret key, public key, symmetric cipher, block cipher, modes of operation, asymmetric cipher, hash functions.

2.1 **Cryptography Terminology**

We first introduce some basic terminology of cryptography. Others will be defined as they are used in the text.

Plaintext: Original intelligible message or data that is fed into the algorithm as input.

Cipher text: Scrambled message produced as output. It depends on the plain text and the secret key. For a given message, two different keys will produce two completely different cipher texts. The cipher text has the appearance of random stream of data.

Secret key: The secret key is a parameter of the encryption algorithm. The algorithm will produce a different output depending on the specific key being used at the time. The exact operations performed by the algorithm depend on the key.

Encryption process: Transforming the plaintext into cipher text by the encryption algorithm as parameterized by the secret key.

Decryption process: Transforming the cipher text back into plaintext by the decryption algorithm as parameterized by the secret key

2.2 **Symmetric Encryption**

Two forms of encryption are in common use: symmetric encryption and public-key, or asymmetric, encryption. Symmetric encryption is a form of cryptography in which encryption process and decryption process are performed using the same key. It is also known as conventional encryption. Symmetric encryption transforms plaintext into cipher text using a secret key and an encryption algorithm.

An n-bit block cipher is a function $E : V_n \times K \rightarrow V_n$, such that for each key $K \in \mathbf{K}$, C=E(P,K) is an invertible mapping (the encryption function for K) from V_n to V_n. Here V_n denotes the space of n-bit sequences, P denotes the plaintext, C is the cipher text. Encryption and Decryption are respectively denoted as

$C = E (P, K)$,

$P = D (C, K)$,

The mapping D: V_n x $\mathbf{K} \rightarrow V_n$ is the inverse of E for a particular key K.

Using the same key and a decryption algorithm, the plaintext is recovered from the cipher text. The two types of attack on an encryption algorithm are cryptanalysis which is based on properties of the encryption algorithm, and brute-force which involves trying all possible keys. Traditional symmetric ciphers use substitution and transposition techniques. Substitution techniques map plaintext bits into cipher text bits. Transposition techniques systematically transpose the positions of plaintext bits. Rotor machines are sophisticated pre-computer hardware devices that use substitution techniques. Steganography is a technique for hiding a secret message within a larger one in such a way that others cannot discern the presence or contents of the hidden message. An example is the watermarking technique.

There are two requirements for secure use of symmetric encryption: We need a strong encryption algorithm. At least, we would like to have an algorithm such that an attacker who knows the algorithm and has sufficient cipher texts would not be able to decipher the cipher text or figure out the key. Moreover, the attacker should be unable to decrypt cipher text or discover the key even if he or she has a number of cipher texts together with the plaintext that produced each cipher text. Sender and receiver must have exchanged and stored copies of the secret key in a secure way and must keep the key secure. If someone can discover the key and knows the algorithm, all communication using this key is readable. It is assumed that decryption of message is impractical on the basis of the cipher text with knowledge of the encryption/decryption algorithm. In other words, algorithm is not to be kept secret; only the secret key should be kept secret. This is known as Kerchoff principle [2]. It makes symmetric encryption feasible and widespread because of this feature of symmetric encryption.

Kerchoff principle allows us to develop low-cost chip implementations of data encryption algorithms. These chips are widely used as a part in a number of products. The principal security problem is maintaining the secrecy of the key while using symmetric encryption.

2.3 **Block Cipher**

In cryptography, a block cipher is a symmetric key cipher which operates on fixed-length groups of bits, termed blocks, with an unvarying transformation. When encrypting, a block cipher takes a 128-bit block of plaintext as input, and outputs a corresponding 128-bit block of cipher text. The exact transformation is controlled using a secret key as a second input. Decryption is similar: the decryption algorithm takes, in this example, a 128-bit block of cipher text together with the secret key, and yields the original 128-bit block of plaintext. To encrypt messages longer than the block size a mode of operation is used.

The block size, n, is typically 64 or 128 bits, although some ciphers have a variable block size. 64 bits was the most common length until the mid-1990s, when new designs began to switch to the longer 128-bit length. One of several modes of operation is generally used along with a padding scheme to allow plaintexts of arbitrary lengths to be encrypted. Each mode has different characteristics in regard to error propagation, ease of random access and vulnerability to certain types of attack. Typical key sizes k includes 40, 56, 64, 80, 128, 192 and 256 bits. As of 2006, 80 bits is normally taken as the minimum key length needed to prevent brute force attacks.

2.4 **Advanced Encryption Standard (AES)**

In 1999, NIST proposed a new version of its Data Encryption Standard (DES) that indicated that Data Encryption Standard should only be used for legacy systems and that triple DES be used. 3DES has two attractions that assure its widespread use over the next few years. First, with its 168-bit key length, it overcomes the vulnerability to brute-force attack of DES. Second, the underlying encryption algorithm in 3DES is the same as in DES. This algorithm has been subjected to more scrutiny than any other encryption algorithm over a longer period of time, and

no effective cryptanalytic attack based on the algorithm rather than brute force has been found. Accordingly, there is a high level of confidence that 3DES is very resistant to cryptanalysis. If security were the only consideration, then 3DES would be an appropriate choice for a standardized encryption algorithm for decades to come.

The principal drawback of 3DES is that the algorithm is relatively sluggish in software. The original DES was designed for mid-1970s hardware implementation and does not produce efficient software code. 3DES, which has three times as many rounds as DES, is correspondingly slower. A secondary drawback is that both DES and 3DES use a 64-bit block size. For reasons of both efficiency and security, a larger block size is desirable. Because of these drawbacks, 3DES is not a reasonable candidate for long-term use. As a replacement, NIST in 1997 issued a call for proposals for a new Advanced Encryption Standard, which should have security strength equal to or better than 3DES and significantly, improved efficiency [4].

In 1997, the National Institute of Standards and Technology announced the initiation of a new symmetric-key block cipher algorithm as the new encryption standard to replace the DES. The new algorithm would be named the Advanced Encryption Standard. The algorithm must support at a minimum block sizes of 128-bits, key sizes of 128-, 192-, and 256-bits, and should have a strength at the level of the triple DES, but should be more efficient then the triple DES. In addition, the algorithm(s), if selected, must be available royalty-free, worldwide.

In 1998, NIST announced a group of fifteen AES candidate algorithms. The five AES finalist candidate algorithms were MARS, RC6, Rijndael, Serpent, and Twofish.

These finalist algorithms received further analysis during a second, more in-depth review period named the Round 2. In the Round 2, comments and analysis were sought on any aspect of the candidate algorithms, including, but not limited to, the following topics: cryptanalysis, intellectual property, cross-cutting analyses of all of the AES finalists, overall recommendations and implementation issues. In 2000, NIST announced that it has selected Rijndael to propose for the AES. The Advanced Encryption Standard (AES) was published by NIST (National Institute of Standards and Technology) in 2001.

AES is a symmetric block cipher that is intended to replace DES as the approved standard for a wide range of applications. AES is a block cipher intended to replace DES for commercial applications. It uses a 128-bit block size and a key size of 128, 192, or 256 bits. AES does not use a Feistel structure. Instead, each full round consists of four separate functions: byte substitution, permutation, arithmetic operations over a finite field, and XOR with a key.

High-level cipher algorithm uses Key Expansion using Rijndael's key schedule. Each round involves Sub Bytes, Shift Rows, Mix Columns, Add Round Key operations.

1. Sub Bytes — a non-linear substitution step where each byte is replaced with another according to a lookup table.

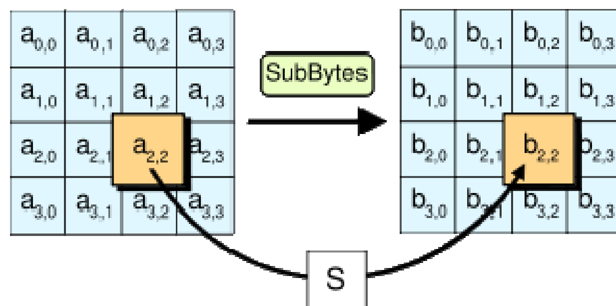

Figure 2.1 Sub Bytes [5]

2. Shift Rows — a transposition step where each row of the state is shifted cyclically a certain number of steps.

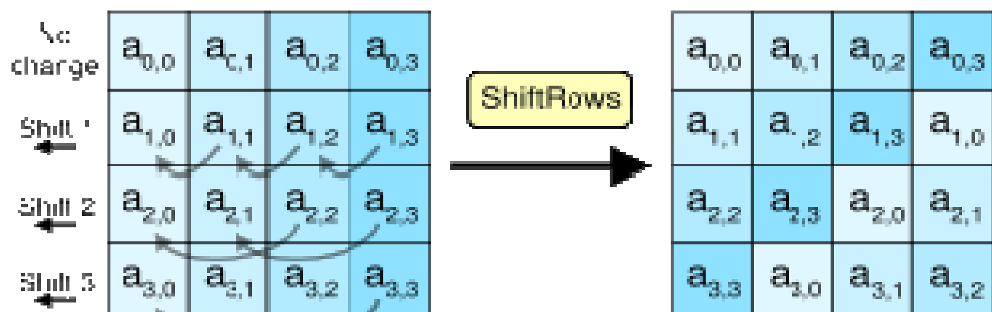

Figure 2.2 Shift Rows [5]

3. Mix Columns — a mixing operation which operates on the columns of the state, combining the four bytes in each column

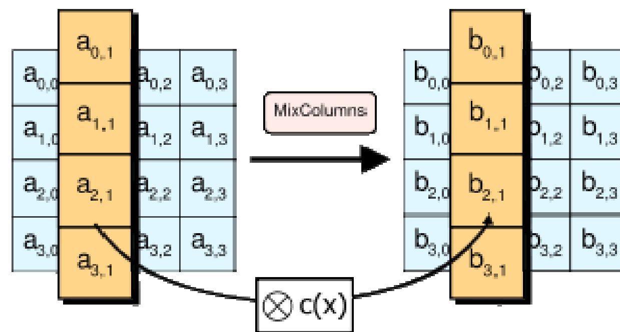

Figure 2.3 Mix Columns [5]

4. Add Round Key — each byte of the state is combined with the round key; each round key is derived from the cipher key using a key schedule.

