IMPROVEMENT OF KEY VALIDATION EFFICIENCY IN A SECURE MOBILE MESSAGING APPLICATION USING SMS-BASED PUBLIC KEY CRYPTOGRAPHY (SB-PKC)

BY

LUQMAN OLADIPO DUROJAIYE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA

JUNE, 2015.

IMPROVEMENT OF KEY VALIDATION EFFICIENCY IN A SECURE MOBILE MESSAGING APPLICATION USING SMS-BASED PUBLIC KEY CRYPTOGRAPHY (SB-PKC)

BY

Luqman Oladipo DUROJAIYE, B. Eng. (UNAD) 2007

MSc/ENG/8546/2009-2010

durojaiyeluqman@yahoo.com

A THESIS SUBMITTED TO THE SCHOOL OF POSTGRADUATE STUDIES, AHMADU BELLO UNIVERSITY, ZARIA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF MASTER OF SCIENCE (MSc) DEGREE IN ELECTRICAL ENGINEERING.

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA.

JUNE, 2015.

DECLARATION

I, **Luqman Oladipo DUROJAIYE** declare that the work in this dissertation entitled "**Improvement of Key Validation Efficiency in a Secure Mobile Messaging Application Using SMS-Based Public Key Cryptography (SB-PKC)**" has been carried out by me in the Department of Electrical and Computer Engineering, Ahmadu Bello University, Zaria. The information derived from the literature has been duly acknowledged in the text and a list of references provided. No part of this dissertation was previously presented for another certificate at this or any other Institution.

Luqman, Oladipo DUROJAIYE ______________________ ______________________

Signature Date

CERTIFICATION

This thesis entitled "**IMPROVEMENT OF KEY VALIDATION EFFICIENCY IN A SECURE MOBILE MESSAGING APPLICATION USING SMS-BASED PUBLIC KEY CRYPTOGRAPHY (SB-PKC)"** by **Luqman Oladipo DUROJAIYE** meets the regulations governing the award of the degree of Master of Science (M.Sc.) in Electrical Engineering of Ahmadu Bello University, and is approved for its contribution to knowledge and literatrary presentation.

Chairman, Supervisory Committee	Signature	Date
(Dr. D.D. Dajab)	______________________	________________
Member, Supervisory Committee	Signature	Date
(Prof. M. B. Mu'azu)	______________________	________________
Head of Department	Signature	Date
(Prof. M. B. Mu'azu)	______________________	________________
Dean, School of Postgraduate Studies	Signature	Date
(Prof. A. Z. Hassan)	______________________	________________

DEDICATION

This work is dedicated to my sister and “mum”, Mrs. Sherifah Adepoju-Durojaiye, for setting me on the path of academic excellence.

ACKNOWLEDGEMENTS

All praises and thanks are due to Almighty Allah, the One Who created all things and decreed all events. I bear witness that, to Him belong all genuine worship and praises.

I owe a great deal to my supervisors; Dr. D. D. Dajab and Prof. M. B. Mu'azu, for taking pain to patiently read through my work and made critical contributions to improve its outputs.

I appreciate the contributions of the entire staff of the Electrical and Computer Engineering Department, Ahmadu Bello University (ABU) Zaria. Special thanks to Prof. B. G. Bajoga and Dr. S. M. Sani for their constructive criticisms and valuable contributions. Same goes to Dr. B. Y. Jimoh, Dr. J. Y. Oricha, Dr. H. Sikiru, Dr. S. G. Garba, Dr. A. D. Usman, Dr. A. O. Akinsanmi, Dr. A. M. S. Tekanyi, Dr. A. Kabir (Abu Bilal), Eng. S. Eleruja, Eng. M. Abdullahi (Minister), Eng. A. Taiwo, Eng. O. Josiah, Eng. E. Okafor and others.

I also appreciate the encouragements of colleagues like Engr. M. Shehu, Engr. A. Umar, Engr. K. Oluyale, Eng. W. Oluyombo and the rest. I will also never forget the fantastic contributions of the members of MSc Control Class ("Monastery"). I am indeed grateful to you all for your invaluable inputs.

This acknowledgement would not be complete without mentioning the role of individuals who played significant role in ensuring that I have a place in computer application development. In this regard, I would like to appreciate, Eng. A. O. Semiu (VodaCom, Lagos), Malam A. Y. Kabir (Iya Abubakar Institute, ABU, Zaria.), R. N. Salaudeen (Tsaboin Tech World, Lagos). My profound appreciation also goes to fellow programmers/developers on programming collaboration platforms like Stackoverflow and JavaRanch for their contributions and suggestions towards making the completion of this work a reality.

Finally, I also want to appreciate the assistance of members of my family; my mother, Mrs. Funmilayo Durojaiye, my siblings Mrs. Sherifah Adepoju-Durojaiye, Kazeem Durojaiye, Rizqah Durojaiye and AbdulAfeez Durojaiye. Special thanks to my wife, Mrs. Bukola Durojaiye-Oguntomole. To my lovely kids, AbdulWahhab, Maryam and Ibrahim, I say very big thank you.

ABSTRACT

Short Message Service (SMS) is one of the most used mobile data services. It is utilized in wide range of applications including information delivery, remote operation of appliances, online purchases/banking transactions. However, some of these applications require that the underlying messaging protocol must guarantee secure communication; which traditional SMS cannot provide. State of the art approach to adding security to SMS messaging involves the use of asymmetric cryptography, supported by Public Key Infrastructure (PKI) for addressing public key verification/validation issues inherent in asymmetric-based cryptosystems. Despite PKI's good features, its key validation mechanism has potential for delay and high resource requirements; making it not appropriate for use in resource-constrained environments like mobile phones. Thus, this work aims to develop an asymmetric-based secure mobile messaging application (dubbed "ProSMS") with improved efficiency during key validation when compared to PKI. A hybrid of symmetric (Twofish), asymmetric (RSA) and Hash function (SHA256) ciphers is implemented to add secure communication requirements to SMS messaging using J2ME programming. These security features were added to traditional SMS messaging with minimal overheads as indicated by the very low (< 1s) runtime requirements of the cryptographic processes. A new approach to public key verification/validation; dubbed SMS-Based Public Key Cryptography (SB-PKC) and earlier PKI were also implemented in the developed application in order to compare their key validation requirements. A major step used to differentiate SB-PKC from PKI is the elimination of the need for frequent request for keys' validity status information by users. The efficiency during key validation in SB-PKC was shown to improve over what is obtainable in PKI; (i) 40ms latency in SB-PKC as against 5000ms in PKI, the significance of which is that usability is improved as higher latency in wireless communication is known to hamper usability. (ii) 143 bytes and 472 bytes of request and response data sizes respectively are required in PKI as against 0 and 202 bytes data required in SB-PKC, the significance of which is that bandwidth-use optimization is higher in SB-PKC than in PKI. The test device used to arrive at these results is Tecno T9 mobile device.

TABLE OF CONTENTS

CHAPTER THREE: MATERIAL AND METHODS

CHAPTER FOUR: RESULTS AND DISCUSSIONS

CHAPTER FIVE: CONCLUSION AND RECOMMENDATIONS

LIST OF FIGURES

LIST OF PLATES

LIST OF TABLES

LIST OF ABBREVIATIONS

Acronym	Definition
API	Application Programming Interface
CA	Certificate Authority
CB-PKC	Certificate-Based Public Key Cryptography
IAIK	Institute for Applied Information Processing and Communication
ID-PKC	Identity-Based Public Key Cryptography
J2ME	Java Platform Micro Edition
J2SE	Java Platform Standard Edition
KGC	Key Generation Centre
PGP	Pretty Good Privacy
PKG	Public Key Generator
PKA	Public Key Authenticator
PKI	Public Key Infrastructure
PTMF	Phaos Technology Micro Foundation
RSA	Rivest Shamir Aldeman
SA	System Authority
SGC-PKC	Self-Generated Certificate- Public Key Cryptography
TTP	Trusted Third Party
WoT	Web of Trust

CHAPTER ONE

INTRODUCTION

1.1 Background

Distributed networks like the internet and the telephone are indispensible tools in today's world. They play important roles in connecting peoples from near and far distant places. However, their use posses a security threat because they allow considerable amount of anonymity, the consequence of which is that peoples' identities can be stolen and the data passing through them can be maliciously used (Noroozi *et al.*, 2014).

A very popular and widely deployed distributed network is the Global Service for Mobile Communications (GSM). With its inception in Europe in 1991, GSM has seen widespread adoption all over the globe. GSM has revolutionized mobile communication, becoming the default global standard for mobile communications with over 90% market share, operating in over 210 countries and territories and connecting over 6.8 billion people as at first quarter of 2014 (Isabona, 2014; Azubuike and Obiefuna 2014). Some of GSM services include voice call, data transmission including internet access, Short Message Service (SMS) etc.

SMS, the most widely used mobile data service is a mechanism for delivery of short messages over telecommunication networks. It is a global wireless service that enables the transmission of alphanumeric messages between mobile subscribers, originally designed as part of GSM, but now available on a wide range of network standards. SMS messaging is accomplished via a store-and-forward mechanism (Toorani and Beheshti, 2008). Figure 1.1 is a block diagram of main network elements of the GSM architecture illustrating the flow of SMS from one user to another.

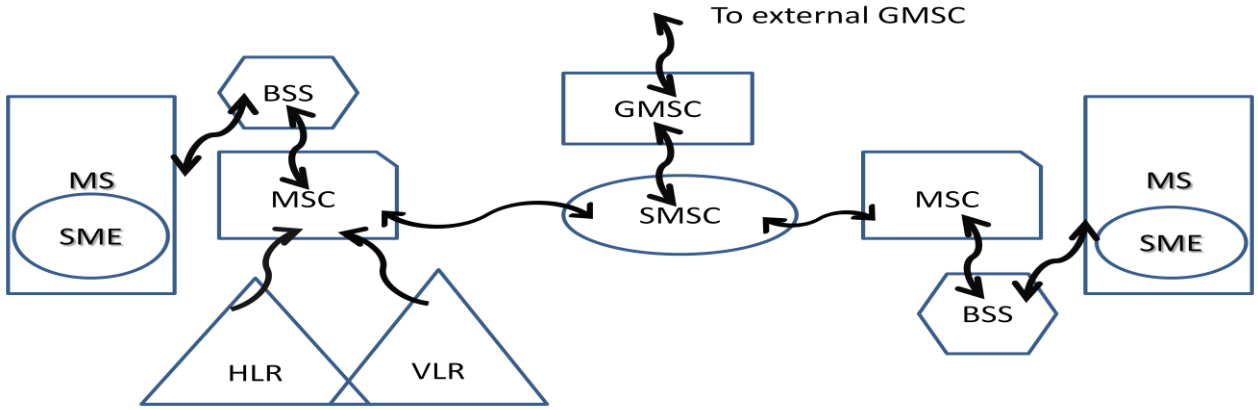


Figure 1.1: Basic Network Elements of SMS Messaging (Gupta, n.d.)

The SME (Short Message Entity) is the messaging software embedded in a mobile station (MS) that handles the sending and receiving of SMS. From a MS, an SMS is sent (and received) via a radio link communication through the Base Station System (BSS) whose basic functionality is transmission of voice and data traffic signals to and from a MS. This is the only part where encryption is optionally applied using very weak A5/1 and A5/2 cryptographic ciphers (Saxena and Payal, 2011; Toorani and Beheshti, 2008). The message is then routed to an appropriate Mobile Switching Centre MSC; the entity responsible for switching connections between mobile stations or between mobile stations and the fixed network. Thereafter, the SMS is further routed to the Short Message Center SMC or Short Message Service Center SMSC which is the entity responsible for the store-and-forward feature of SMS messaging. The SMS is now forwarded to an appropriate MSC that currently serves the recipient MS using routing information (about the current position of the receiving MS) from Home Location Register HLR. Depending on whether the message is intra-network or inter-networks, it may involve the Gateway MSC (GMSC). The GMSC is a mobile network's point of contact with other networks. The visitor location register (VLR), like the HLR, is a database that contains subscribers' temporary information. The MSC uses this information to service visiting subscribers.

SMS' simplicity, ease of use and a host of other factors have encouraged its growth among users (Pujitha and Mallu, 2013). The total number of SMS exchanged worldwide globally between the year 2007 and 2010 tripled, from an estimated 1.8 trillion to 6.1 trillion; approximately 200 000 SMS messages are sent every second (Rao, 2011; Statista 2015). And by first quarter of 2014, about 30 billion SMS messages were sent in UK alone (Statista, 2015). These SMS exchanges are used in a wide variety of applications across many areas of human endeavors; Governance, education, enlightenment etc.

The increased use of SMS has also witnessed its use in the exchange of sensitive information that requires some level of security that is not available in SMS messaging. A good example is the use of SMS for online purchases and banking transactions. A very obvious limitation here is that the communicated SMS (transaction) is not confidential; (1) Network providers have access to copies of all SMS transmitted, (2) intruders can read these SMS when they are in possession (as a result of theft, lost of mobile device etc.) of a user's mobile phone. SMS has also been found to be very useful for sending control information. A good example is the use of SMS for communicating control (sensitive) signals for operating equipments as used in Khiyal *et al*., (2009) for remotely operating appliances. The limitation here is that these appliances need assurance that the received SMS is sent from the desired device (authentication) and that the received SMS has not been changed in transit (integrity). Suresh *et al*., (2014) documents popular instances of compromise of SMS security. Traditional SMS messaging is not originally designed for exchanging sensitive information like these, thus it lacks secure communication requirements. Therefore, if SMS must be used for exchanging sensitive information, secure communication features must be added. One of the ways that this can been done is via cryptography.

Cryptography is one of the most popular tools used today for ensuring data security on potentially insecure platforms like the GSM network and its services like SMS. Cryptography is applied on these networks to enable secure communication requirement of; (i) **confidentiality** (to ensure the exchanged information is not divulged to unauthorized entities, (ii) **integrity** (to ensure the exchanged information is not altered in transit), (iii) **authentication** (to ensure that the communicating entities are "true", so that impersonation is not possible and (iv) **non-repudiation** (to ensure that a sending entity cannot falsely deny a message that originates from it (Schneier, 1996).