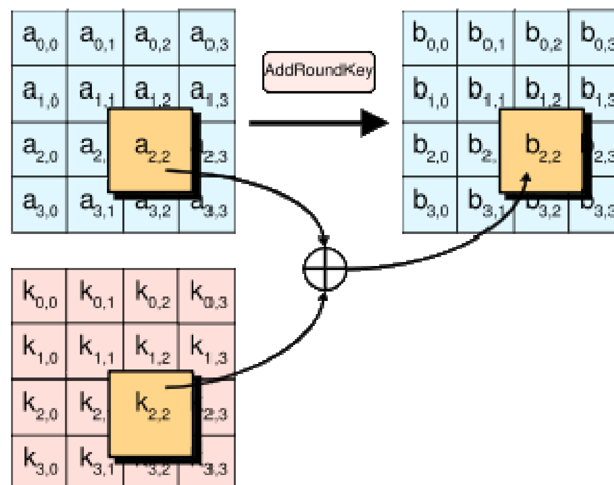

Figure 2.4 Add Round Key [5]

In the Final Round only three steps are done. (No MixColumns) Namely SubBytes ShiftRows AddRoundKey

2.5 **Rivest-Shamir-Adleman (RSA)**

Public-key cryptography, also known as asymmetric cryptography, is a form of cryptography in which a user has a pair of cryptographic keys; a public key and a private key. The private key is kept secret, while the public key may be widely distributed. The keys are related mathematically, but the private key cannot be practically derived from the public key. A message encrypted with the public key can be decrypted only with the corresponding private key.

The two main branches of public key cryptography are confidentiality and authentication. A message encrypted with a recipient's public key cannot be decrypted by anyone except the recipient possessing the corresponding private key. This is used to ensure confidentiality. A message signed with a sender's private key can be verified by anyone who has access to the sender's public key, thereby proving that the sender signed it and that the message has not been tampered with. This is used to ensure authenticity.

An analogy for public-key encryption is that of a locked mailbox with a mail slot. The mail slot is exposed and accessible to the public; its location is in essence the public key. Anyone knowing the street address can go to the door and drop a written message through the slot; however, only the person who possesses the key can open the mailbox and read the message.

An analogy for digital signatures is the sealing of an envelope with a personal wax seal. The message can be opened by anyone, but the presence of the seal authenticates the sender. We assume that wax seals can not be copied or forged.

A central problem for public-key cryptography is proving that a public key is authentic, and has not been tampered with or replaced by a malicious third party. The usual approach to this problem is to use a public-key infrastructure (PKI), in which one or more third parties, known as certificate authorities, certify ownership of key pairs. Another approach, used by PGP, is the "web of trust" method to ensure authenticity of key pairs.

Public key techniques are much more computationally intensive than purely symmetric algorithms. The judicious use of these techniques enables a wide variety of applications. In

practice, public key cryptography is used in combination with secret-key methods for efficiency reasons. For encryption, the sender encrypts the message with a secret-key algorithm using a randomly generated key, and that random key is then encrypted with the recipient's public key. For digital signatures, the sender hashes the message using a cryptographic hash function and then signs the resulting "hash value". Before verifying the signature, the recipient also computes the hash of the message, and compares this hash value with the signed hash value to check that the message has not been tampered with.
The RSA cryptosystem, named after its inventors R. Rivest, A. Shamir, and L. Adleman, is the most widely used public-key cryptosystem. It may be used to provide both secrecy and digital signatures and its security is based on the intractability of the integer factorization.

Algorithm: Key generation for RSA:

Each entity A should do the following:

1. Generate two large random (and distinct) primes p and q, each roughly the same size.
2. Compute $n = pq$ and $\varphi = (p - 1)(q - 1)$.
3. Select a random integer e, $1 < e < \varphi$, such that $\gcd(e, \varphi) = 1$
4. Use the extended Euclidean algorithm to compute the unique integer d, as $1 < d < \varphi$, such that $ed \equiv 1 \pmod{\varphi}$.
5. A's public key is (n, e); A's private key is d. [3]

The integers e and d in RSA key generation are called the encryption exponent
and the decryption exponent, respectively, while n is called the modulus.

Algorithm: RSA public-key encryption/decryption:

B encrypts a message m for A, which A decrypts.

1. *Encryption.* B should do the following:
   (a) Obtain A's authentic public key (n, e).
   (b) Represent the message as an integer m in the interval $[0, n - 1]$.
   (c) Compute $c = m^e \bmod n$.
   (d) Send the cipher text c to A.

2. *Decryption.* To recover plaintext m from c, A should do the following:

(a) Use the private key d to recover $m = c^d \mod n$. [3]

Example (RSA encryption with artificially small parameters):

Key generation: Entity A chooses the primes p = 2357, q = 2551, and computes n =pq = 6012707 and φ = (p−1)(q−1) = 6007800. A chooses e = 3674911 and, using the extended Euclidean algorithm, finds d = 422191 such that ed ≡ 1 (mod φ). A's public key is the pair (n = 6012707, e = 3674911), while A's private key is d = 422191.

Encryption: To encrypt a message m = 5234673, B uses an algorithm for modular exponentiation to compute c = me mod n = 52346733674911 mod 6012707 = 3650502,
and sends this to A.

Decryption: To decrypt c, A computes cd mod n = 3650502422191 mod 6012707 = 5234673. [3]

2.6 Hash Function

In cryptography, a cryptographic hash function is a transformation that takes variable-length input string and returns a fixed-size string, which is called the hash value. Hash is also known as message digest. Hash functions with this property are used for a variety of computational purposes, including cryptography. The hash value is a concise representation of the longer message or document from which it was computed. The message digest is a sort of "digital fingerprint" of the larger document. Cryptographic hash functions are used to do message integrity checks and digital signatures in various information security applications, such as authentication and message integrity.

The two most-commonly used hash functions are MD5 and SHA-1. A typical use of a cryptographic hash would be as follows: Alice poses a tough math problem to Bob, and claims she has solved it. Bob would like to try it himself, but would yet like to be sure that Alice is not bluffing. Therefore, Alice writes down her solution, appends a random number, computes its hash and tells Bob the hash value while keeping the solution and nonce secret. When Bob comes

up with the solution himself a few days later, Alice can prove that she had the solution earlier by revealing the nonce to Bob. This is an example of a simple commitment scheme; in actual practice, Alice and Bob will often be computer programs, and the secret would be something less easily spoofed than a claimed puzzle solution.

Another important application of secure hashes is verification of message integrity. Determination of whether or not any changes have been made to a message or a file, for example, can be accomplished by comparing message digests calculated before, and after, transmission or any other event.

A message digest can also serve as a means of reliably identifying a file; the source code management system uses the sha1sum of various types of content file content, directory trees, ancestry information, to uniquely identify them.

A related application is password verification. Passwords are usually not stored in clear text, for obvious reasons, but instead in digest form. To authenticate a user, the password presented by the user is hashed and compared with the stored hash. This is sometimes referred to as one-way encryption.

For both security and performance reasons, most digital signature algorithms specify that only the digest of the message be "signed", not the entire message. Hash functions can also be used in the generation of pseudorandom bits.

SHA-1, MD5, and RIPEMD-160 are among the most commonly-used message digest algorithms as of 2005 [2]. Hashes are used to identify files on peer-to-peer file sharing networks. For example, in an ed2k link, a MD4-variant hash is combined with the file size, providing sufficient information for locating file sources, downloading the file and verifying its contents. Magnet links are another example. Such file hashes are often the top hash of a hash list or a hash tree which allows for additional benefits [2].

Chapter 3
Pretty Good Privacy (PGP)

PGP is an encryption system that provides cryptographic privacy and authentication. It was originally created by Philip Zimmermann in 1991. The primary goal of PGP is to provide cryptographic privacy and authentication for e-mails. PGP is a Public Key Infrastructure and allows individuals to verify bindings between public keys and e-mail addresses by following a chain of certificates.

PGP encryption uses public-key cryptography and includes a system which binds the public keys to a user name. PGP makes the certificate issuer responsible; to inform people about certificates he revoked.