Cryptography comes in two flavors; symmetric and asymmetric cryptography (Public Key Cryptography or PKC). While symmetric cryptography can provide confidentiality and integrity functionalities, asymmetric cryptography can provide all the four secure communication requirements (Lisonek and Drahansky 2008). However, PKC suffers a major drawback; Identity binding; how to prove that the public key component of a PKC key pair is genuine. This has been an open problem since the concept of asymmetric cryptography was developed (Gutmann 2013; Lee *et al*., 2011; Toorani 2011; Yuan, 2004; Stallings, 2004; Schneier, 1996).

Public Key Verification (PKV) schemes are used as support systems to help prove the genuineness or otherwise of users' public keys in a PKC environment. Over the years, a number of PKV schemes have emerged. Examples include Public Key Infrastructure (PKI), Web of Trust (WoT), Identity-Based Public Key Cryptography (ID-PKC), etc.

The most popular PKV scheme in use is the PKI (Wiesner, 2013). Despite the advantages of PKI over the other PKV schemes, PKI has the following shortcomings;

1. Key Validation is delay-ridden (high latency) and thus affects usability (Stark *et al*., 2012).

High latency is inherent in PKI's key validation process because, it utilizes *chatty* protocol, where Relying Parties (RPs) or users have to make request for a key's validity status to a Trusted Third Party (TTP) Server and wait for a response. RPs or users are required to *regularly check, on a real time basis,* if the status of a key (belonging to another user that secure communication is to be established with) has changed at a particular time before the key can be used. The fact that this check is done on a real time basis, implies there is potential for delays (from the CA servers) or outright denial of service during peak hours or when the servers have technical issues (Gutmann, 2013). This is further compounded with network connectivity issues of the client device as at the time of the request. Mobile phone network connectivity is particularly susceptible to large irregularities and fluctuations (Kumar *et al*., 2013).

2. It has high overhead (recurrent internet roundtrips for key validation (which also implies recurrent battery, CPU, memory etc. use) and significant storage space requirements (Medani *et al*., 2011)

PKI's chatty protocol approach to validation-check implies a regular use of clients device's resources like battery, CPU, memory, bandwidth etc. for this operation; these resources are comparatively smaller on mobile devices. Furthermore, their use in this way constitutes a waste, when the outcome of any two consecutive instances of revocation-check (or validation-check) is the same.

Thus, this work implemented a PKC-based secure mobile messaging application. An improved approach to PKV is also implemented to improve the efficiency during key validation by minimizing the latency and high overhead requirements.

1.2 Problem Statement

Recent works on SMS vulnerabilities have been in the direction of applying asymmetric (PKC) cryptographic ciphers as they can afford all the four secure communication requirements. However, identity binding problem inherent in PKC-based solution has not been properly addressed in the context of the resource-constrained nature of the typical mobile phone environment. Many works that attempt addressing the identity binding problem in this area of research (SMS/mobile environment) have simply "dumped" the PKI-based solution as used in the wired environment. PKI-based solutions for SMS are not optimized for mobile environments; the use of digital certificates requires large storage space, long (and recurrent) time processing (during certificate/key revocation-check) and substantial (and recurrent) bandwidth/internet connectivity requirements, and consequently inefficient use of mobile phones' limited resources (Medani *et al*., 2011; Sharma, 2009). In the light of these, there is need for an alternative PKV scheme whose latency and other overheads for key validation are lower than in PKI. The scheme in addition to requiring lower resources for key validation should not sacrifice any of the good features of PKI. This is what this work is set out to achieve.

1.3 Aim and Objectives

The aim of this work is to develop a PKC-based secure SMS messaging mobile application that utilizes a PKV scheme, which in comparison with PKI has a comparatively lower latency and overhead costs during key validation and yet does not sacrifice any of the good features of PKI.

To achieve this aim, the following objectives have been set;

I. Develop J2ME codes to create a PKC-based mobile SMS messaging application (ProSMS) with all secure communication requirements of confidentiality, integrity, authentication and non-repudiation.

II. Propose (and implement) an improved PKV scheme to be used for identity binding in the developed mobile application (ProSMS) and

III. Compare the resource-cost requirements during key validation in the improved PKV scheme with that of the PKI.

1.4 Methodology

The following methodology was adopted in carrying out this research:

1. Selection of an appropriate cryptosystem design based on which type can provide all the four secure communication requirements.
2. Selection of appropriate ciphers/algorithms based on (1) above and other pertinent parameters like security etc.
3. Development of a communication protocol based on (2) that enables secure SMS communication on mobile phones.
4. Development of the codes (SMS app) to implement the communication protocol in (3). Take run time requirements of the cryptographic processes to evaluate the added overhead cost.
5. Propose the SB-PKC PKV Scheme detailing its actors and their relationships.
6. Implementation of pertinents parts of the SB-PKC by writing programming codes. Take public key validation requirements data for the SB-PKC scheme.
7. Development of codes to implement public key validation using PKI (via OCSP). Take public key validation requirements data for the PKI scheme.
8. Testing (simulation) of the application’s functionalities.
9. Testing (on real device) of the application’s functionalities.
10. Validate by comparing public key validation requirements of SB-PKC and PKI.

1.5 Significant Contributions

The significant contributions derivable from this work are as follows:

1) Development of a hybrid cryptosystem for secure SMS messaging that utilizes session keys, in the manner used in the popular secure e-mail clients like PGP; where encryption key (symmetric) is only used once per message, the asymmetric key is used for secure key exchange purpose and users themselves are responsible for the generation of all their keys.
2) Experimental measurement of the average latency that mobile phone users experience during key validation processs under PKI scheme. It was found that, it takes an average of 5000ms to obtain key validation information.
3) Proposal and Development of a PKV scheme with comparatively (over PKI) lower latency and overhead requirements during key validation. The latency experienced during public key validation in the improved scheme is 40ms as against 5000ms in PKI. Furthermore, the scheme reduced to the minimum (0), number of internet roundtrips requirements, thus conserving bandwidth and other associated resources like battery consumption, etc.

1.6 Dissertation Organization

The general introduction has been presented in Chapter One. The remaining chapters are structured as follows: A detailed review of the relevant literature and pertinent fundamental concepts are presented in Chapter Two. Chapter Three discusses the methodology adopted in achieving the set objectives. The results obtained were analyzed and discussed in Chapter Four. Chapter Five presents the conclusion and recommendations for further work. Quoted references and Appendices are also provided at the end of the dissertation.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

This chapter is divided into two parts. In the first part, fundamental concepts relevant to the dissertation are discussed while in the second part, similar works are reviewed.

2.2 Review of Fundamental Concepts

In this section, fundamental concepts around this work are presented, including definition of pertinent terms and other matters related to the subject matter.

2.2.1 Cryptography

Cryptography, an ancient art (Kessler, 2014), is the science of secret writing used to provide security to data (Schneier, 1996); be it data at rest or data in transit. It involves the use of some mathematical functions called cryptographic algorithms or ciphers. The cipher for example, is responsible for the transformation of the data (message) into unreadable equivalent (or ciphertext) and vice versa. Often, users have to apply "keys" to these ciphers to achieve specific action; encryption, decryption, digital signature etc. Depending on the nature of keys used, ciphers can be symmetric or asymmetric.

2.2.1.1 *Symmetric Cryptography*

The first sets of cryptographic ciphers are symmetric with examples like Twofish, AES, RC6 etc. They use one and the same key (K) for encryption and decryption and are mostly used for achieving *confidentiality*; one of the four secure communication requirements. Equations 2.1 and 2.2 show how a symmetric cipher is used to achieve confidentiality of the communicated information. A

$$K(M) = C \tag{2.1}$$

$$K(C) = M \tag{2.2}$$

sender *encrypts* (changing of plaintext or original message M, to ciphertext C) using symmetric key K, and sends C to the receiver. The receiver *decrypts* (changing the received ciphertext C, back to plaintext M) using the same key. The ciphertext C is an unintelligible string of characters; such that if it is intercepted, the attacker cannot make any sense from it. Symmetric ciphers are fast and highly efficient during encryption and decryption. They work and are secure on the assumption that communicating parties have secure mean(s) of exchanging their shared key K. One popular way to do this is, for users to meet face to face to exchange K (Schneier, 1996). It is however, obvious this approach does not scale well for use on large networks like telecommunication networks where this research intends to apply cryptography. In this regard, symmetric cryptography is said to suffer from “safe distribution of keys”. To address this problem, Diffie and Hellman (1976) proposed asymmetric cryptography in 1976 (Schneier, 1996).

2.2.1.2 *Asymmetric Cryptography*

Asymmetric or PKC is a type of cryptography in which two (“public key” and “private key”) different but mathematically related keys are used (Diffie and Hellman, 1976). The public key is referred to as “public” because it will be made available to the general public; anyone with a user’s public key can encrypt message with that public key (Sharma, 2009). The private key is referred to as “private” to imply only its owner would be privy to it. The private key is used for decryption. For every key pair in a PKC, it is impossible to obtain the private key from the public key. PKC provides solution to the safe key distribution problem inherent in symmetric cryptosystems; this time, keys don’t have to be distributed in secret before secure communication can take place; because what is exchanged is the public key (user C or an attacker can also have it); used for encryption only and not for decryption (Schneier, 1996). Each user in a PKC has own pair of private or secret key (Sk) and public key (Pk). So that users A and B have SkA/PkA and SkB/PkB pair of keys respectively (Voarse, 2009). While equation 2.3 shows how user B can encrypt

$$PkA(M) = C \tag{2.3}$$

$$SkA(C) = M \tag{2.4}$$

(for confidentiality) message M meant for user A, equation 2.4 shows how the resulting ciphertext C, is converted back to the original message M during decryption by user A in a PKC-based cryptosystem. Examples of PKC ciphers include RSA (Rivest, Shamir and Aldeman), ElGamal, DH (Diffie-Hellman), etc (Schneier, 1996). The RSA cipher is used in this work. The following mathematical computation is involved in RSA key generation, RSA encryption and Decryption (Lai 2008; Ireland 2010)

a. RSA Key Generation

1. Randomly generate two large primes, p and q, of approximately equal size.
2. Calculate:

$$n = p * q \tag{2.5}$$

and

$$\varphi = (p-1)(q-1) \tag{2.6}$$

Where n is of the required bit length (e.g. 1024 bits in this work). n is referred to as the modulus.

3. Randomly select an integer e, such that $1 < e < \varphi$ and HCF $(e, \varphi) = 1$. e is referred to as the public exponent or encryption exponent.
4. Calculate the secret exponent d, $1 < d < \varphi$, such that $e*d \equiv 1 \pmod{\varphi}$ or $d \equiv e^{-1} 1 \pmod{\varphi}$. d is known as the secret exponent or decryption exponent.
5. The RSA public key is (n, e) and the RSA private key is (d, p, q).

b. RSA Encryption

To use RSA for encryption, a sender (user A) does the following:

1. Obtains the recipient B's public key (n, e).
2. Represents the plaintext message as a positive integer m, 1 < m < n.
3. Computes the ciphertext c;

$$c = m^e \bmod n \qquad (2.7)$$

4. Sends the ciphertext c to user B.

c. RSA Decryption

At the receiver’s end, recipient B does the following:

1. Uses his private key (n, d) to compute m;

$$m = c^d \bmod n \qquad (2.8)$$

2. Extracts the plaintext from the message representative m.

Apart from message encryption (for confidentiality), PKC algorithms can also be used for other purposes; secure exchange of symmetric key and achieving non-repudiation and authentication secure communication requirements. The following are briefs on these two concepts.

1. Secure Exchange of Symmetric Key

Symmetric ciphers are more efficient than asymmetric ciphers; most of them are many times faster (Schneier, 1996). Thus, in some scenarios, like those involving bulk/large data encryption, asymmetric ciphers are not used for encryption; rather they are used to securely exchange symmetric cipher keys (to address the safe key distribution problem of symmetric ciphers) which are used for the actual encryption (Voarse, 2009). Equations (2.3) and (2.4) still represents how this is done except that what is encrypted is key K and not message M. Example of PKC algorithm that can be used this way are RSA (Rivest Shamir Aldeman), ElGamal etc. (Schneier, 1996). Other means include the use of special PKC ciphers like Diffie-Hellman (DH) etc. which cannot only securely exchange symmetric keys, but can protect against Man in the Middle (MITM) attack. MITM attack involves a privileged attacker, who cannot only intercept the key (that is being exchanged), but can replace it with another one (Schneier, 1996). Such that when the sender (say user B) is encrypting a message meant for "user A", he is in effect encrypting a message that can be decrypted by the attacker. The following computations show how two communicating partners can securely exchange symmetric keys using DH (Kessler, 2014).

User A	User B
1. Choose a secret (private) value $X_A = 2$	Choose $X_B = 3$
2. Generate 3 and 7. Then sends them to B	Receives 3 and 7

Where the integer "7" in an ideal or "real life "case is a large random value and "3" is primitive modulus of "7".

3. Calculate (public key) $Y_A = 3^2 \bmod 7 = 2$	Calculate $Y_B = 3^3 \bmod 7 = 6$
4. Sends public key = 2 to B	Sends public key = 6 to A
5. Computes symmetric $K_A = 6^2 \bmod 7 = 1$	$K_B = 2^3 \bmod 7 = 1$

The final output K_A is the symmetric key that can now be used for data encryption. This simple illustration shows that DH can be used to arrive at the same value of K_A (i.e at step 5) at both ends even though different public and private key values are used by the communicating parties. All the exchanges are done without encryption, yet the final value is protected from eavesdropping, even if the exchanges are intercepted by an adversary.

2. Non-Repudiation and Authentication

Asymmetric ciphers are also used for achieving authentication and non-repudiation properties of secure communication via signature creation and verification (Voarse, 2009; Schneier, 1996).