This authentication model scales up in an unhierarchical manner. In the PGP "web of trust," any individual can be a Certificate Authority (CA) for any other principals in the system by signing their "key certificates" which is simply a pair name key. Evidently, the signing relationship forms a web structure. Any single "CA" in the web is not well trusted or not trusted at all. The theory is that with enough such signatures, the association name, key could be trusted because not all of these signers would be corrupt. Thus, when Alice wants to establish the authenticity of Bob's key, she should request to see a number of Bob's "key certificates." If some of the issuing "CAs" of these certificates are "known" by Alice "to some extent," then she gains a certain level of authenticity about Bob's public key. Alice can demand Bob to provide more "certificates" until she is satisfied with the level of the trust [1]. To reduce to complexity of PGP for some technical purposes, we simplified general PGP structure by not use level of authenticity and Certificate Authority structure. We used exchange and verification of public keys each other instead.

Commonly, when encrypting a message, the sender uses the public key half of the recipient's key pair to encrypt a symmetric cipher session key. That session key is used, in turn, to encrypt the plaintext of the message. The recipient of a PGP-encrypted message decrypts the session key using his private key (the session key was encrypted by the sender using his public key). Next, he decrypts the cipher text of the message using the session key. Use of two ciphers in this way was chosen, despite higher complication, in part because of the very considerable difference in operating speed between asymmetric key and symmetric key ciphers (the difference is often a factor of 1000 or more). This approach also makes it easily possible to send the same encrypted message to two or more recipients. Figure 3.1 gives the operations of PGP encryption and decryption.

PGP provides a confidentiality and authentication service that can be used for electronic mail and file storage applications. PGP is based on algorithms that have survived extensive public review and are considered extremely secure. Specifically, the package includes RSA, DSS, and Diffie-Hellman for public-key encryption; CAST-128, IDEA, and 3DES for symmetric encryption; and SHA-1 for hash coding. It has a wide range of applicability, from corporations that wish to select and enforce a standardized scheme for encrypting files and messages to individuals who wish to communicate securely with others worldwide over the Internet and other networks. It was not developed by, nor is it controlled by, any governmental or standards organization. For those with an instinctive distrust of "the establishment," this makes PGP attractive. PGP is now on an Internet standards track (RFC 3156).

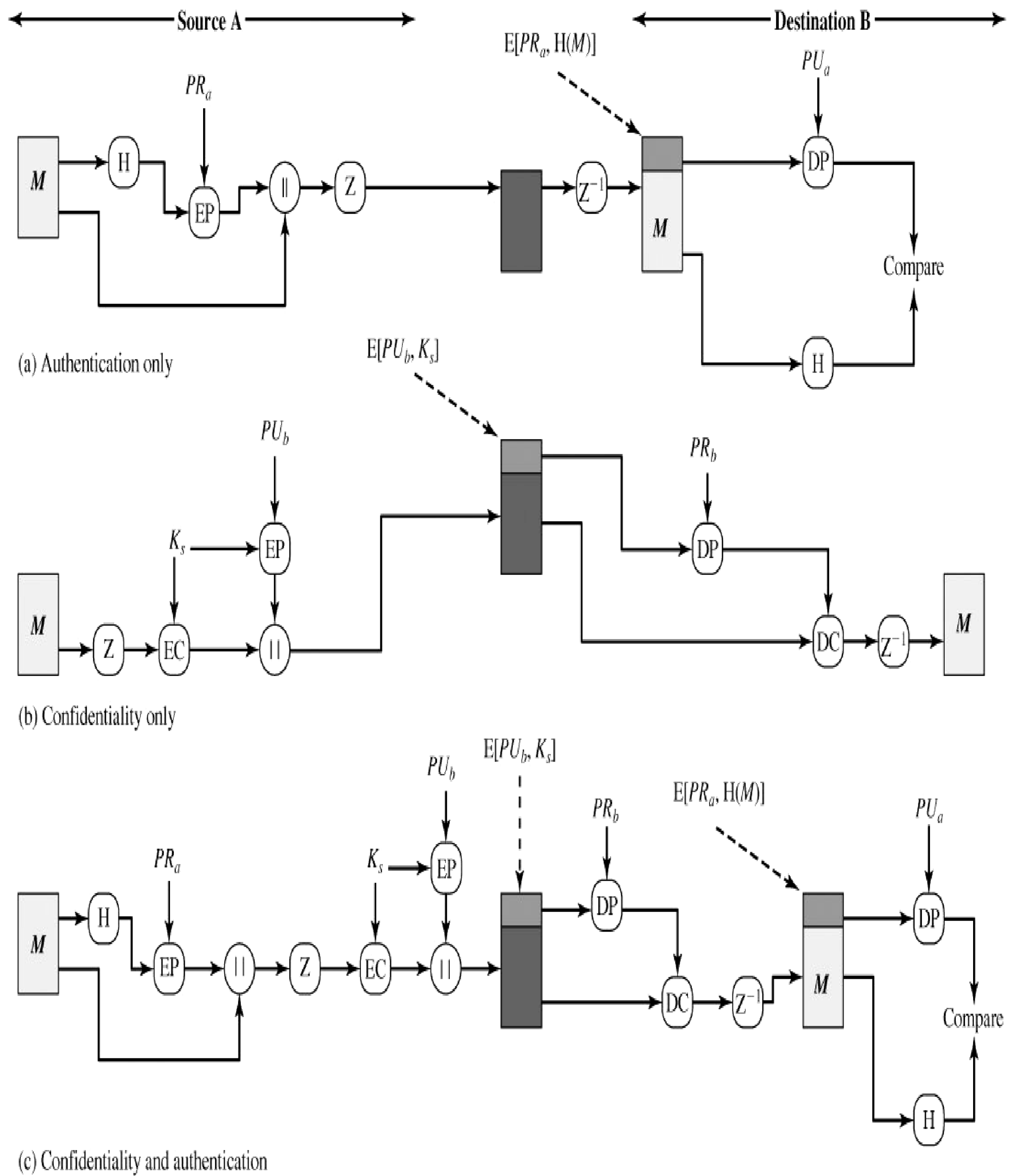

Figure 3.1 PGP architecture [1]

PGP has following parts: Digital signature, Message encryption, Compression, Email compatibility, Segmentation. Digital signature refers to hash code of a message that is created using SHA-1. This message digest is encrypted using DSS or RSA with the sender's private key

and included with the message. Message encryption refers to a message encryption using AES, CAST-128, IDEA or 3DES with a one-time session key generated by the sender. The session key is encrypted using Diffie-Hellman or RSA with the recipient's public key and included with the message. Message may be compressed, for storage or transmission, using ZIP.

Chapter 4
Java 2 Platform Micro Edition (J2ME)

Sun Microsystems has defined three Java platforms, each of which addresses the needs of different computing environments:

- Java 2 Standard Edition (J2SE)
- Java 2 Enterprise Edition (J2EE)
- Java 2 Micro Edition (J2ME)

The Java platform consists of the Java application programming interfaces (APIs) and the Java virtual machine (JVM). Java APIs are libraries of compiled code that we can use in our programs. They let you add ready-made and customizable functionality to save you programming time. Java programs are run by another program called the Java VM. Rather than running directly on the native operating system, the program is interpreted by the Java VM for the native operating system. This means that any computer system with the Java VM installed can run Java programs regardless of the computer system on which the applications were originally developed.

J2ME specifically addresses the large, rapidly growing consumer space, which covers a range of devices from tiny commodities, such as pagers to mobiles phones. Like the larger Java editions, Java 2 Micro Edition aims to maintain the qualities that Java technology. The idea behind J2ME is to provide comprehensive application development platforms for devices and applications. Furthermore, J2ME allows device manufacturers to open up their devices for widespread third-party application development and dynamically downloaded content without losing the security or the control of the underlying manufacturer platform [6]. J2ME has two configurations: Connected Device Configuration (CDC) and Connected Limited Device Configuration (CLDC).

4.1 Connected Device Configuration (CDC)

CDC has been defined as a version of Java 2 Standard Edition (J2SE) with the CLDC classes added to it [7]. Therefore, CDC was built upon CLDC, and as such, applications developed for CLDC devices also run on CDC devices. CDC provides a standardized, portable Java 2 virtual machine for consumer electronic and embedded devices, such as smart phones. These devices run a 32-bit microprocessor and have more than 2 MB of memory.

4.2 Connected Limited Device Configuration (CLDC)

CLDC was created by the Java Community Process, which has standardized Java building block for small, resource-constrained devices. The J2ME CLDC configuration provides for a virtual machine and set of core libraries to be used within an industry-defined profile [8]. A profile defines the applications for particular devices by supplying domain specific classes on top of the base J2ME configuration. CLDC outlines the most basic set of libraries and Java virtual machine features required for each implementation of J2ME on highly constrained devices. CLDC targets devices with slow network connections, limited power, 128 KB or more of non-volatile memory, and 32 KB or more of volatile memory. CLDC devices use non-volatile memory to store the run-time libraries and Virtual Machine, or another virtual machine created for a particular device.

4.3 Bouncy Castle API

BouncyCastle is a collection of API (Application Programming Interface) used in cryptography. It includes APIs for both the Java and the C# programming languages. One of the very early design considerations of BouncyCastle came from one of the developers being active in Java ME development.

The low-level API is a vendor specific set of API's that implement all the underlying cryptographic algorithms. The intent is to use the low-level API in memory constrained devices (J2ME) or when easy access to the Java cryptography libraries is not possible.