Equation 2.5 shows how PKC is used to generate a signature created by user B. The signature S is created by using B’s private key SkB, known to B alone, so that user A (or any other user, who is in possession of a copy of B’s public key PkB) can verify S using B’s public key PkB (Voarse, 2009).

$$SkB(M) = S \tag{2.9}$$

$$PkB(S) = M \tag{2.10}$$

Non-repudiation of a sender is implied as follows; the sender (B) cannot repudiate any transaction accompanied by its signature; because if B’s signature is verified using B’s public key in others’ possession, the transaction could not have been “authorized” or signed by anyone other than B, since only B is in possession of the private key used for creating the signature in the first place. Authentication is implied as follows; the sender is “true” (authenticated) if the signature created by its private key is verifiable using its public key in others’ possession.

The following mathematical computation is involved in RSA digital signature creation and verification (Lai (2008); Ireland (2010)).

(i) RSA Digital signature

To use RSA for digital signature, a sender (user A) does the following:

1. Creates a message digest of the information to be sent.
2. Represents this digest as an integer m between 1 and n-1.
3. Uses her private key (n, d) to compute the signature s;

$$s = m\^{}\, d \bmod n \tag{2.11}$$

4. Sends this signature s to the recipient, B.

(ii) RSA Signature verification

At the receiver end, recipient B does the following:

1. Uses sender A's public key (n, e) to compute integer v;

$$v = s^e \bmod n \tag{2.12}$$

2. Extracts the message digest from this integer.
3. Independently computes the message digest of the information that has been signed.
4. If both message digests are identical, the signature is valid.

Despite all these advantages offered by PKC over symmetric ciphers, it has a major drawback too; "Identity binding" (Yuan, 2004; Toorani, 2011; Gutmann, 2013). That is, how to prove that a particular public key does belong to a "claimed identity". PKC can prove that a key does belong to the "user" (authentication) but cannot prove whether the identity claimed (i.e. identity attached to the public key) by the "user" is true or not. Thus, the many approaches to PKC issues revolve around identity binding: building "trust" for public keys (Lee *et al*., 2011) via PKV schemes.

2.2.2 PKV Schemes

To address identity binding in PKC-based systems, quite a number of PKV schemes have been proposed over the years. The schemes can be categorized into two broad approaches; *explicit* or *implicit schemes* (Gentry, 2003).

2.2.2.1 *Explicit Schemes*

Explicit schemes are PKC management approaches that employ the use of an external object like digital certificates for the purpose of binding a user's public key and his identity (Gentry, 2003). In explicit schemes, the genuineness of a public key on its own (and the identity it claimed to represent) cannot be established, unless with a proof other than the key. RPs rely on the "proof" to trust the public key and the identity it represents. Popular examples include Public Key Infrastructure (PKI) and Web of Trust or WoT (Weisner, 2013).

1. Public Key Infracstructure (PKI)

PKI was proposed in 1978 by Kohnfelder N. (Lee *et al*., 2011). It is the earliest and most popular PKV scheme (Wiesner, 2013). PKI is in popular use in secure information exchange on the internet, like those involving online purchases, banking transactions etc. Users' browsers are configured to confirm the true identity of HTTPS websites before confidential data is exchanged with them using PKI (Wiesner, 2013).

In a PKI environment (Figure 2.1), users generate their public and private key pairs themselves (Lee *et al*., 2011). Because users are in control of their private key (i.e. the private value is not known to anyone else other than the users), PKI is said to be free from *key escrow*; a situation where there is the potential for the security of a cryptosystem system to be compromised because users' private key (or decryption tool) is known to some other entities. After users have generated their key pairs, the public key component is registered with a Registration Authority (RA), which subsequently submits the outcome of the registration to a Trusted Third Party (TTP) entity called Certificate Authority (or Certification Authority or CA) (Balachandra *et al*., 2012; Ray and Biswas, 2011).

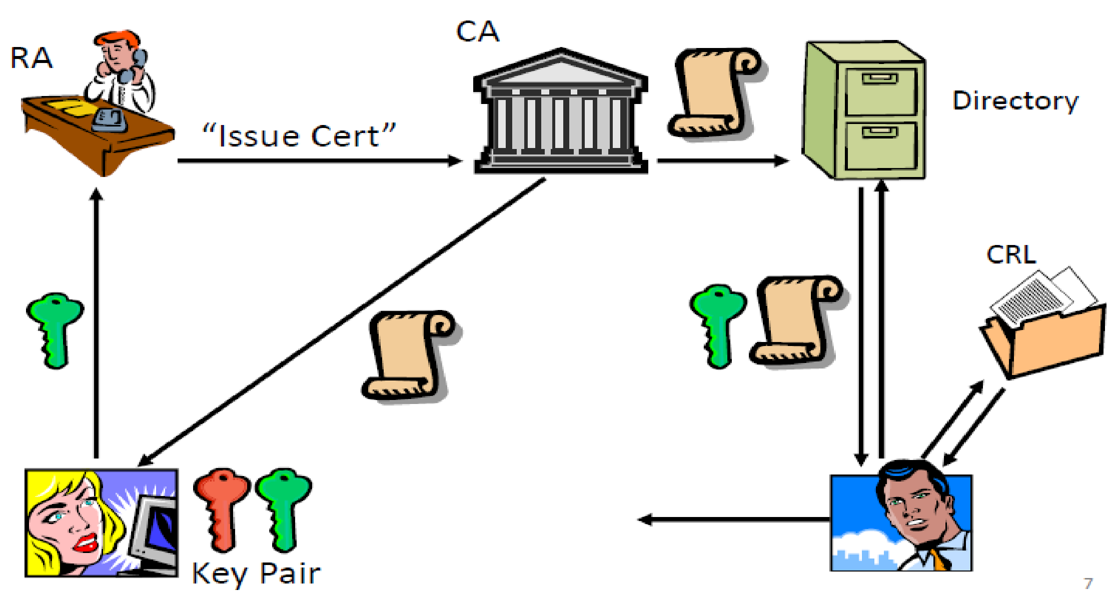


Figure 2.1: PKI Components (Toorani, 2011)

Ideally during a user's registration of his/her public key, a CA (or when RA does this on behalf of a CA) is expected to conduct necessary verification process about the person (ownership of a public key, Name (identity), contact info etc.) of the user before a certificate is produced on behalf of the user (Kessler, 2014; Ray and Biswas, 2011). A digital certificate (see Figure 2.2) is a CA's attestation that a particular public key that appears on a certificate has been verified to belong to the user on whose behalf the certificate is issued (Gentry, 2003). Certificates are archived in a CA's managed public directory so that RPs can access them when they need. It is the (CA's) attestation via a signed certificate, that other users rely upon, to trust or otherwise the genuineness of a particular user's public key and the associated name or identity (Stark *et al*., 2012).

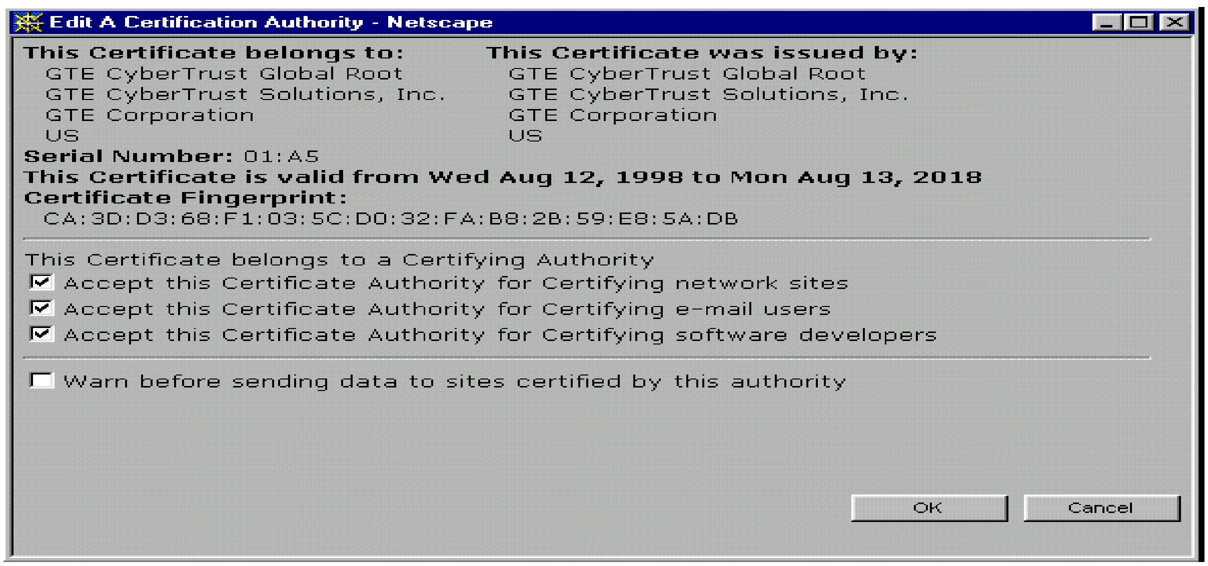


Figure 2.2: Digital Certificate (Kessler, 2014)

When users (or RPs) come across a user's public key (and its associated identity) for the first time, they need to perform two checks; verification-check and validation-check (Hou *et al*., 2012)

Verification-Check: RPs want to be sure that the identity claimed by owner of the key is true; a user is not disguising as another. To do this, they download the user's certificate (from the CA's directory) and verify that it is “truly” signed by the CA; by verifying the CA’s signature on the certificate using the CA’s public key (equation 2.10) in their possession. Verification-check is required once per public key/certificate life cycle to ensure that keys are genuine.

Validation-Check: RPs want to be sure that a previously held valid key (or confirmed to be genuine as in previous section) is still valid or has not been revoked. In this regard, verification of a key is not enough; RPs need more assurance using validation-check (Hou *et al.*, 2012). CAs implement revocation of public keys by including validity (see Figure 2.2) information field in certificates. A public key or its certificate is no longer valid or considered “revoked” after the validity period specified on the certificate. However, certificates/keys can also be revoked by the CA in the following instances, even before their expiration period (Gutmann 2013; Hou *et al.*, 2012; Topalovic *et al*., 2012; Stark *et al*., 2012; Ray and Biswas, 2013; Fujishiro *et al*., 2010; Gentry, 2003);

1. When the corresponding private key to the public key that the certificate certifies has been compromised.
2. When the certificate has been issued in error.
3. When the owner of the certificate does not have the privileges that the certificate conferred any longer.
4. When users no longer have access to their private key due to loss of the key itself or password to access it.

PKI, unlike some other PKV schemes, has a mechanism to handle key revocation checking. The checking of keys’ revocation status in PKI can be done via many approaches; Certificates Revocation List or CRL, Online Certificate Status Protocol or OCSP, Certificate revocation system or CRS, Certificate Revocation Tree or CRT, Certificate verification Scheme Multi-Hashing or

CVS-MH etc. However, CRL and OCSP are the most widely deployed (Wiesner, 2013; Stark *et al*., 2012; Fujishiro *et al*., 2010).

i. A CRL is a list of all certificates that have been revoked by a CA. The CA constantly updates the list. So to use the CRL for checking that a public key is not revoked, a user is required to download the last updated CRL. Afterwards the certificate to be validated is checked against the list. This approach is however, not popular because it consumes considerable amount of bandwidth, particularly for large CAs that has a high number of user's certificates (Stark *et al*., 2012; Gentry, 2003). This is so because, the list has the potential to grow so large that the process of checking a certificate against it requires considerable bandwidth. CRL size of large CAs can become as large as several megabytes of data (Mutton 2014, Topalovic *et al*., 2012). Other shortcomings of CRL include; checking time could be very high, inability to provide real time revocation status information etc.

ii. In OCSP (documented as RFC 2650) however, RPs check one (1) certificate's revocation status per request sent to OCSP severs or OCSP responders operated by the CA or a delegated Authority. These servers are responsible for fetching certificates' revocation status from a CA's CRL servers. With this feature, users' end devices do not have to download large size of data (containing list of revoked certificates) but make request and waits for OCSP server to output response information; the response information on per public key per request basis is what is downloaded by the user. OCSP response data size is small (about 400 – 2000 bytes) but the time it takes to output a response by an OCSP server can range from as low as 1s to as high as several seconds (Helme, 2014; Langley 2012). Thus, there is the potential for latency/delay that consequently affects usability (Stark *et al*, 2012). In anticipation of this reality, regulations stipulating performance benchmarks for OCSP response times requires CAs to ensure maximum response time of 10s (CA/Browser, 2013). The overall time it takes an RP to obtain key status time is also compounded by network roundtrip time from the RP device to the OCSP server. As a result, many users end up bypassing OCSP check because of its delay (Gutmann 2013; Topolavic *et al.*, 2012; Stark *et al.*, 2012). The effect of this latency during key revocation has become so problematic that some browsers like Chrome has seized to support OCSP checking since 2012 (Langley 2012; Stark 2012; Topolavic *et al.*, 2012; Wiesner, 2013). Worst still, users sometimes cannot get a certificates revocation status as the OCSP responders can be down (see Figure 2.3) at the time of the request, thereby leading to denial of service (Gutmann, 2013).

The inefficiency in PKI's public key status checking is as result of its approach which requires clients' devices to make regular request to revocation servers irrespective of whether a change (in state from a previous request) has occurred or not. In client-server environments, like the key status

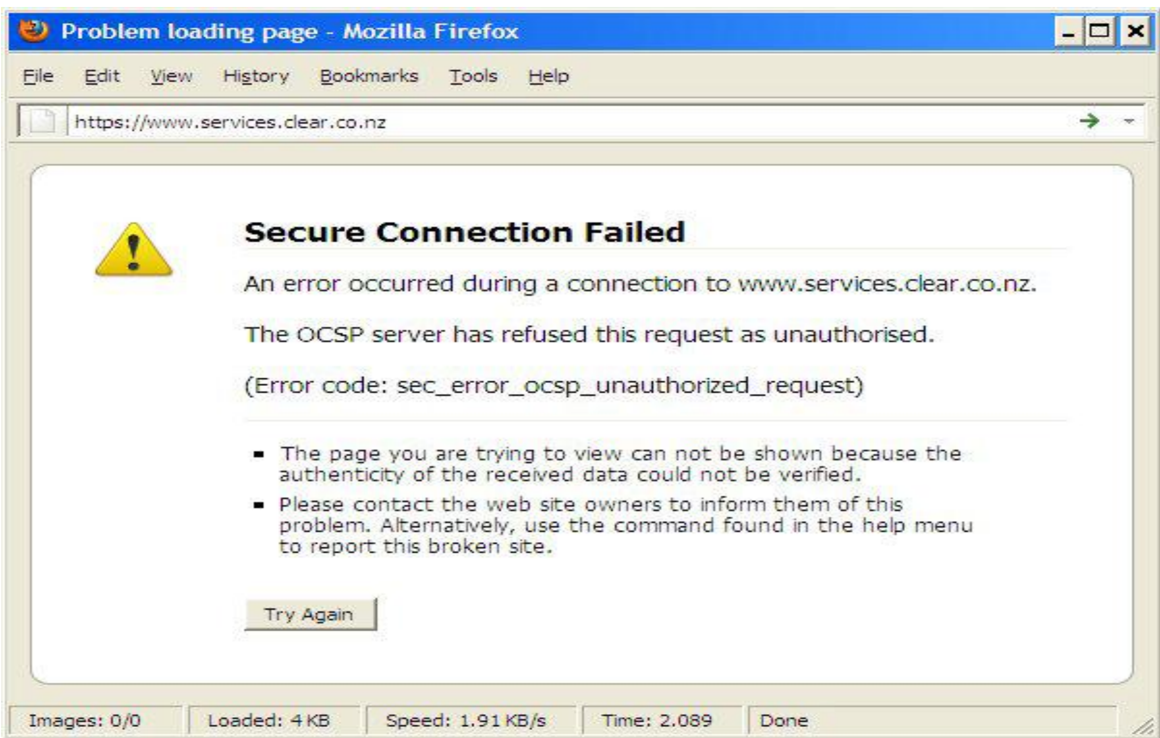


Figure 2.3: Denial of Service in OCSP (Guttman, 2013)

checking in PKI as detailed above, Yuan (2004) pointed out the disadvantage particularly for small-resource devices like mobile phones thus;

"Many enterprise mobile applications, such as an Instant Messaging client or a database monitoring client, need to be updated with real-time server status at all times. Since the HTTP protocol is ubiquitously supported in all J2ME devices, inexperienced developers sometimes program the device to initiate periodic HTTP connections to poll the server for its status. The polling frequency must be much faster than the expected server status-change frequency to keep the device updated. The constant polling results in a lot of redundant data. It is a waste of bandwidth, server resources, and time…"

A more efficient approach to this as suggested by Yuan (2004) is to delegate the server to "push" a notification to the client when a change has occurred via some form of control information (e.g. SMS, etc.), rather than the client making the request. This is the approach that the proposed SB-PKC adopts in order to address latency and other related shortcomings during public key status checking in PKI.

2. Web of Trust (WoT)

A Web of Trust is another PKV scheme that is similar to the PKI in that it uses digital certificates to bind users' identities to their public keys (Sharma, 2009). It is however, different because, while a PKI utilizes centralized trust system via a CA as detailed in section 2.2.2.1.1, WoT uses a decentralized trust system; rather than a central TTP entity certifying users' public keys, users themselves certify other users' keys (Wiesner, 2013; Alexander *et al.,* 2011). In WoT, users form a chain of trust from one user to another (Figure 2.4). If A and B are two users who trust each other, each one in turn vouches for the other's trustworthiness (genuineness of public key) by appending his/her signature on the other's certificate; the more signatures on a particular certificate, the more the trust it confers on the public key. In WoT, signing a certificate implies, a user (say A) is attesting that another user's (B) key that appears on B's certificate, truly belongs to as a CA does

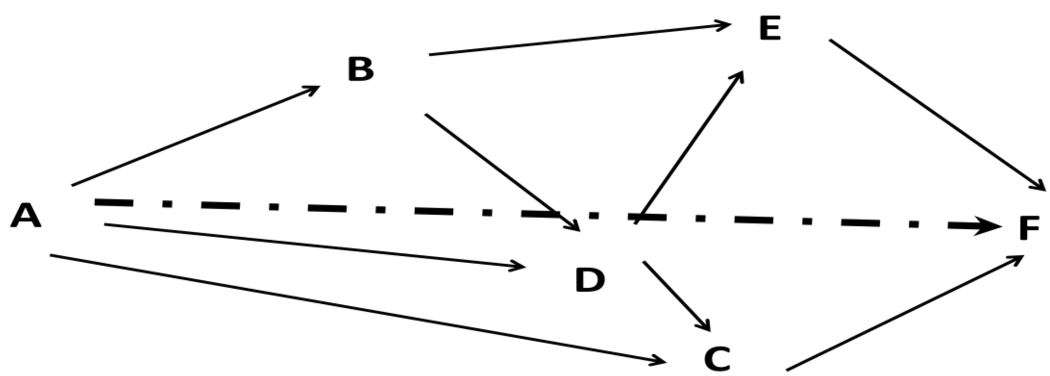


Figure 2.4: Web of Trust (Gutmann, 2013)

for its clients in PKI. If A need to communicate with F, but do not know F directly, but it does know B and C, who in turn know D and E, who know F. So A can establish a trust link to F through a number of indirect links. User (say B) who is known to D to be trustworthy but not to C, can establish communication with C by presenting its certificate, that is signed by D to C. This way, C can now trust B's public key by relying on D's attestation on B's certificate via his signature. C can verify D's signature (created by D's private key), using D's public key in its possession (equation 2.10).

Due to lack of a centralized identity certifying entity, users are made to collect others' public-keys and certificates on their own machines, thus they don't have to pay for certification service. The WoT model of trust is very flexible and gives users much freedom when it comes to certification (Kessler, 2014). Yet, WoT has not gained wide deployment as PKI (Wiesner, 2013). Also, revocation of compromised keys is a challenge in a PKI (Kessler, 2014).

2.2.2.2 *Implicit Schemes*

Implicit schemes are PKC management approaches that do not employ the use of any external object like digital certificates for the purpose of binding a user's public key and his identity (Lee *et al.*, 2011; Gentry, 2003). In these schemes, the keys are *implicitly* self-certifying; if they exist and can be used. Example implementations include Identity-Based Public Key Cryptography (ID-PKC), Certificateless Public Key Cryptography (CL-PKC), Certificate-Based Public Key Cryptography (CB-PKC) and Self-Certified Public Key Cryptography.

1. Identity-Based Public Key Cryptography (ID-PKC)

ID-PKC was proposed in 1984 by Shamir A. (Noroozi *et al.,* (2014); Toorani, 2011; Lee *et al*., 2011; Lu, 2011). It solves the need to use certificates to prove the ownership of a public key. It does this by generating users public keys from their "unique" identities. The assumption is that, a confirmation is not required as a public key in this scheme cannot be fraudulently claimed. The identity used could be any arbitrary string like IP Address, Phone Number etc. The corresponding private key is generated by a TTP called a Private Key Generator (PKG), using user's identity and a Master secret key of the PKG (Noroozi *et al.*, (2014); Lu, 2011).

The direct derivation of users' public keys from their unique identities in ID-PKC scheme eliminates the need for certificates and their problems (Lu, 2011). Thus, for a user (A) to make use of another user's (B) public key for secure message exchange, A need not verify the key as the owner could not have claimed a key other his own; When A encrypts with B (or any other user's) key, the encrypted data cannot be decrypted by anyone except the individual whose identity was used to create the encryption key. And as such, it affords ease of use because keys can be used directly without recourse to any verification process. However, ID-PKC suffers a major setback; it has potential key escrow (Lai *et al*., 2011; Toorani, 2011; Lu, 2011; Gentry, 2003). The PKG has knowledge of the private key and thus, can impersonate users and decrypt their encrypted messages. Consequently, this destroys the non-repudiation functionality that PKC ordinarily affords. This perhaps explains why in some countries (e.g. the EU), ID-PKC-based schemes are not acceptable, since standards guiding the use of cryptography in those countries require users to be in "sole control" of their key pairs (Toorani, 2011). Furthermore, a compromise of the PKG's key will have catastrophic consequence, as it implies the attacker will not only be able to fraudulently create new users, he will also be able to compromise old users' security (Toorani, 2011). Finally, ID-PKC does not have a mechanism for revoking a compromised key.

2. Self-Certified Public Key Cryptography (SC-PKC)

SC-PKC was proposed in 1991 by Girault M. (Toorani, 2011; Lee *et al*., 2011; Lu, 2011). It is different from its predecessors; PKI and ID-PKC. SC-PKC is neither identity-based, since the public key is not generated from user identities, nor certificate-based, because digital certificates are not involved. The user generates a private key and secret value. It exchanges the secret value with the TTP called System Authority (SA). The SA computes the user's public key value from the secret; essentially, a "signature" of the secret value (Toorani, 2011; Lai *et al*., 2011). When this key is exchanged with another user (RP), the relying party needs not bother about the genuineness of the key, because the encrypted message can only be decrypted by the one whose secret value was used to generate the public (or encrypting) key. Thus, an impostor cannot claim another user's public key in SC-PKC scheme. As a result of this impossibility, key verification is not required. Furthermore, it is certificate-free and key escrow-free. However, SC-PKC lacks a key revocation mechanism.

3. Certificatelesss Public Key Cryptography (CL-PKC)

CL-PKC was proposed in 2003 by Al-Riyami S. S. and Paterson K. G. (Toorani, 2011; Lu, 2011). It is designed to address the key escrow problem inherent in ID-PKC. CL-PKC scheme is designed as new paradigm which lies between ID-PKC and PKI (Lu, 2011; Toorani, 2011). In CL-PKC, a user's identity is concatenated with a value generated by the TTP (called Key Generation Center or KGC) to create the public key. Similarly, to generate users' private key, it requires a "secret value" from the user and "partial secret value" generated by the KGC. This way the scheme is not affected by key escrow problem; the KGC does not have the entire private key of users and thus cannot impersonate them (Toorani, 2011; Lu, 2011). However, the problem of revocation of compromised key still persists (Toorani, 2011). Furthermore, CL-PKC is susceptible to key replacement attack which consequently leads to DoD or Denial-of Decryption (Lai *et al*., 2011; Toorani, 2011); in which case a legitimate user will not be able to decrypt encrypted message sent to him.

4. Certificate-Based Public Key Cryptography (CB-PKC)

CB-PKC was proposed in 2003 by Gentry C. (Toorani, 2011; Lu, 2011). As the name implies it requires the use of certificates managed by a CA on behalf of users. However, the certificate is essentially used as a decryption tool rather as identity document as in PKI and WoT. The scheme is implicit; RPs (sender) need not bother about key verification because at the decrypting end, users are authenticated on a real time basis by the CA before they can perform decryption (Gentry, 2003). This authentication is done via creation of certificates by the CA. In CB-PKC, it takes a receiver its private key and the *latest certificate update* to decrypt encrypted messages. This way, a users' current status as at the time of decryption is verified (i.e. verification for true ownership and validation (revocation-check) in case of a compromised key) before its given access to be able to perform decryption (Toorani, 2011). The implication here is that, the "on the demand" or "real time" key revocation or validation as required in PKI is required here too except that, while in PKI the burden is borne by the sender, in CB-PKC, it is borne at the receiver end. So the latency and high overhead as a result of frequent request to the CA (and its related consequences) in PKI is still present in this scheme.

5. Self-Generated Certificate Public Key Cryptography (SGC-PKC)

SGC-PKC was proposed in 2007 by Liu J. K, Au M. H. and Susilo W (Lu, 2011). SGC-PKC was designed to address the DoD shortcoming of CL-PKC and yet, preserves all its advantages. Similar to CL-PKC, every user is given a “partial secret key” by the KGC, and the user in turn generates his own secret value. He computes his private key by combining the secret value and the “partial secret key”. He also generates a certificate using his private key. The purpose of this self-generated certificate is similar to the one in traditional PKC. This certificate is to bind the user’s identity and his public key. The certificate can be verified using the user’s identity and public key, as the users corresponding private key was used to create it (Lai *et al*., 2011). It is this certificate that helps to prevent the DoD attack as the key replacement attack exploitable in CL-PKC, is not possible any longer; as users public keys and its copy on the certificate needs to match. Revocation of compromised key is still a problem in SGC-PKC.

2.2.3 Robust PKV Schemes

In all, it can seen that certain design targets are essential to any robust PKV scheme, which each of the previous PKV scheme tried to achieve. They can be summarized as follows;

1. Freedom from key escrow.
2. Presence of a revocation mechanism to handle compromised keys.
3. Simple validation-checking mechanism that neither consumes considerable resource (at either the users’ end or at the TTP’s end) nor has the potential to cause latency, as these would affect usability.
4. Security against attacks.
5. Implicit usage. To enable usability as users of implicit schemes don’t have to worry about identity theft.
6. Low storage requirement.

2.3 Review of Similar Works

This section presents a review of literature with respect to attempts to enabling secure communication requirements for SMS messaging. Several works have used either symmetric, asymmetric or a combination of both to improve the security of SMS.

Croft and Olivier, (2005) developed a secure SMS exchange for mobile phones. The implementation used "unbreakable" cryptography – one time pad (OTP). An OTP is a randomly generated symmetric key that is used once per communication exchange. In this implementation, the OTP is generated from the hash of a combination of International Mobile Subscriber Identity (IMSI), Temporary Mobile Subscriber Identity (TMSI), International Mobile Equipment Identifier (IMEI) and the Local Area Identifier (LAI), with only TMSI being the only source of randomness in the generated OTP or key. These components of the key, are also known by the GSM network servers. Thus, this solution relies on one user (sender) to send an SMS encrypted using the OTP key to the network provider, so that it is decrypted by the server. The server generates a new OTP key, encrypts the SMS with it and sends to the recipient. OTP is generally assumed to provide strong security. However, it has some major limitations that make it highly impractical to gain wide deployments. The obvious one is the need for communicating parties to securely exchange the OTP pads. That is why the GSM server is used in the work. So the implementation provides "partial" confidentiality as the network providers still have access to the communicated information.