We used some core classes of the BouncyCastle API and we only get the required classes. For cryptographic purposes, BintegerClass that can hold larger integer numbers, and SecureRandom that generates random numbers, are basic required classes. While generating RSA key, we used RSAKeyPairGenerator Class with the RSAKeyParameter Class which are shown in Figure 4.1 and Figure 4.2.

We used SHA1DigestEngine that is shown in Figure 4.3, Class to generate SHA-1 hash value and RSAEngine Class that is shown in Figure 4.4, to for RSA encryption and decryptions. AESEngine Class that is shown in Figure 4.4, is used for AES encryption and decryption.

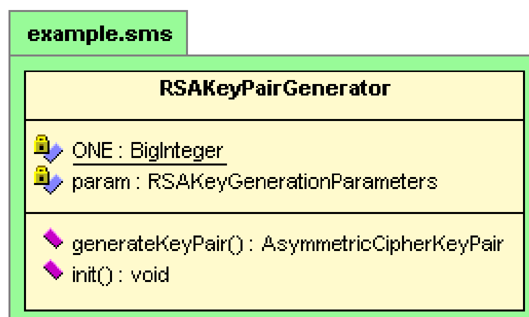

Figure 4.1 Class diagram key pair generator

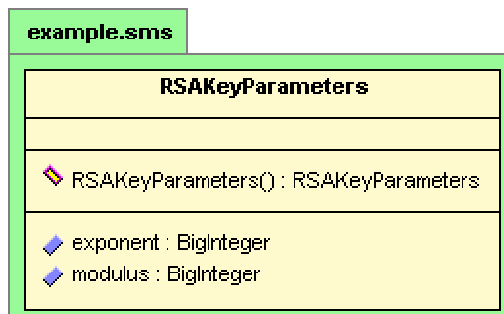

Figure 4.2 Class diagrams of key parameters

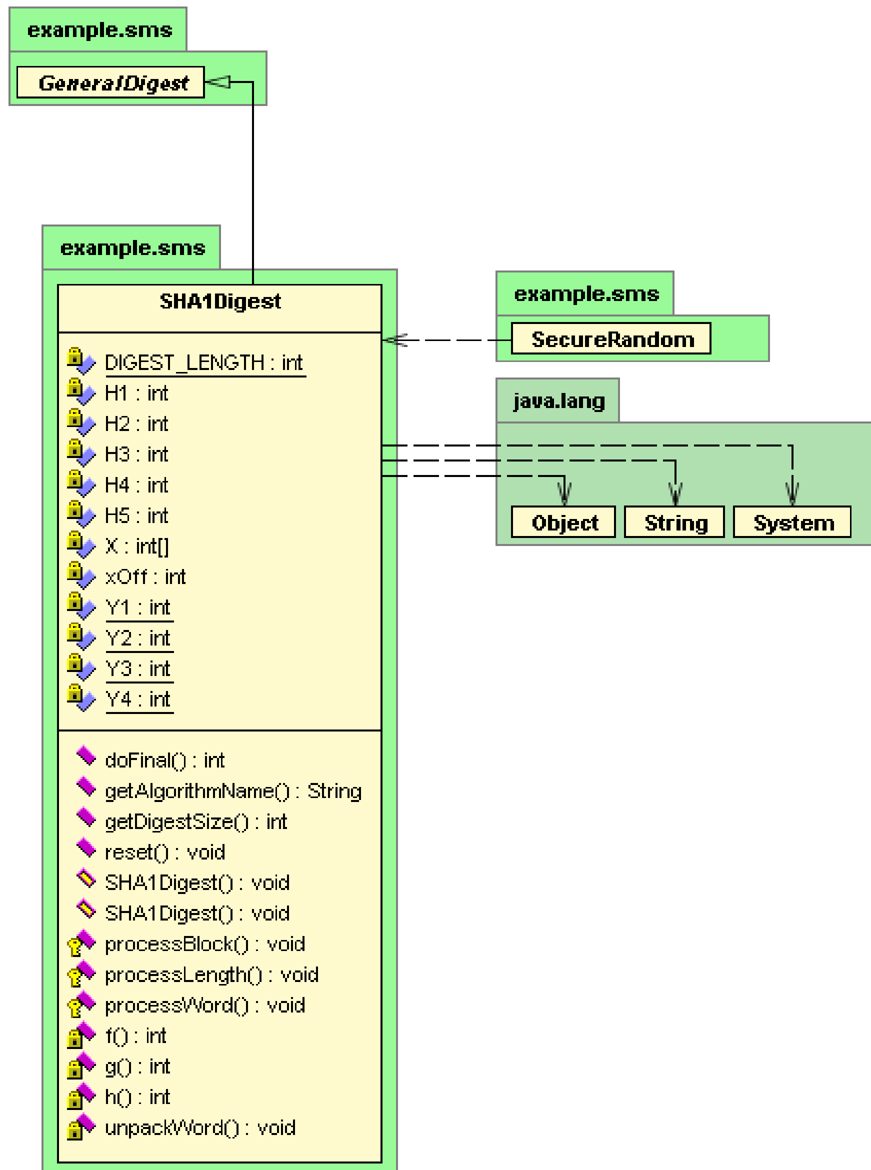

Figure 4.3 Class diagram of SHA-1 hash algorithm

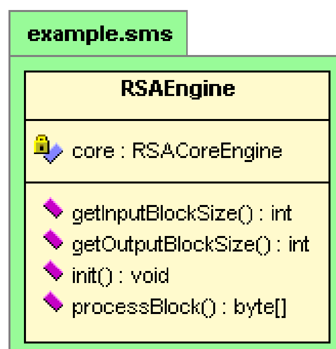

Figure 4.4 Class diagram of RSA algorithm

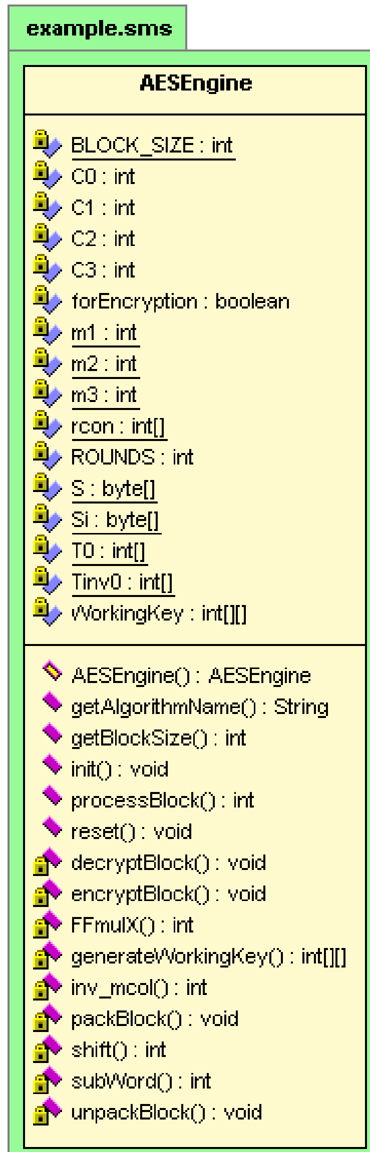

Figure 4.5 Class diagrams of AES

Chapter 5
Implementation

The primary goal of Pretty Good Privacy (PGP) is to provide cryptographic privacy and authentication for communication. The goal of the master's thesis was to implement an instance of simplified PGP application for mobile devices such as cellular phones using java programming language over java 2 micro edition platform for the encryption of SMS.

Most of the cellular phones support java. This general usage of Java programming language let us do our implementation in Java 2 Micro Edition with the configuration CDLC.

We aimed to use general and easy-to-use algorithms in the implementation. For this purpose, SHA-1 algorithm is selected as hash function to generate message digest, RSA 1024- bit algorithm is selected to sign message digest and encrypt symmetric encryption session key, AES algorithm is selected for message encryption process. General encryption process is shown in Figure 5.1 and general decryption process is shown in Figure 5.2. After installing application, Alice opens application to send message to Bob that is "Hello World!" Alice's phone number is 555 00 00. She sees the menu in Figure 5.3.