Lisonek and Drahansky, (2008) developed a secure end to end SMS exchange application for mobile phones using Python programming language. The underlining cryptographic algorithm is RSA asymmetric cipher; RSA cipher was used to achieve all the four secure communication properties. Because an asymmetric algorithm is involved, there is need for a scheme for verifying users' identities. The authors proposed the use of PKI scheme for this purpose. The drawbacks of the implementation however, include (i) the inefficiency of using RSA algorithm for bulk encryption; many PKC ciphers (including RSA) are highly resource consuming, thus, they are only used in applications where symmetric ciphers are not applicable like in signature creation and verification; thus applying RSA for bulk encryption of SMS data is a problem. (ii) the high resource requirement for managing digital certificate. As detailed above, the use of PKI requires a high overhead at users and the CA's end, particularly for public key revocation checking purpose. It requires recurrent revocation request from the users' devices which are mobile phones in this case. Significant amount of mobile phone's resources like internet connectivity, battery, bandwidth and CPU will be regularly required to make use of the solution. Worst still, the likelihood of recurrent delay will likely affect the use of the solution as the revocation checks are done on a real time basis; unfortunately, this real time revocation-check is central to the security of PKI.

Anuar *et al*., (2008) developed a J2ME-based SMS messaging application (dubbed M-PKI) for mobile phones. The solution is PKC-based and provided three out of four of the requirements for secure communication. The PKI scheme is used to protect against identity theft. M-PKI is a 3 - tire SMS mobile app; SMS messaging are categorized into three varying security needs; ordinary (without encryption), internal (with AES symmetric encryption, that can be used by a small group of people e.g. staff of a company) and confidential (that uses RSA asymmetric cipher). The class 1 category of SMS did not provide any security and as such a reinvention of the conventional SMS. Although, the author advanced argument for its inclusion; it was done so that users are accustomed to using the SMS solution as an all-in one solution; that can be used to send SMS of varying security needs without having recourse to the native SMS application on the mobile phone. The class 2 SMS would have been very secure and efficient, as the AES cipher used is highly secure and very fast for bulk encryption of data, but the fact that symmetric encryption has key distribution problem, it is not ideal for communication between people separated by long distance; thus, the suggestion of its use by a close group of people. The class 3 SMS used PKC for all the secure communication requirements. The disadvantage here is the use of RSA for encryption purpose; RSA can be used for encryption, but it does it far slower than many symmetric ciphers would. To address the PKV issue in the solution, PKI is proposed. So PKI issues as identified in Lisonek and Drahansky (2008) also applies here. Furthermore, this work took some steps that are fundamentally inappropriate. An example is the delegation of the CA to perform key pair generation on behalf of users, thereby introducing *artificial* key escrow in the developed system.

Zhao *et al*., (2008) developed a secure mobile SMS application using J2ME. The application utilizes a combination of symmetric and asymmetric cryptography. The ECC asymmetric cipher was used for signature signing and verification to provide integrity and authentication security requirements. They used a symmetric algorithm to encrypt SMS for confidentiality purpose. Two users' (A and B) of this scheme will securely exchange (and save) a symmetric key K with each other using ECC asymmetric cipher. Subsequently they can exchange confidential message using the shared symmetric key. Thus, this scheme is void of bulk encryption problem as noted in Lisonek and Drahansky (2008). Other secure communication requirements are added using the ECC asymmetric cipher. Because an asymmetric cipher is involved, there is need for a PKV scheme with which users can verify that public keys belong to those who claim their ownership. To handle this, ID-PKC key management was deployed. ID-PKC scheme as detailed above has the advantage that users cannot successfully steal others' identity; user A, for instance cannot claim another user's public as its own. Furthermore, the use of certificate is not required. Consequently, the problems associated with certificate and its management is not present. The scheme provided integrity, confidentiality and authentication requirements but not non-repudiation. Non-repudiation feature is not provided, since the PKG provides the private key to the users. The caveat in the work (and with all ID-PKC-based solution) is the key escrow problem.

Toorani and Beheshti, (2008) developed a J2ME-based Secure SMS (SSMS) messaging application for m-payment systems. In addition to providing all the four secure communication requirements, SSMS also has a mechanism with which a TTP can help prove that a particular transaction did take place at some point in time. The solution deployed elliptic curve asymmetric cipher for creating and verifying signatures (for authentication and non-repudiation purposes) and AES symmetric cipher for SMS encryption. All secure SMS communication are routed via an SMS gateway/CA. A user (A) who wishes to securely communicate (make payment order via SMS) to B, sends (via SMS) user B's identity (IDb; MISDN) to the SMS Gateway/CA Server to obtain B's public key value and its validity status. The CA communicates ID_B, PK_B (B's public key) and signed $OCSP_B$ back to A. A use's the CA's public key (assumed in the work to have been securely exchanged with all users) to verify B's key status ($OCSP_B$). If $OCSP_B$ is positive, A uses ID_B and PK_B to generate a secret key K_A. Afterwards, both users use K_A to encrypt and decrypt their messages subsequently. This scheme is free from bulk encryption problem but has a partial key escrow problem; as user's public and private keys are generated at the CA. The key escrow is partial because, the private key is not used for encrypting neither the message nor the symmetric key, however, users' signature can be forged. Secondly, delay/latency issues and inefficient use of mobile phone's resources during key revocation checking is also present as users at both ends must make request to the SMS Gateway/CA Server for key status validation. Furthermore, the solution suffers from slow operation as communication between parties is not straightforward; at all times that communication is to be established between any two users, users must first exchange (and receive) some data with the CA, before the normal SMS communication between users can take place.

Al-Bakri and Kiah, (2010) developed a J2ME application for secure SMS communication. The solution uses hybrid cryptography. The hybrid solution consisted of symmetric (AES) and asymmetric (NTRU and DH) ciphers. This scheme is very fast and secure. It is a true end to end encryption scheme that provided all the four secure communication requirements. The implementation used AES algorithm to encrypt SMS for confidentiality purpose. Two users' (A and B) of this scheme will use the DH asymmetric cipher to exchange their NTRU public key. Subsequently, they used the exchanged NTRU public key to exchange (and save) an AES symmetric key. Subsequently they can exchange confidential message using the shared AES symmetric key. Thus, this scheme is void of bulk encryption problem as noted in Lisonek and Drahansky (2008), and Anuar *et al*., (2008). The developed solution requires that users of the scheme need to register and obtain an authentic copy of the application from a TTP entity to prevent identity stealing. With this, verification is addressed but validation issues still persisit; users do not have a way of confirming that a once verified public key has been revoked or not.

De Santis *et al*., (2010) developed a J2ME-based secure SMS messaging application for mobile phones dubbed SEESMS (Secure Extensible and Efficient SMS). The solution was designed to give users options to choose from a number PKC algorithm options; RSA, DSA and ECC. The implementation can also accept as add-on other PKC algorithms by users; thus, the term "Extensible". The PKC algorithms are used to provide all the four secure communication requirements. The trust of the work is the extensive performance evaluation measurements taken and conclusions drawn from them. It was emphasized that, users need to know the price (run time cost) and implication of each PKC relative to the level of security they provide. For example, the study found that RSA is generally faster than ECC but a little slower than DSA during signature creation. For signature verification, RSA is fastest followed by DSA, then ECC. Similar result was obtained for battery energy consumption because energy consumption is proportional to the execution time and complexity of the cryptographic cipher itself. RSA was found to be more battery friendly, followed by DSA and then ECC. Before any user can make use of the solution it has to register with the TTP entity. Furthermore, when two new users want to communicate, they can only obtain each others' public key from the TTP. This way, identity stealing is addressed. However, the scheme did not have a mechanism for revocation of previously held valid keys. Furthermore, the scheme uses asymmetric cipher for encryption which is less efficient.

Saxena and Payal, (2011) provided the four secure communication requirements to SMS messaging using PKC algorithms. The work is similar to the work of De Santis *et al*., 2010, because multiple asymmetric algorithms are involved. The work is the development of a J2ME application that provided users with options to use either RSA or ElGamal PKC algorithms for SMS encryption and decryption. Users also have the options to select either DSA or ECDSA digital signature algorithms. With these algorithms, a true end to end secured SMS can be exchanged. The problems in the work however, are (i) there is limitation to the flexibility that the solution purportedly affords, because many of these algorithms are not inter-operable; for example an RSA private key signed document cannot be verified by ElGamal public key. (ii) The use of PKC ciphers for bulk encryption. (iii) The solution did not define any PKV management scheme implemented nor did it mention anything about binding users' keys to specific identities. Thus, identity stealing is possible in the solution as users do not have a means to verify the genuineness or otherwise of a public key and its associated identity.

Medani *et al*., (2011) reviewed works on security solutions for SMS from 2001 to 2010. The review looked at major works involving symmetric, asymmetric (PKC) or a combination of both algorithms. It argued that the best solution so far provided are the ones involving the use of PKC algorithms; since it implied these implementations will not have key distribution problems as in symmetric-algorithm-based solutions. It went further to review works based on the infrastructure built around such PKC to support "identity binding". It concluded that, the PKI approach is by far the best approach; since in classical PKI solutions, users are in "total control" of their private keys or their decryption tool and there is mechanism for revocation of compromised key. It however, noted that despite the high level of security that is provided by these (PKI) implementations, there are concerns about the low-resource capabilities in mobile phones. When PKI is used on mobile phones, a lot of pressure is put on the mobile phone's resources during key/certificate management activities like checking for the revocation status of public keys, downloading of certificates etc.

Stoner, (2013) developed a secure mobile messaging application (dubbed PGP Messaging) for android devices. PGP Messaging uses ECC asymmetric cipher for adding authentication and non-repudiation secure communication properties, AES for encryption of the text message to add confidentiality and SHA-256 was used for message integrity. Users are meant to exchange symmetric key via ECC-based DH agreement protocol, the consequence of which is that if the same text message is to be sent more than once, then the resulting ciphertext is the same; which is at variance with what is obtainable in the original PGP (use in secure E-mail). Because asymmetric cryptography is involved, there's need for public key verification. Thus, WoT PKV scheme as used in original PGP was implemented in the developed solution. A major drawback of the use WoT as detailed earlier (chapter 2) is that, it does not have any mechanism to address revocation of users' public keys when their private keys are compromised.

Pereira *et al*., (2013) developed a two-tier secure SMS solution; one that can deployed on mobile devices (using J2ME) and the other that can be deployed on the SIM card (using C language). The solution used AES symmetric cipher, ECC asymmetric cipher and implemented CL-PKC as the PKV scheme to address the identity binding in the developed solution. The solution affords the four secure communication requirements and it is free from key escrow. However, as noted earlier, CL-PKC scheme does not have mechanism for handling key revocation.

Suresh *et al*., (2014) used an improved form of RSA algorithm; High Performance Rivest Shamir Aldeman (HP-RSA) to add secure communication requirements to SMS messaging. The trust of the work is the performance comparison between the regular RSA algorithm and the improved HP-RSA, which shows HP-RSA to require lower runtime requirements than RSA. However, the solution did not implement any PKV scheme to address identity binding in the developed solution, thus identity stealing is inherent in the solution.

From literature, it is observed that the current trend in the area of secure communication using SMS is the use of asymmetric cryptography. However, the asymmetric-based solutions in use either do not address identity binding at all or implement PKV schemes with one limitation or the other. One of such approach as seen in the reviewed works is the use of PKI. PKI would have been appropriate if and only if, the high latency and overhead can be minimized. Thus, this work proposes a PKV scheme that is similar to PKI but avoids the PKI components that are responsible for its high latency and overhead. This is achieved by replacing the frequent request/response mechanism used for key validation in PKI with a one-time response information (provided by the TTP) approach.

2.4 Summary

Key verification and validation is central in any cryptographic system that utilizes asymmetric cipher in its operation. PKI, the most popular used PKV is not optimized for use in constrained environments like mobile phones; particularly for key validation purpose. Thus, an improved method is proposed and implemented in this work. In PKI approach, public key status checking implies that "an existing (known to be valid) public key has been revoked unless a TTP states otherwise". And that is why users' must regularly confirm (via online internet connectivity) at every point in time for its current status. In the improved scheme, "an existing (known to be valid) public key has not been revoked unless a TTP states otherwise". This way, users do not have to regularly deploy their device's resources (like bandwidth) for checking keys status with the TTP. Rather, the TTP sends appropriate information to users when the status of a key changes. Finally**,** certificates are not used. Rather than RPs obtain other users' certificates and checking for authenticity and integrity, the TTP can help RPs confirm the genuineness of a user's public key and the identity attached to it. This way, the extra storage space for digital certificates and resources like bandwidth for processing them are no longer required.

CHAPTER THREE

MATERIAL AND METHODS

3.1 Introduction

In this chapter, we describe the detailed procedure carried out in the development of the secure messaging application using J2ME programming. The chapter is divided into two parts. In the first part, the materials used for the development of the application are presented, while in the second part, specific actions taken to implement each step of the methodology are presented.

3.2 Materials

This part of the chapter presents the tools/materials used for this research and how they are used.

3.2.1 Materials Used

The materials used to accomplish the aim and objectives of the work include:

Programming Software: Cryptographic ciphers are implemented on different devices by computer program written in specific programming language/software (Examples include Java, C++, Python etc.). Java programming language is the choice used in this work because Java is optimized for implementation on resource-constraint (comparatively low memory, battery, CPU etc) devices like mobile phones, PDAs, setup-boxes etc. A pruned-down subset of the larger Java library (J2SE) dubbed J2ME is designated for use in these environments. Apart from this, Java's platform-independence, free and open source features etc. are some of the attractive features that makes Java (or J2ME) the ideal candidate in this work.

Development Environment: To implement (write and test codes) any programming language, a development tool (or environment) is required. A development environment is composed of a hardware and software components.

The hardware device used in the development of the mobile application is an HP laptop (Intel ® Core ™ 2 Duo CPU U7600 @ 1.2GHz, 1.2GHz).

The software component composes of the underlining operating system of the laptop and the J2ME development software. While the Operating System on the laptop is Windows7, the J2ME development software is Netbeans (version 7.0.1) Integrated Development Environments (IDE). IDEs are better development software options as they combine in one container, all the necessary tools including a text editor (for typing the programming codes), debuggers (for correcting code errors), Software Development Kits or SDKs (for simulation and testing) etc. Netbeans IDE is free and very easy to use. Netbeans IDE is available at http://netbeans.org.

Lightweight Cryptographic Library: Many a cryptographic functionalities are not available on the original J2ME Application Programme Interface (API). Therefore, the use of external cryptographic library becomes inevitable. Bouncy Castle (BC), an external cryptography API is chosen over others (like IAIK, B3, PTMF APIs etc.) because it’s open source, free, has the largest collection of cryptographic algorithms etc. The BC library is available free at http://bouncycastle.org. Its major drawback is its large footprint (a consequence of supporting a large number of cryptographic algorithms). This large footprint can be reduced using byte code obfuscators.

Byte Code Obfuscator: The use of a very large cryptographic library like BC necessitate that a byte code obfuscator be used. Obfuscators help in removing unused classes and thus reduce the size of the final application. Obfuscators generally reduce application by several times (Yuan, 2004). For example the developed application (ProSMS) is obfuscated or shrunk from 1.3MB to 120KB. The obfuscator used is a free obfuscator “Proguard” (version 4.4) as it outperforms its competitors (Eric, 2013). The Proguard byte code obsfucator is available at http://proguard.org.

Test Device: Since the developed application is a mobile application, then a mobile phone test device becomes imperative. Any J2ME enabled mobile phone is an ideal candidate, however, Tecno (T9 model) mobile phone is used to test the developed application's functionalities and obtain measured results.

3.2.2 Installation and Configuration

The installation of all the software tools on the hardware device is done as follows**:**

1. Netbeans IDE Software, Proguard and Bouncy Castle Cryptographic Library (for mobile) are all downloaded (and saved on the desktop of the computer) from http://netneans.org, http://proguard.com and http://bouncycastle.org respectively.
2. The Netbeans IDE is installed by double-clicking (and following the prompts to "Finish"), the Netbeans Executable file on the desktop.
3. The Netbeans IDE is launched by double-clicking the IDE icon on the desktop.
4. Then a project (the secure messaging application: ProSMS) is created on the Netbeans IDE Window. ProSMS is created by;

   > file > new > New Project > Java ME > Mobile Application > Next > Project Name (type in "ProSMS" as the name of the project) > Next > Finish.

   A project named "ProSMS" is now displayed in the Projects Pane on the left hand corner of the IDE Window.
5. Then the Proguard Obfuscator is added. The Proguard obfuscator is added by:

   < ProSMS > Properties > Libraries and Resources > Add Library > ProGuard Obfuscator > Add Library > OK.
6. Then the Bouncy Castle Library is added. The Library is added by:

   < ProSMS > Properties > Libraries and Resources > Add Jar/Zip > Deskstop > lcrypto-j2me-148 > Open > zips > Open > cldc_bccore_classes > Open > OK.

With the above steps, the Development Environment is now ready for use. Also the developed application, in its smallest form is created. What remains is the addition of further lines of codes to implement whatever function(s) the application is meant to execute.

7. The developed application is tested 2 ways. The first is in a simulation environment available on the Netbeans IDE to see the behaviour of the functionalities of the application. The second test is on a real device, Techno T9 model. The developed application is transferred from the development environment unto the test device via Bluetooth.

3.3 Method

This section details the implementation of each of the methodology as outlined in section 1.4.

3.3.1 Selection of a Cryptosystem Type

As established in section two, specific type of cryptographic ciphers are best applied to achieve specific secure communication objectives. While asymmetric ciphers are best for adding authentication and non-repudiation functionalities, symmetric ciphers are most efficient for message encryption (for confidentiality) and Hash Function ciphers are best for adding integrity functionality.

Therefore, since the main objective (first objective) of the research is the development of a messaging application with all secure communication functionalities, the chosen cryptosystem type adopted in this work is Hybrid Cryptosystem; one that utilizes a combination of symmetric, asymmetric and Hash Function ciphers to achieve all the four requirements of secure communication.

3.3.2 Selection of Appropriate Ciphers

Having determined in the preceding section that a combination of symmetric, asymmetric and hash function cryptographic ciphers is needed to meet the secure communication objective of this research, the specific ones used are chosen as follows.

3.3.2.1 *Selection of a Symmetric Cryptographic Cipher*

The symmetric cipher adopted in this work is Twofish block cipher because it is adjudged to be very secured (Schneier *et al*., 2000). Even though it has three (128, 192 and 256) possible key length, the 128 bits key length is used in this work as this is considered secured enough (Kessler, 2014; Stallings, 2004).

3.3.2.2 *Selection of an Asymmetric Cryptographic Cipher*

The Asymmetric algorithm adopted in this work is RSA. RSA is a full-fledged (used for both encryption and digital signing) and the simplest asymmetric cipher to date (Schneier, 1996; Anuar *et al.*, 2008; Ireland 2010). RSA key size of 1024 bits is used in this work, as this is considered secured enough (Kessler, 2014; Stallings, 2004).

3.3.2.3 *Selection of Hash Function*

Secure Hash Algorithm or SHA-2 (SHA-256 option), developed by NIST in 2001 is adopted in this work. SHA-2 families of hash functions are more secure than earlier hash functions like SHA1, Message Digest or MD4, etc. SHA-256 can take up data of about 2^{64} bits in size and produces 256 bits hash value (Kessler, 2014).

3.3.3 Development of a Communication Protocol

Having determined to use a cryptosystem that combines symmetric, asymmetric and hash ciphers, then a protocol of how they are used in a single cryptosystem for the purpose of secure messaging becomes important. The communication protocol used here is patterned to the PGP hybrid cryptosystem used for secure e-mail communication; where a per session symmetric key is used to encrypt the message, and the symmetric key itself is encrypted with the recipient's public key. The use of session key has the advantage that, it increases the amount of work required to compromise a message protected by it.

Thus, when two communicating partners (A and B) want to exchange SMS securely using the developed application, the following steps are taken:

1. A sends B his public key (and vice-versa)
2. For B to send secure message to A, B needs to;
    a. Generates a random session key (using symmetric algorithm), K.
    b. Encrypts the session key K using A's public key to produce Pk(K).
    c. Encrypts the real SMS (or message M) with K to produce K(M).
    d. Pass K (M) through SHA256 Hash Function to produce H[K(M)].
    e. Signs H [K (M)] using its private or secret key Sk to obtain Sk{H[K(M)]}
2. Sends as secure SMS a combination of Pk(K), K(M), and Sk{H[K(M)]}.
3. At the receiving end, A decrypts the encrypted symmetric key by using the its own private key (since A's public key in B's possention was used for the encryption). The decrypted symmetric key is subsequently used to decrypt the encrypted message.

3.3.4 Development of Codes for Secure SMS Messaging

This section details the actual writing of the codes of the application related to secure SMS messaging as detailed in section 3.3.3. The ProSMS application created in the development environment in section 3.2.2 is developed further by adding relevant J2ME codes to implement all the functionalities required. Essential parts of the developed application are presented in succeeding sections.

3.3.4.1 *Required (Major) Interfaces*

Here the User Interfaces (UIs) with which the user interacts with the application are created. In all, there are 38 UIs (see page 86) in the developed application. The rationale for creating the major ones and how they are created are outlined as follows:

1. Access Control: An interface for password access/denial is created A password registration, confirmation and acceptance/denial is required to prevent unauthorized access to the application. The password is stored in a record store (PasswordStore). Figure 3.1 shows the flow chart diagram for the access acceptance/denial interface.

J2ME code used to implement this access control can be found in page 95. The code is adapted from Berggren and Winnem (n.d).

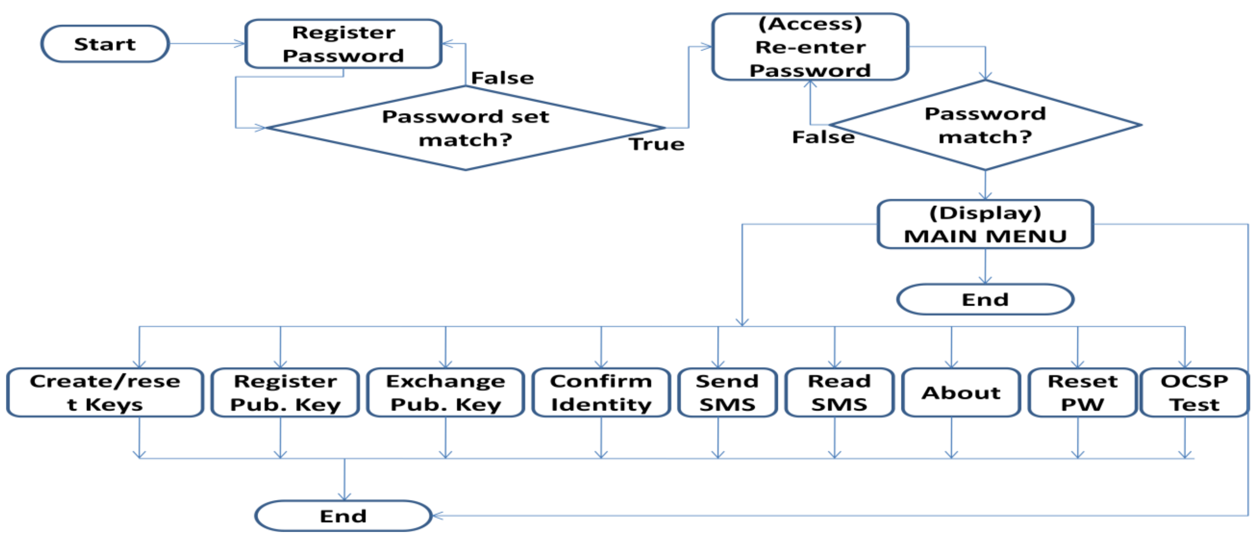


Figure 3.1: Password Registration and Access Acceptance/Denial Flow Chart

2. Main Menu: The Main Menu (MM) UI houses the links (nine in total) to other UIs in the application (see Plate 3.1). The MM itself is reached after a successful login process via the access control UI. J2ME code used to implement the MM is as follows;

```
String[] MainMenu = {"Create/Reset Keys","Register/Exchange Public Key with PKA","Exchange Public Key with Users","Confirm Identity","Send SMS", "Read SMS","About","Reset Password", "OCSP Test"};
```

Plate 3.1 shows the Test Device’s display of the MM of the developed application.

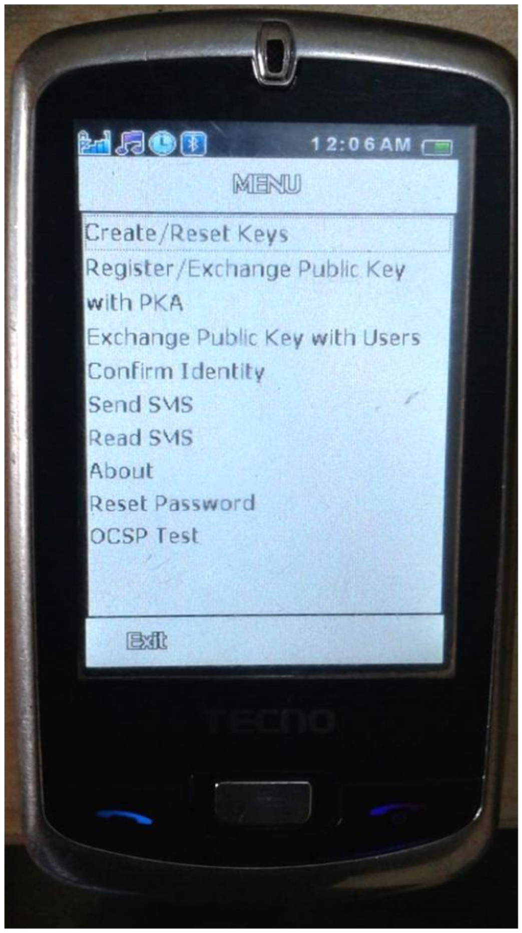


Plate 3.1: Display of the Main Menu

Subsequently, commands are given to enable each of the nine (9) links on the MM to proceed to their respective destination (UI) when clicked by the user.

The J2ME code for implementing a move from the MM to the UIs of the links is as follows;

```
public void commandAction(Command c, Displayable s) {
...
if(c == List.SELECT_COMMAND) {
int menuindex = menuList.getSelectedIndex();
else if(menuindex == 0) { // THIS VALUE “0” IMPLIES THE FIRST LINK (CREATE/RESET KEY)
updateTicker();
alertAbout = new Alert("Crypto Keys", "", null, AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,f1);//Go TO f1 (RSA KEY GENERATION UI) WHEN THIS LINK IS PRESSED
}
//OTHERWISE IMPLEMENT SAME DEPENDING ON THE LINK PRESSED
else if(menuindex == 1) {//I.E. IF SECOND LINK (REGISTER/EXCHANGE PUBLIC KEY WITH PKA)
...
}
```

The user interfaces that results from the MM are the following;

a. **Create or Reset Key**: Since RSA asymmetric cipher is used in this work, a UI for generating and saving the generated RSA key is required (see section 3.3.4.2 for implementation codes). Thus, an interface for RSA key generation (and storage) is created. The generated RSA key is stored in a record store (RSAKeyPairStore). Figure 3.2 is a flow chart diagram for RSA key generation and storage.