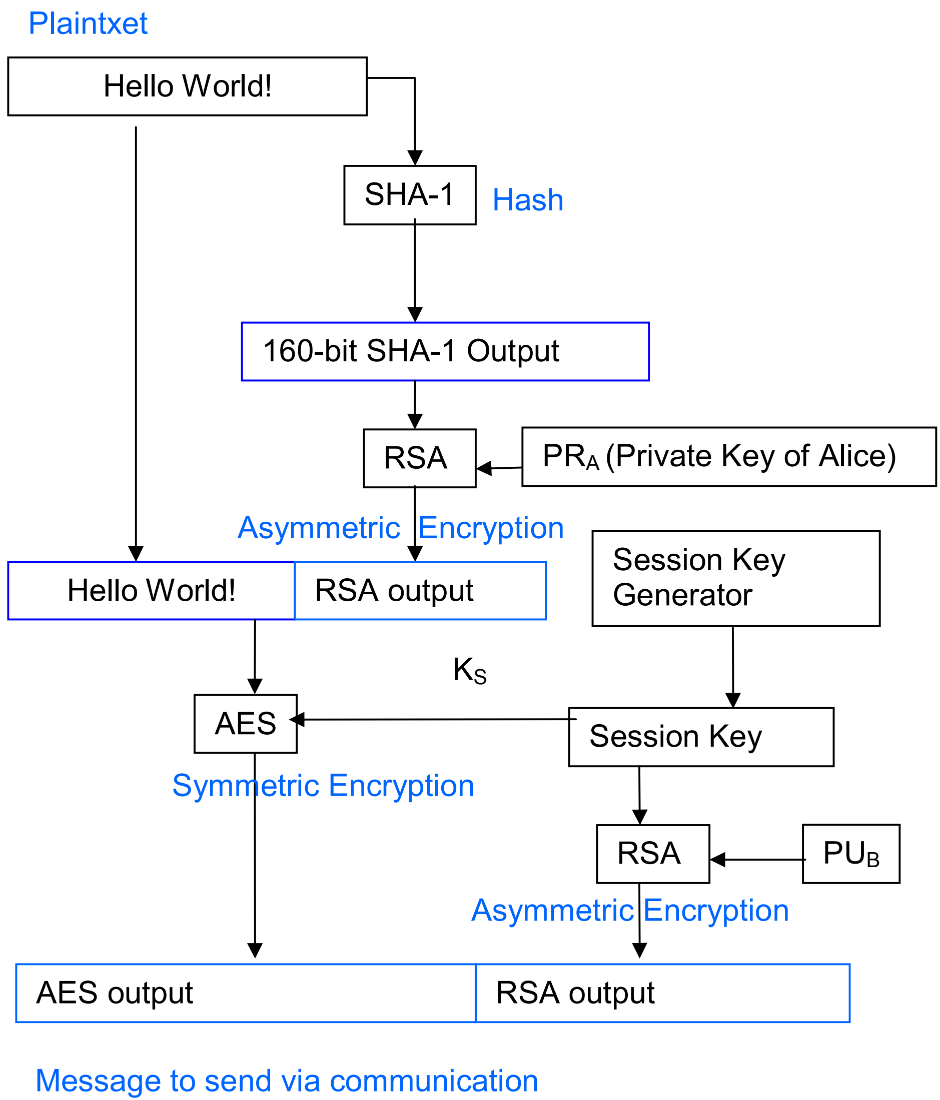

Figure 5.1 Encryption process

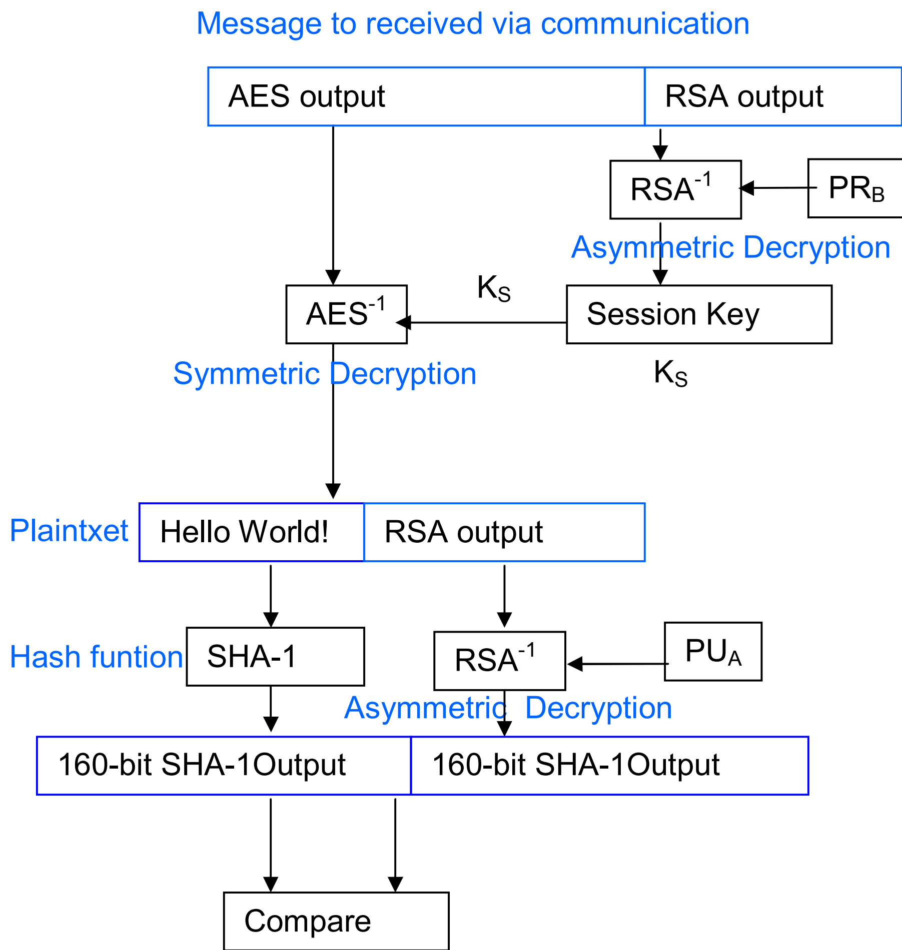

Figure 5.2 Decryption process

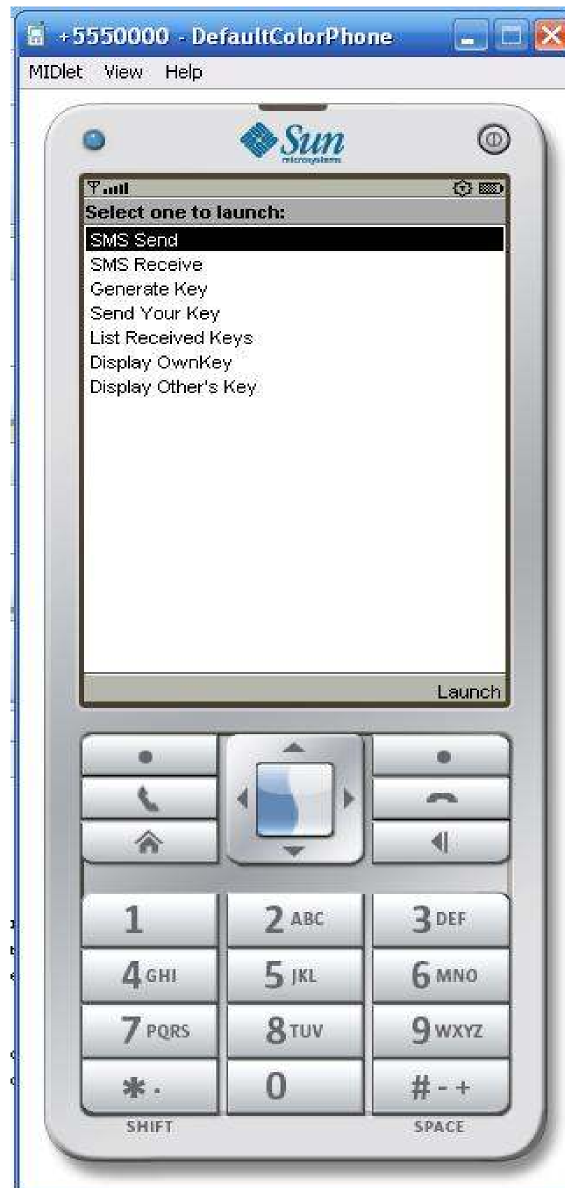

Figure 5.3 Menu of the application

5.1 Key Generation

Alice uses Generate Key midlet to generate RSA key for her because she is using the application for the first time that is shown in Figure 5.4. This midlet is needed to be used at least once before using other midlets and communication.

Alice will select Generate Key option to create an RSA key and then she will enter her phone number to the textbox as shown in Figure 5.4.

The application gets the phone number from the user because mobile phones do not support and allow any J2ME application to access private files of mobile phone. Alice has to enter his phone number correctly for application to generate his key. When she presses OK button, the code segment that is shown in Figure 5.5 will generate an asymmetric key pair.

While generating RSA key of Alice, the program generates the key passing variables public exponent, a secure random number, bit length, and certainty to the RSAGenerationParameters method. We define p and q new as instances of Biginteger Class that can hold and operate over integers with length much larger than normal int or double variable classes of Java, and set their bitlength to half our bitlength parameter. Then we generate p, prime and (p-1) relatively prime to e by using our secure random parameter in by the help of BigInteger class.

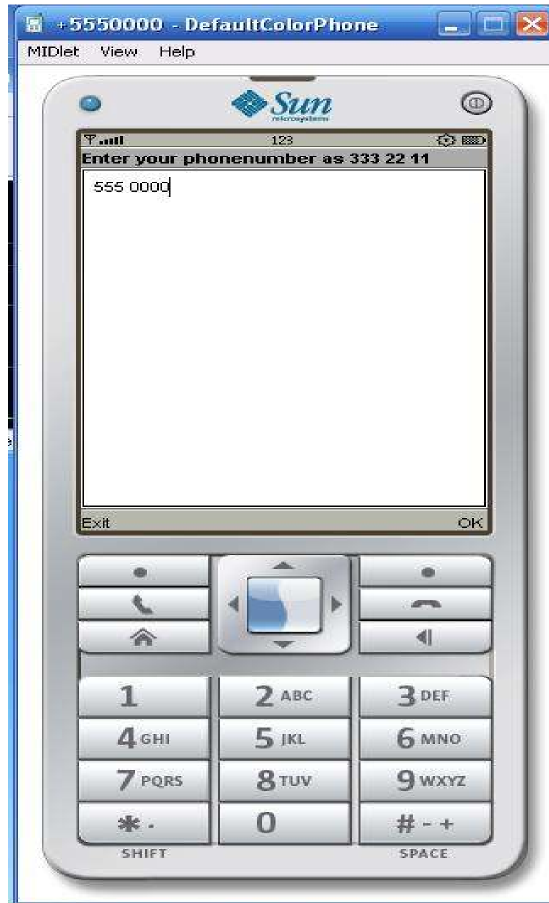

Figure 5.4 Generate Key Midlet

```
//create an instance of RSA Key Generator
RSAKeyPairGenerator  pGen = new RSAKeyPairGenerator();
// create generation parameters
RSAKeyGenerationParameters genParam=new
RSAKeyGenerationParameters( BigInteger.valueOf(0x11), new
SecureRandom(), 1024, 25);
// initialize Key Generator with generation parameters
pGen.init(genParam);
// generate RSA key pair
AsymmetricCipherKeyPair  pair = pGen.generateKeyPair();
```

Figure 5.5 Generator Code Segment

BigInteger class has its constructor method to create a BigInteger from a given secure random and bitlength. We repeat to generate BigInteger until we find one that is relatively prime to e and that BigInteger is probably prime with a probability of $1-(1/2)^{certainty}$. Variable e and p-1 must have greatest common divisor as 1. Then we generate q, prime and (q-1) relatively prime to e, and not equal to p, and we generate q using same constraints as p. Then we calculate n as product of p and q form the formula n=p*q. While phone is waiting to for the key generation process, midlet shows the screen as shown in Figure 5.6.

Then we calculate d from the formula $d= e^{-1} \bmod (p-1)(q-1)$. Alice's public key is the pair (n , e), while Alice's private key is d . Alice now has her own RSA public key and private key. As same as Alice, Bob must create his own RSA key to use application. After generation of the RSA keys, keys are stored in Record Store named as "ownkey" on the mobile phone as shown in Figure 5.7.The code segment that stores the key is shown in figure Figure 5.8. Record Stores are simpler versions of databases; it has an id part as integer and data part as bytes.

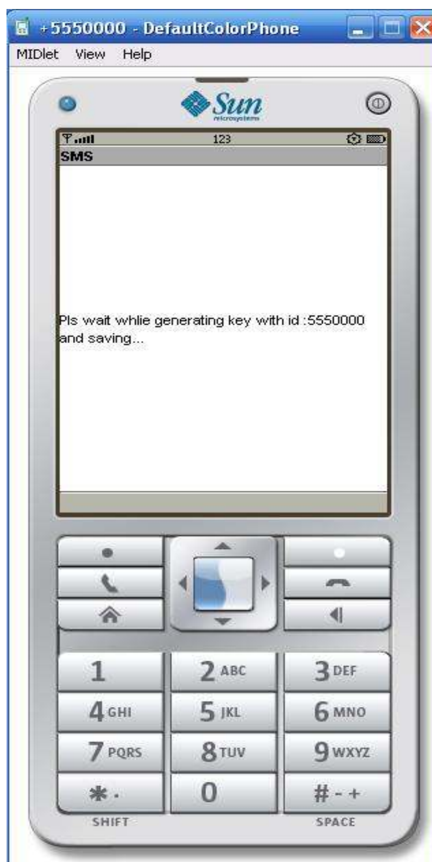

Figure 5.6 Generate Key Midlet

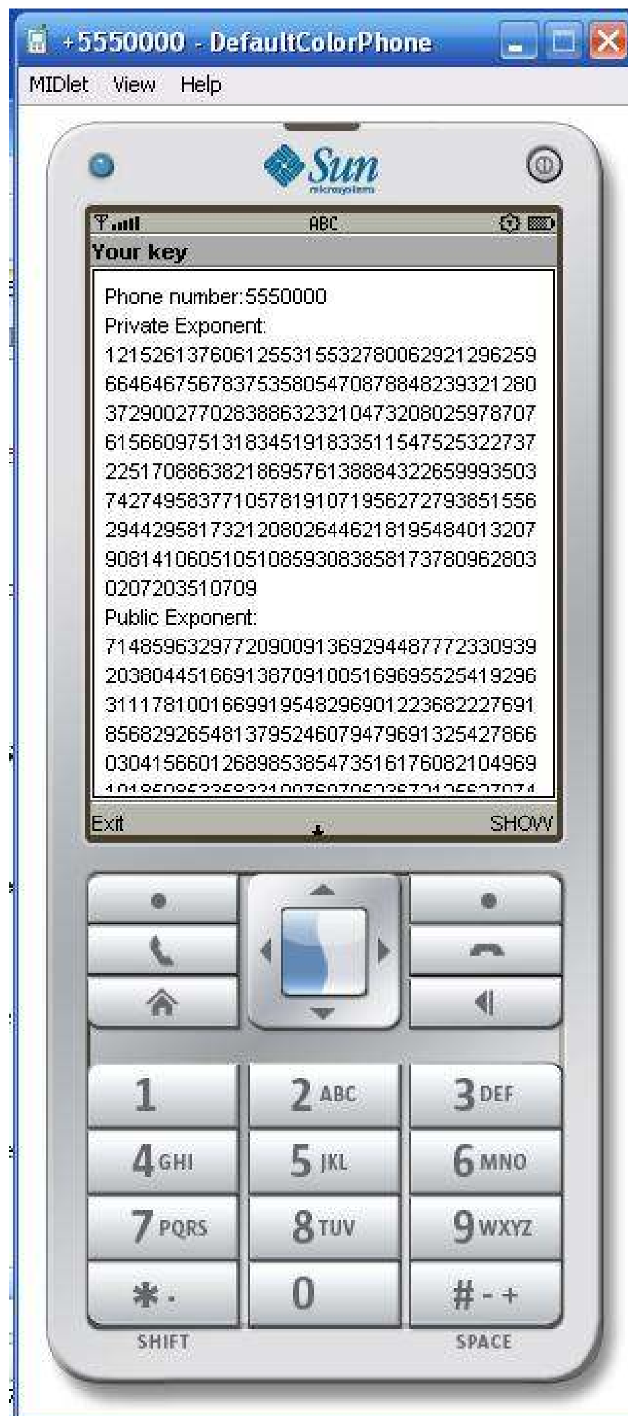

Figure 5.7 Display Ownkey Midlet

```
//define record store
RecordStore recordStore;
//open recordstore
recordStore =RecordStore.openRecordStore("ownkey", true);
//construct key with id as string
keyring=address.toString()+","+
((RSAKeyParameters)pair.getPublic()).getModulus().toString().toUpperCase()+","
+((RSAKeyParameters)pair.getPublic()).getExponent().toString().toUpperCase()+"
,"
+((RSAKeyParameters)pair.getPrivate()).getModulus().toString().toUpperCase()+"
,"
+((RSAKeyParameters)pair.getPrivate()).getExponent().toString().toUpperCase();
// write key as byte to recordstore
id=recordStore.addRecord(keyring.getBytes(), 0,keyring.getBytes().length);
```

Figure 5.8 Code segment Stores Ownkey

5.2 Key Exchange

After generating their RSA keys, they have to exchange their RSA public keys Alice use Send Your Key Midlet to send her key to Bob and Bob open his SMS Receive Midlet to receive Alice's key shown in Figure 5.10. We are using receiver midlet because mobile phones do not allow J2ME application to any SMS inbox or SMS outbox folders of the phone due to security issues. Mobile phones only allow J2ME application to use SMS port of the mobile phone. By listening the SMS port our J2ME application can get the copy of SMS.

The received public keys are stored in Record Store named as "keyring". List Received Key midlet list the all received keys from communication partners as shown in Figure 5.9. Before storing the public key, we append the phone number of sender that is read from SMS port in the beginning of the public. We are reading phone number of the sender especially from the SMS port, because of avoiding any attacker to act as Bob.

Figure 5.9 List Received Keys Midlet

Both Bob and Alice have each other's public key and own private key stored in Record Stores. To avoid impersonators, they need to confirm their public key by calling other on the phone and by comparing some bits of the public keys. To be able to compare public keys they need to view

their public keys. In order to this they use Display Own Key and Display Other's Key Midlets reading from the Record Stores. Display Own Key

Midlet displays phone number; public and private key of this person and Displays Other's Key Midlet is displays phone number and public key of one of the communication partner as shown in Figure 5.11.

Figure 5.10 Send Your Key Midlet and SMS Receive Midlet

Alice opens Display Own Key Midlet and calls Bob and Bob view Alice's public key by entering Alice's phone number to Display Other's Key Midlets. On the phone

Figure 5.11 Display Own Key Midlet and Display Other's Key Midlet

Bob asks Alice 31st, 52nd, 61st characters of the public key. If the characters are correct that means no impersonators are likely to be between Bob and Alice and they exchanged public keys successfully.