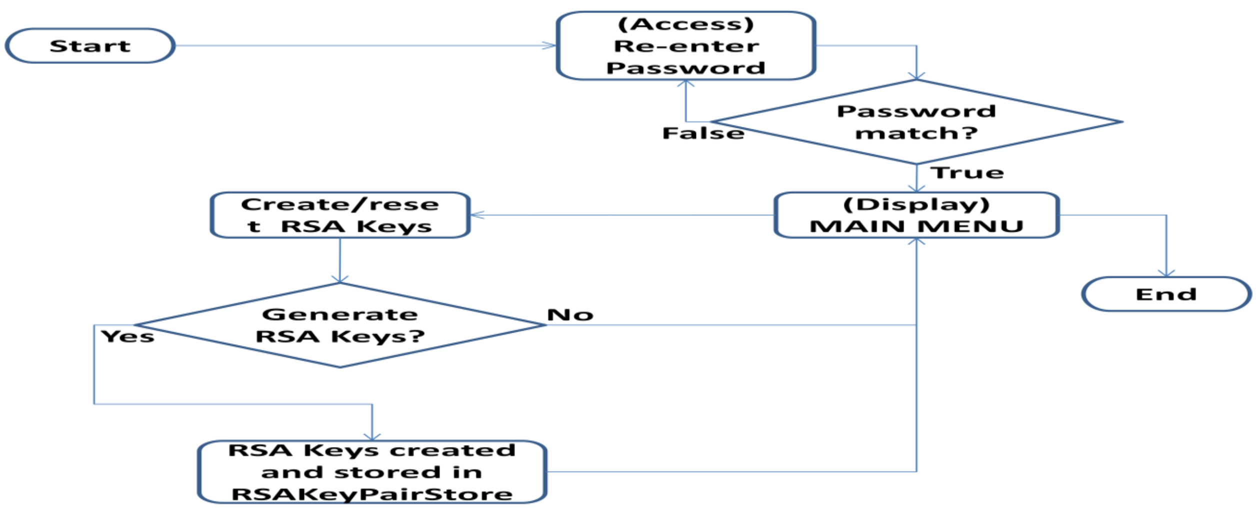


Figure 3.2: RSA Key pair Generation and Storage Flow Chart Diagram.

b. **Register/Exchange Public Key with Public Key Authenticator (PKA)**: From (a) above, because an asymmetric cipher is involved; there is a need for a framework to bind every public key to specific individuals/entities. Thus, an interface for exchanging (and registration) a user's public key with a PKA (the PKA is Trusted Third Party Entity) is created. Diffie-Hellman (D-H) key agreement protocol is implemented on this interface (see section 3.3.6.1 for implementation codes). J2ME's Wireless Messaging API (WMA) was implemented to exchange the resulting D-H data between the user and the PKA as SMS.

c. **Exchange Public Key with Users:** In any cryptosystem that asymmetric ciphers are used, users must be in possession of other communicating partners' PKD or Public Key Details which include public keys, username, phone number etc. Thus, a UI that enables users to exchange the public key component of the RSA key pair generated and stored in (a) is created. J2ME's Wireless Messaging API (WMA) was implemented for sending and receiving users' PKD as SMS. The received PKD is stored in the PhonebookRecordStore.

d. **Confirm Identity**: The PKD of users received in (c) need to be verified to be genuine or not. Thus, an interface to enable users perform this task is created. To verify a user's PKD, the application fetches the user's PKD from the PhonebookRecordStore, signs it, and sends the signed message as SMS to the PKA. The signing/verification of message is possible because the user and the PKA have exchanged their public keys in (b).

e. **Send SMS**: Since this application is essentially a mobile messaging application, an interface for secure messaging (Sending of SMS) is created. This interface prompts the user to select a communicating partner from the list of users with whom key exchange has taken place (i.e. (c) above). When a user is selected, the user's PKD is fetched from the PhonebookStore and presented to the user alongside a textbox where SMS/message can be typed in. When the user presses "send" button, the typed SMS is passed through a number of cryptographic processes to add the all secure communication functionalities to the SMS before it is sent (see section 3.3.4.2 for implementation codes). The time it takes each of the cryptographic process is recorded and displayed on RUNTIME_MESSAGING interface (page 86). The result of the test conducted on section 3.3.8 (which is a consequence of the sending of SMS) is obtained from the RUNTIME_MESSAGING UI.

f. **Read SMS:** This is the interface for gaining access (for checking) saved messages.

g. **About**: This is the interface for guidelines (Help) on general use of the application.

h. **Reset Password**: An interface to change password is also created. This enables the user to change the password as many times as possible, particularly when users' feel their password might have been compromised.

i. **OCSP Test**: This interface is created to enable the application to send and receive OCSP data. This become imperative as the second objective of this work is to measure resource/cost requirements for using PKI (via OCSP) on mobile phone. The time it takes to perform OCSP test is recorded and displayed on OCSP_MEASURE UI. The result of the test conducted on 3.3.9 (which is a consequence of OCSP Test) is obtained from the OCSP_MEASURE interface (see pages 130 to 132 for implementation codes).

j. **Miscellaneous Interfaces:** (i) An interface (logical) for receiving SMS (and further processing) is created. The interface is not accessible manually, it's the application that automatically fetches it when an SMS is received by the application to display the received message; the SMS can be either be DH key details (received from the PKA as in (b) above, users PKD (received from other users as in (c) above, response to verification requests and revocation information (received from the PKA as in (d) above) or secure SMS (received from other users as in (e) above. (ii) the various interfaces for presenting results of test like runtime requirement for cryptographic processes, OCSP test, etc.

3.3.4.2 *Cryptographic Processes*

In the developed application, cryptographic processes 1 through 5 are used to add secure communication requirements by transforming SMS data from one state to another. These codes are adapted from Yuan (2004).

1. Twofish Key Generation: A method for (constantly) generating Twofish key at every instant a user sends a secure SMS. J2ME code used in generating Twofish key is as follows;

```
public void generateTwofishKey () throws Exception {
            long difference = System.currentTimeMillis();
            SecureRandom sr = new SecureRandom();
            Twofishkey = new byte [16]; //i.e. 16bytes 16* 8 = 128 bits
            sr.nextBytes(Twofishkey);
            TwofishinitV = new byte [16];
            sr.nextBytes(TwofishinitV);
            difference = System.currentTimeMillis() - difference;
```

```
f7.append(" Twofish generation of 128 bits key size took" +
difference + "ms");
}
```

2. Twofish Encryption and Decryption: Methods for symmetric encryption and decryption of SMS message using Twofish cipher to add confidentiality security functionality to the communicated messages (SMS). The size of SMS data used to test the encryption and decryption is 120 bytes. J2ME code used for Twofish encryption and decryption are as follows:

Encryption Process

```
public byte [] TwofishDecrypt2 () throws Exception {
            long difference = System.currentTimeMillis();
            byte[] inBytes1 = Base64.decode(newMessgtwo.getString().getBytes());
            BufferedBlockCipher cipher =
            new PaddedBufferedBlockCipher(
            new CBCBlockCipher(new TwofishEngine()));
            ParametersWithRandom piv = new ParametersWithRandom (
            (new KeyParameter(bnmjhg)));
            cipher.init(false, piv);
            byte[] result = new byte[cipher.getOutputSize(inBytes1.length)];
            int len = cipher.processBytes(inBytes1, 0,
            inBytes1.length, result, 0);
            try {
            cipher.doFinal(result, len);
            difference = System.currentTimeMillis() - difference;
            } catch (CryptoException ce) {
            }
            newMessgthree2.setString(new String(result).trim());
            f15.append(" Twofish Decryption took " +  difference + "ms");
            String srtf = result.toString();
            return result;
            }
```

Decryption Process

```
public byte [] TwofishDecrypt () throws Exception {
            long difference = System.currentTimeMillis();
             byte[]                           inBytes1                        =
             Base64.decode(ReceivedKeyDetails6.getString().getBytes());
            BufferedBlockCipher cipher =
            new PaddedBufferedBlockCipher(
            new CBCBlockCipher(new TwofishEngine()));
            ParametersWithIV piv = new ParametersWithIV (
            (new KeyParameter(Twofishkey4)), TwofishinitV4);
            cipher.init(false, piv);
            byte[] result = new byte[cipher.getOutputSize(inBytes1.length)];
            int len = cipher.processBytes(inBytes1, 0,
            inBytes1.length, result, 0);
```

```
try {
cipher.doFinal(result, len);
difference = System.currentTimeMillis() - difference;
} catch (CryptoException ce) {
}
newMessgthree2.setString(new String(result).trim());
f15.append(" Twofish Decryption took " +  difference + "ms");
String srtf = result.toString();
return result;
}
```

3. RSA Key Generation and Serialization: Method for generating RSA key pairs and subsequently serializing (saving) them. A key size of 1024 bits is used, as this key size is considered secured (Kessler, 2014; Stallings, 2004). J2ME code used for RSA key generation and serialization are as follows:

```
public void generateRSAKeyPair () throws Exception {
            long difference = System.currentTimeMillis();
            SecureRandom sr = new SecureRandom();
            BigInteger pubExp = new BigInteger("10001", 16);
            RSAKeyGenerationParameters RSAKeyGenPara =
            new RSAKeyGenerationParameters(pubExp, sr, 1024, 80);
            RSAKeyPairGenerator RSAKeyPairGen = new RSAKeyPairGenerator();
            RSAKeyPairGen.init(RSAKeyGenPara);
            AsymmetricCipherKeyPair keyPair = RSAKeyPairGen.generateKeyPair();
            RSAprivKey = (RSAPrivateCrtKeyParameters) keyPair.getPrivate();
            RSApubKey = (RSAKeyParameters) keyPair.getPublic();
            difference = System.currentTimeMillis() - difference;
            display.setCurrent(RUNTIME_MESSAGING );
             RUNTIME_MESSAGING  .append("  RSA  Key  pair  generation  took  "  +
             difference + "ms");
            openRecStore();
            SerializeRSAKeyPair ();
            closeRecStore();
            if (RSApubKey == null && RSAprivKey==null){
            throw new Exception("Generate RSA keys first!" );
            }else {
            }
            }
//Serializion of the private key component
public void SerializeRSAKeyPair() // writes the generated RSA Key Pairs to the
                                    // RecordStore
            {
            BigInteger mod = RSAprivKey.getModulus();
            String mods = mod.toString();
            BigInteger privExp = RSAprivKey.getExponent();
            String privExps = privExp.toString();
            BigInteger pubExp = RSAprivKey.getPublicExponent();
            String pubExps = pubExp.toString();
            BigInteger dp = RSAprivKey.getDP();
            String dps = dp.toString();
```

```
BigInteger dq = RSAprivKey.getDQ();
String dqs = dq.toString();
BigInteger p = RSAprivKey.getP();
String ps = p.toString();
BigInteger q = RSAprivKey.getQ();
String qs = q.toString();
BigInteger qInv = RSAprivKey.getQInv();
String qInvs = qInv.toString();
String [] strings = {mods, privExps,pubExps,dps,dqs, ps,qs,qInvs};
writeStream(strings);
}
```

4. RSA Encryption and Decryption of Twofish Key: Methods for RSA encryption and decryption of Twofish session key, for ensuring secure transfer of Twofish secret keys between communicating partners. J2ME code used for RSA encryption and decryption of Twofish keys are as follows:

Encryption Process

```
public byte [] RSAEncryptTwofish () throws Exception {
             long difference = System.currentTimeMillis();
             String asdfghj = TwofishCombinedField.getString();
             byte[] inBytes = TwofishCombinedField.getString().getBytes();
             if (newRSApubKey == null){
             throw new Exception("Generate RSA keys first!");
             }
             AsymmetricBlockCipher eng = new RSAEngine();
             eng = new PKCS1Encoding(eng);
             eng.init(true, newRSApubKey);
             difference = System.currentTimeMillis() - difference;
             f7.append(" RSA Encryption of Twofish (session) key took " +
             difference + "ms");
             byte [] vbnm2 = eng.processBlock(inBytes, 0, inBytes.length);
             String asdfgh2 =  new String (vbnm2);
             newMessgtwob.setString(new String(asdfgh2));
             return eng.processBlock(inBytes, 0, inBytes.length);
             }
```

Decryption Process

```
public byte [] RSADecryptTwofish () throws Exception {
             long difference = System.currentTimeMillis();
             byte[] inBytes1 = ReceivedKeyDetails7.getString().getBytes();
             if (newRSAprivKey == null){
             throw new Exception("Generate RSA keys first!");
             }
             AsymmetricBlockCipher eng = new RSAEngine();
             eng = new PKCS1Encoding(eng);
             eng.init(false, newRSAprivKey);
             difference = System.currentTimeMillis() - difference;
```

```
f8.append(" RSA Decryption of Twofish key of data of size" +
inBytes1.length + "bytes" + " took " +  difference + "ms");
newMessgthree.setString(new String(eng.processBlock(inBytes1, 0,
inBytes1.length)).trim());
String TwofishKeynew = newMessgthree.getString(); //Twofishkey
TwofishinitV
String takeKey = TwofishKeynew.substring(0,
TwofishKeynew.indexOf("#"));
String takeinitV =
TwofishKeynew.substring(TwofishKeynew.indexOf("#")+1,
TwofishKeynew.length());
return eng.processBlock(inBytes1, 0, inBytes1.length);
}
```

5. RSA signature creation and verification: Methods for RSA Signature creation and verification to add authentication and non-repudiation functionalities to the communicated messages (SMS). J2ME code used for RSA signature and verification are as follows:

Signature Creation

```
public byte [] RSASign () throws Exception {

        long difference = System.currentTimeMillis();
        byte[] inBytes2 = newMessgtwo.getString().getBytes();
        if (newRSAprivKey == null)
        throw new Exception("Generate RSA keys first!");
        SHA256Digest dig = new SHA256Digest();
        RSAEngine eng = new RSAEngine();
        PSSSigner signer = new PSSSigner(eng, dig, 64);
        signer.init(true, newRSAprivKey);
        signer.update(inBytes2, 0, inBytes2.length);
        difference = System.currentTimeMillis() - difference;
         f7.append("Digital Signature of data of size" + inBytes2.length +
         "bytes" + " took " +  difference + "ms");
        String vbn = signer.generateSignature().toString();
        byte [] vbnm = signer.generateSignature();
        String asdfgh =  new String (vbnm);
        newMessgtwoa.setString(new String(asdfgh));
        return signer.generateSignature();
        }
```

Signature Verification

```
public boolean RSAVerify () throws Exception { //ReceivedKeyDetails6
        long difference = System.currentTimeMillis();
        byte[] mesg = ReceivedKeyDetails6.getString().getBytes();
        byte [] sig = ReceivedKeyDetails4.getString().getBytes();
        if (newRSApubKey == null)
        throw new Exception("Generate RSA keys first!");
        SHA256Digest dig = new SHA256Digest();
        RSAEngine eng = new RSAEngine();
        PSSSigner signer = new PSSSigner(eng, dig, 64);
```

```
signer.init(false, newRSApubKey);
signer.update(mesg, 0, mesg.length);
difference = System.currentTimeMillis() - difference;
 mainForm4.append("  Signature  verification  message  of  size"  +
 mesg.length +
"bytes" + "and Signature of size" + sig.length + "bytes " + "took" +
difference + "ms");
if (signer.verifySignature(sig) == true){
mainForm4.removeCommand(confirmUser);
mainForm4.addCommand(decrypt);
 alert = new Alert("Message Verication", "This message HAS NOT BEEN
 modified. Click"
+ "decrypt" + "in the menu to obatain the original message", null,
AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4);
}
else if (signer.verifySignature(sig) == false){
 alert  =  new  Alert("Message  Verication",  "Oops,  This  message  HAS
 BEEN modified!. Click"
+ "Discard" + "in the menu to delete it", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4);
}
return signer.verifySignature(sig);
}
```

3.3.4.3 *Data Storage using Record Management Store (RMS)*

Important data (log in details, phonebook details etc) without which the application cannot function properly are required to be securely stored in the application. Persistent storage in J2ME applications can be done using Record Store, Resource File or File Connection APIs. Whereas Record Store is generally applicable, Resource files and File Connection APIs are optional APIs; they are optionally supported on some java enabled phones. Thus, the record store is utilized via the Record Management System RMS API. The following record stores are created;

a. PhoneBookStore for saving Phonebook/Users' contact details. The code (see pages 125 - 126) used for creating Phonebook for the application is adapted from Anuja (2011).
b. ReceivedSMSStore for saving received Secure SMS.
c. PasswordStore for saving user's Password details.
d. RSAKeyPairStore for saving user's RSA asymmetric key pair.