After the public key exchange, Alice wants to send his message through encryption. For this purpose, Alice opens SMS Send Midlet to and enters Bob's phone number as shown in Figure 5.12. Midlet checks if Bob's public key exists in the "keyring" record store. If the public key does not exists in the record store, midlet warns Alice to receive public key of Bob. If the public key exists in the record store, midlet asks for the message as shown in Figure 5.13 and performs PGP encryption using message as plain text as shown in Figure 5.14 and sends it to the Bob as shown in Figure 5.15.

Figure 5.12 SMS Send Midlet and SMS Receive Midlet

Figure 5.13 SMS Send Midlet and SMS Receive Midlet Cont'd

Figure 5.14 SMS Send Midlet and SMS Receive Midlet Cont'd

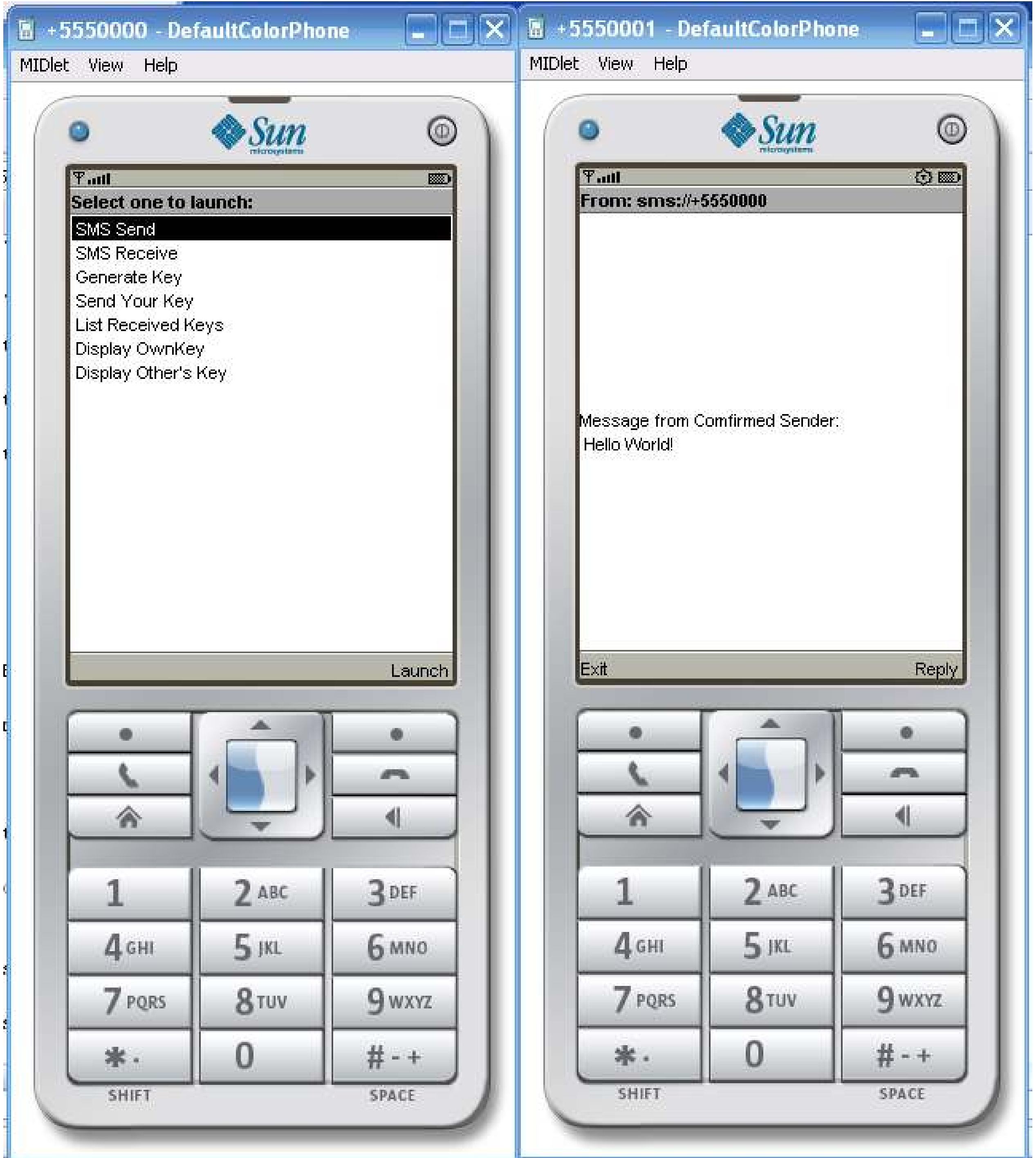

Figure 5.15 SMS Send Midlet and SMS Receive Midlet Cont'd

5.3 Encryption Process

Our message to send is the simple sentence “Hello World!” Program takes this sentence from Alice. Our plaintext=”Hello World!” At first, we generate an hash value for the plaintext using SHA-1 algorithm of which the code segment is shown in Figure 5.16.

```
//get message to a string
String plaintext=messageBox.getString();
//initialize 160 bit result byte array
byte[] result=new byte[20];
//create an instance of SHA1Digest class
SHA1Digest sha1=new SHA1Digest();
//initialize SHA-1 operation
sha1.update(plain,0,plain.length);
// get the result and store in result array
sha1.doFinal(result,0);
```

Figure 5.16 SHA-1 hash generator code

And SHA-1 algorithm produces 160-bit output as

```
2ef7bde608ce5404e97d5f042f95f89f1c232871
```

Generally hash algorithms are very fast algorithms. Next we produce the signature of the hash. For this, we encrypt sha-1 output with 1024-bit RSA algorithm using private key of Alice and the output is

```
77510099b4f1fd7d5e892708dbd26cc3b16825609d5ac508479a28acfe6e8f886
7c732ed0ac3c4c88d8eae8633e0df2af505a4346f57bafaf61e6f4353c97a92f1
70647eb235d6f162331bb0080bc337f0d8f3bba9da9b09bac0861fc62f5e18d37
91c9180bbc564fa603bc4273ae1f4f77f36e10b524cc64a5688bc01f4f327.
```

Code segment for RSA algorithm is shown in Figure 5.17.

```
// create an instance of RSAEngine
AsymmetricBlockCipher   eng = new RSAEngine();
//set encoding Public key cryptography standart 1
eng = new PKCS1Encoding(eng);
//initialize  key parameters
RSAKeyParameters    privParameters=new    RSAKeyParameters(false,
module, privateExponent);
// do encryption and store result in data named byte array
data = eng.processBlock(data, 0, data.length);
```

Figure 5.17 RSA encryption and decryption code

RSA algorithm uses some classes in implementation for key parameters, key generators, random generator and private key certificate parameter.

The time for RSA encryption and decryption is around 1500 milliseconds. After encryption of sha-1 output with RSA using Alice's private key, Message and RSA output are concatenated. Concatenated output:

77510099b4f1fd7d5e892708dbd26cc3b16825609d5ac508479a28acfe6e8f886
7c732ed0ac3c4c88d8eae8633e0df2af505a4346f57bafaf61e6f4353c97a92f1
70647eb235d6f162331bb0080bc337f0d8f3bba9da9b09bac0861fc62f5e18d37
91c9180bbc564fa603bc4273ae1f4f77f36e10b524cc64a5688bc01f4f3274865
6C6C6F20576F726C6421.

Then we encrypt concatenated output with 128-bit AES encryption using a session key. We use CFB block cipher mode as mode of operation. The code segment is shown in Figure 5.18. The average time for 1 SMS message is around 15 milliseconds. The AES output is:

4f931068319b073591bbead142c3794c23819bf9691b2a67d4676531a33236f32
bdb4641aafcd33d67defde79d0a775a7a3c4f8cabe4c1d1ad9e9000fd6c406c67
416fa8f613cb9d734693449d6d6ff1c44ff6d8c187d5c64de05c4c2d6efd3b1aa
e99e06214bdd53449f826d246c78d21e3658b83859973db4b85f099be727967ee
d1d44a45f200e3e48d22.

```
// create an instance of AES Key generator
CipherKeyGenerator a=new CipherKeyGenerator();
byte[] aeskey;
byte[] initialvector;
//initialize key generation parameters
a.init(new KeyGenerationParameters(new SecureRandom(),128));
// generate an AES key
aeskey=a.generateKey();
//initialize key generation parameters for initial vector
a.init(new KeyGenerationParameters(new SecureRandom(),128));
// generate an initial vector
initialvector=a.generateKey();
// create an CFB mode Block cipher which uses 128 bit AES
CFBBlockCipher cipher2= new CFBBlockCipher(new AESEngine(),128);
//initialize Block cipher wit key and initial vector
cipher2.init(true,new                              ParametersWithIV(new
KeyParameter(aeskey),initialvector ));
//decode plaintext to ASCII coded hexadecimal
byte[] aesinput=Hex.decode(plainRSA);
byte[] aesoutput=new byte[aesinput.length];
byte[]  in = new byte[16];
byte[]  out = new byte[16];
long aessta=System.currentTimeMillis();
//feed Block cipher by cutting plaintext into BC’s bit length
for(int i=0;i<aesinput.length;i=i+16){
      for(int j=0;j<16;j++){
            if((i+j)<aesinput.length){
                  in[j]=aesinput[i+j];
            }
      }
      //do the encryption current block
      cipher2.processBlock(in, 0, out, 0);
      System.out.println("AES             Block              "+i+":"+new
      String(Hex.encode(out)));
      //combine results of blocks
      for(int j=0;j<16;j++){
            if((i+j)<aesinput.length){
                  aesoutput[i+j]=out[j];
                  }
            }
      }
      long aesend=System.currentTimeMillis();
System.out.println("Time     for     aes     :"     +(aessta-aesend));
System.out.println("AES output"+new String(Hex.encode(aesoutput)));
```

Figure 5.18 AES CFB Mode Block Cipher Encryption code

After the encryption of output with AES using a session key, the session key is encrypted with another RSA using Bob’s public key. The output of the RSA of session key is:

```
2a403e4abe29fa4a9edc576267796cb60c42bd087dcd37c1a8b9a3106c1cffa79
31c8b053b1fe5bc58b9c10c6ff5bf949184bfc3d0c4a9a03f43004637f57c4c1e
4e2e29a561acf1f83b7282ebd0ec60c3de455e50f08fbbfeafaf3447db8e0a11f
54e58c6d5dea43de1c3032dd8606bfc707e67a3315f5cc6b1496b1b186739
```

Then the final SMS message is constructed by concatenating the AES output and RSA output of session key. “SMS” string is appended at the beginning to signal that the message contains an encrypted message. If the SMS starts with “KEY”, the coming message includes a public key of the sender. The text to send is:

```
SMS,4f931068319b073591bbead142c3794c23819bf9691b2a67d4676531a3323
6f32bdb4641aafcd33d67defde79d0a775a7a3c4f8cabe4c1d1ad9e9000fd6c40
6c67416fa8f613cb9d734693449d6d6ff1c44ff6d8c187d5c64de05c4c2d6efd3
b1aae99e06214bdd53449f826d246c78d21e3658b83859973db4b85f099be7279
67eed1d44a45f200e3e48d222a403e4abe29fa4a9edc576267796cb60c42bd087
dcd37c1a8b9a3106c1cffa7931c8b053b1fe5bc58b9c10c6ff5bf949184bfc3d0
c4a9a03f43004637f57c4c1e4e2e29a561acf1f83b7282ebd0ec60c3de455e50f
08fbbfeafaf3447db8e0a11f54e58c6d5dea43de1c3032dd8606bfc707e67a331
5f5cc6b1496b1b186739
```

The average total time for the encryption process is about 2000 milliseconds.

5.4 Decryption Process

SMS Receive Class receives text message from the SMS port of the phone and gets encrypted text message as:

SMS,4f931068319b073591bbead142c3794c23819bf9691b2a67d4676531a3323
6f32bdb4641aafcd33d67defde79d0a775a7a3c4f8cabe4c1d1ad9e9000fd6c40
6c67416fa8f613cb9d734693449d6d6ff1c44ff6d8c187d5c64de05c4c2d6efd3
b1aae99e06214bdd53449f826d246c78d21e3658b83859973db4b85f099be7279
67eed1d44a45f200e3e48d222a403e4abe29fa4a9edc576267796cb60c42bd087
dcd37c1a8b9a3106c1cffa7931c8b053b1fe5bc58b9c10c6ff5bf949184bfc3d0
c4a9a03f43004637f57c4c1e4e2e29a561acf1f83b7282ebd0ec60c3de455e50f
08fbbfeafaf3447db8e0a11f54e58c6d5dea43de1c3032dd8606bfc707e67a331
5f5cc6b1496b1b186739

Then it breaks this message into pieces as encrypted message part and encrypted session key part. Encrypted message part is:

4f931068319b073591bbead142c3794c23819bf9691b2a67d4676531a33236f32
bdb4641aafcd33d67defde79d0a775a7a3c4f8cabe4c1d1ad9e9000fd6c406c67
416fa8f613cb9d734693449d6d6ff1c44ff6d8c187d5c64de05c4c2d6efd3b1aa
e99e06214bdd53449f826d246c78d21e3658b83859973db4b85f099be727967ee
d1d44a45f200e3e48d22.

and the encrypted session key part is :

2a403e4abe29fa4a9edc576267796cb60c42bd087dcd37c1a8b9a3106c1cffa79
31c8b053b1fe5bc58b9c10c6ff5bf949184bfc3d0c4a9a03f43004637f57c4c1e
4e2e29a561acf1f83b7282ebd0ec60c3de455e50f08fbbfeafaf3447db8e0a11f
54e58c6d5dea43de1c3032dd8606bfc707e67a3315f5cc6b1496b1b186739

Then, using Bob’s private key, the session key of AES encryption is decrypted from the encrypted session key part with RSA decryption. The result session key of AES

0eefa7cec42816ab0ba07e104701eae90eefa7cec42816ab0ba07e104701eae9

Then, using this AES key, encrypted message part is decrypted with AES decryption. The code segment is shown in Figure 5.19 and the result is:

```
77510099b4f1fd7d5e892708dbd26cc3b16825609d5ac508479a28acfe6e8f88
67c732ed0ac3c4c88d8eae8633e0df2af505a4346f57bafaf61e6f4353c97a92
f170647eb235d6f162331bb0080bc337f0d8f3bba9da9b09bac0861fc62f5e18
d3791c9180bbc564fa603bc4273ae1f4f77f36e10b524cc64a5688bc01f4f327
48656c6c6f20576f726c6421.
```

Result includes original message and signed digest part. By separating the message and signed digest part, we get the message

```
48656c6c6f20576f726c6421
```

in hexadecimal coded form constructed from the ASCII values of each characters of our message "Hello World!"

We then check the integrity and the authenticity of the message. To do this, we take the message and find its hash value using SHA-1 algorithm and we decrypt it using Alice's private key. If the results are equal, then we verify our sender and message. If not, message is changed and there is an attacker between sender and receiver that act as impersonator or there is a transmission error. In our case Sha-1 out and result of RSA decryption are both equal to

```
2ef7bde608ce5404e97d5f042f95f89f1c232871
```

We successfully decrypt and verify our message and sender

```
//create 128bit AES Block cipher
CFBBlockCipher cipher2= new CFBBlockCipher(new AESEngine(),128);
//set key and initial vector
cipher2.init(false,       new       ParametersWithIV(       new
KeyParameter(aeskey),initialvector ));
//decode into ASCII coded hexadecimal
byte[] aesinput=Hex.decode(encryptedMessagePart);
//create result array
byte[] aesoutput=new byte[aesinput.length];

byte[]  in = new byte[16];
byte[]  out = new byte[16];
//feed Block cipher
for(int i=0;i<aesinput.length;i=i+16){
     for(int j=0;j<16;j++){
          if((i+j)<aesinput.length){
               in[j]=aesinput[i+j];
           }
     }
     //process current block write result in out array
     cipher2.processBlock(in, 0, out, 0);
     System.out.println("AES   Decpryption   block   "+i+":"+new
     String(Hex.encode(out)));
     //combine result in an aesoutput array
     for(int j=0;j<16;j++){
          if((i+j)<aesinput.length){
               aesoutput[i+j]=out[j];
           }
      }
}
```

Figure 5.19 AES CFB Mode Block Cipher Decryption code

Chapter 6
Conclusions

In this project we aimed to implement a security structure in limited wireless device application domain while sending SMS. We chose to implement simplified version of PGP security application. We simplified PGP by not using level of authenticity and certificate authority structure. According to our knowledge, there was no public implementation of PGP for mobile devices, So we choose to implement PGP in CLDC restrictions.

Throughout this thesis, java application is implemented by using Java 2 Micro Edition in order to demonstrate the PGP system with java on Java wireless toolkit. Algorithms are implemented with CDLC configurations. Virtual phones are created at different hosts which are sender and receiver. Record Stores are used by the implemented application. Because, we wanted to stick to the standards in mobility application, we used J2ME. We choose to implement our application on the basis of Bouncy Castle Package, due to its ease of use and the integrity of its classes.

We implemented our simplified version of PGP successfully by using SUN Wireless Toolkit on Mobile Phone Emulators. We tested our application on emulators and implemented RSA key generation, key exchange, encryption and decryption processes and storage of keys. Overall test was successful. We could not test on real devices because of Signing Midlet Process is too long and costly.

As a future work, integration of our application to real devices could be done. As a second one, full implementation of key ring with trust issues and certificates authority structure can be done.

References

[1] Stallings, W., *Cryptography and Network Security Principles and Practices, Prentice Hall Publishing*, 4th Edition, 2005.

[2] Mao, W., *Modern Cryptography: Theory and Practice*, Prentice Hall Publishing, 1st Edition, 2003.

[3] Fmenezes, A. J., Oorschot, P. C., Vanstone, S. A., *Handbook of Applied Cryptography*, CRC Press, Boca Raton, 1997.

[4] Shamir, A., Tromer, E., *On the coast of Factoring RSA-1024*, CryptoBytes, 2003.

[5] AES -Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Aes.

[6] Friedel, D., Potts, A., *Java Programming Language Handbook,* The Coriolis Group, Scottsdale, 1996.

[7] Riggs, R., Taivalsaari, A., *Programming with Wireless Devices with J2ME,* Addison Wesley, 2003.

[8] Li, S., Knudsen, J., *Beginnig J2ME:From Novice to Professional,* A Press, 2005.