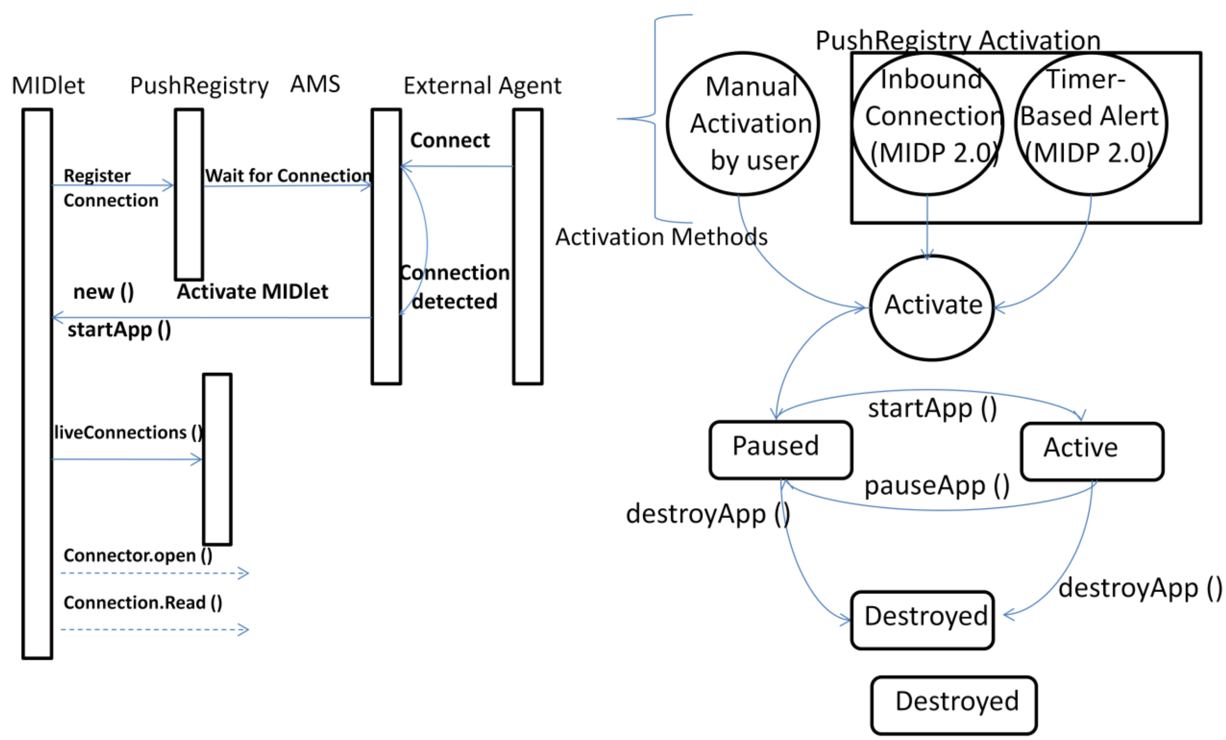
3.3.4.4 *Port for Receiving SMS*

Computing devices, including mobile phones receive data via designated ports. For mobile phones, when specific ports are not designated for receiving SMS, the SMS is received at the default port (number 0); thus, the SMS is received in the phone's native inbox. For an application like ProSMS, SMS must be received in "inbox" other than the mobile phone's native inbox so that necessary further processing like, decryption, verification etc. can be implemented on the received messages. Thus, a port number "5000" is created (see page 91 for implementation code). The port is used to receive the following messages, distinguished with different preceding values to enable the application differentiate (the messages) and take specific actions when they are received.

1. Users' Public Key Details. This is a "text" type SMS identified with preceeding code "0000" (see page 99 for implementation code)
2. DH key agreement first data. This is a "text" type SMS identified with preceeding code "1111" (see page 99 for implementation code)
3. Secure SMS messages. This is a "binary" type SMS identified with preceeding code "2222". (see page 100 for implementation code)
4. Revocation data. This is a "binary" type SMS identified with preceeding code "3333". (see page 100 for implementation code).
5. Response to a PKV request. This is a "binary" type SMS identified with preceeding code "4444". (see page 100 for implementation code).
6. DH key agreement second data. This is a "text" type SMS identified with preceeding code "5555" (see page 99 for implementation code)
7. Public key encrypted with a symmetric key obtained via DH key protocol. This is a "binary" type SMS identified with preceeding code "6666". (see page 99 for implementation code).

3.3.4.5 *Auto-launch Capability*

The application is enabled for both normal (manual activation by user) startup and automatic launch (via an inbound connection). Since this is messaging application, implementing only a user-effected startup of the application would not be effective; as it implies the application will have to be in active mode always (which is not ideal). The automatic startup is activated whenever there is an incoming SMS while the application is in the "off" mode. J2ME's Push Registry functionality is used to achieve this. The Push Registry enables MIDlets to be self-launched (without user initiation) automatically (Enrique, 2003). Figure 3.3 (a and b) shows the Schematic Diagrams of Network Activation Sequence.



(a). Network Activation Sequence (b). MIDlet Activation Life-Cycle

Figure 3.3: Schematic Diagrams of Network Activation (Enrique, 2003)

Auto-launch is implemented in this work by adding the following line of code;

```
//Auto-Launch implementation

PushRegistry.registerConnection("sms://:5000","prosms_1", "*");
```

3.3.4.6 *Use (summary) of the Secure Messaging Aspect of the Mobile Application*

Following the protocol illustrated in section 3.2.4 and writing J2ME codes as illustrated in succeeding sections results in the creation of the secure messaging application "ProSMS". Users can use ProSMS to exchange secure SMS using the following steps;

1. Users are made to first choose (and save) "log in" details in the form of password.
2. Then, they are required to first generate (and save) a pair of private and public RSA keys
3. Afterwards, they can also exchange only the public key component of the key pair with others via traditional SMS.
4. When two users (say A and B) have exchanged their public keys between each other, either of them (say B) can use the application to prepare an SMS message, **M**.
5. Then $\mathbf{M}$ is encrypted using symmetric Twofish (128-bit key, $\mathbf{Ks}$) algorithm to arrive at $\mathbf{Ks}(\mathbf{M})$. This affords confidentiality of the SMS message. The Twofish key ($\mathbf{Ks}$) is used only once (i.e. one-time) or as session key (only for one SMS exchange); for every SMS a new Twofish key is generated.
6. The Twofish key (or session key, $\mathbf{Ks}$) is encrypted using an intended recipient's (say A's public key in B's possession) public key (**Pk**) to obtain $\mathbf{Pk}(\mathbf{Ks})$. This serves two purposes;
   - (i) Secure exchange of the generated session key, as only the corresponding private key of the intended recipient (A) can decrypt and thus be used to obtain the session key.
   - (ii) It affords authentication of the recipient. It is a confirmation that the intended recipients' public key matches the corresponding private key that would be used to recover the session key and subsequently be used to obtain the original message.
7. $\mathbf{Ks}(\mathbf{M})$, the output from 5 above, is passed through SHA256 message digest to obtain a digest **D** of the encrypted message $\mathbf{Ks}(\mathbf{M})$. This affords integrity of the message. At the receiving end, another digest (**D'**) is calculated and compared with D. If D = D', then the

integrity of the message **Ks**(**M**) has not been compromised. But if D ≠ D', then **Ks**(**M**) would have been changed in transit.

8. The non-repudiation property is achieved by digital signing ("encryption") of the message digest (D from 6 above) using the sender's private key (**Sk**) to obtain **Sk**(**D**). At the receiver's end, this is verified by the sender's public key available with the receiver of the message. The implication of this is that, if this signature is verified by the sender's public key in others' possession, then this message could not have emanated from anyone other the real sender, since only he is in possession of the signature signing tool (corresponding private key).

9. **Sk**(**D**), **Pk**(**Ks**) and **Ks**(**M**) (i.e. digital signature, encrypted session key and encrypted SMS respectively) are concatenated to obtain **SecMsg**.

$$SecMsg = Sk(D) + Pk(Ks) + Ks(M) \tag{3.1}$$

10. It is **SecMsg** that is sent as secure SMS to the receiver. Figure 3.4 is flow chart diagram for sending SMS.

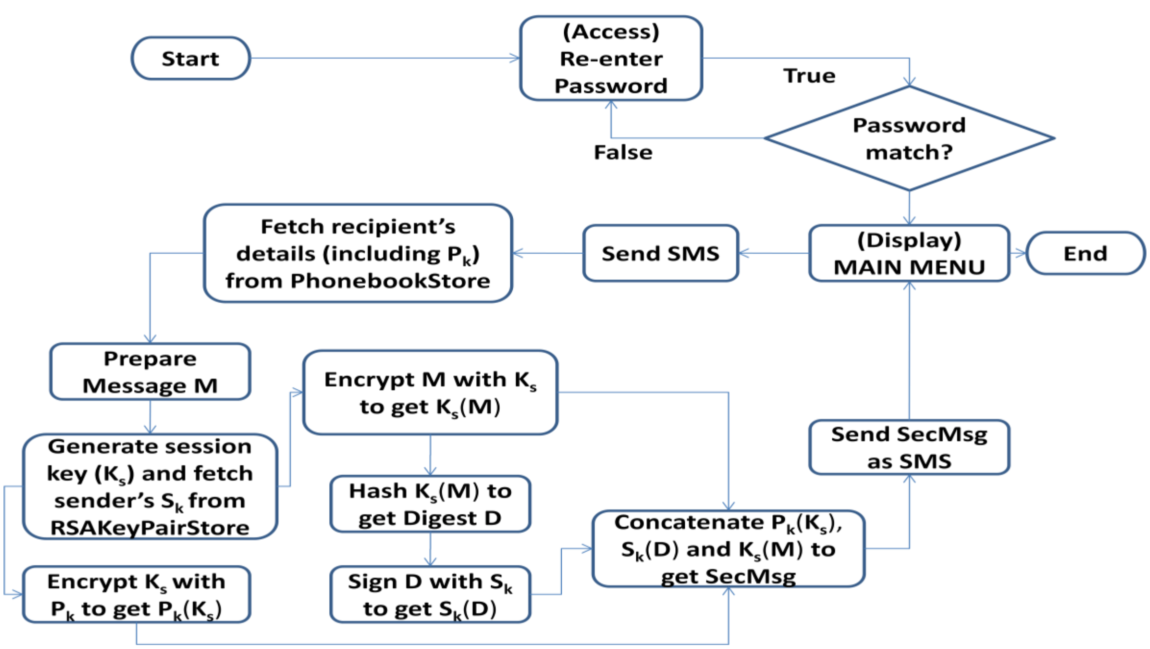


Figure 3.4: ProSMS Sending SMS Flow Chart Diagram

At the receiver's end, the ProSMS application is used to recover the original message M using the above steps in reverse order.

1. The SMS message is received as $\mathbf{SecMsg}$. This is separated into its components as $\mathbf{Sk}(\mathbf{D})$, $\mathbf{Pk}(\mathbf{Ks})$ and $\mathbf{Ks}(\mathbf{M})$.
2. The sender's identity is confirmed (authenticated) by using its public key in the receiver's possession to verify the digital signature $\mathbf{Sk}(\mathbf{D})$ signed using the sender's private key. If the signature verification fails (i.e. when the "sender's" public key cannot verify the signature) the user is notified and the process is brought to end (i.e. message is deleted). If the signature verification is successful (at this point, the user is sure the message is truly from the sender), this yields the digest D (recall that the signature in step 7 at the sender's end is actually an "encryption" process; used to "encrypt" (i.e. sign) D. Thus, this reverse process is actually a "decryption" process; used to recover D.
3. Thereafter, a new message digest is calculated by passing the encrypted SMS component of the received message (i.e. $\mathbf{Ks}(\mathbf{M})$) through SHA256 message digest algorithm to obtain D'. This new digest is compared with that obtained in 2 above. If D = D', then the integrity of the message $\mathbf{Ks}(\mathbf{M})$ has not been compromised. But if D ≠ D', then **K**ansit. This is how integrity of the exchanged message, $\mathbf{Ks}(\mathbf{M})$ is achieved.
4. The encrypted Twofish key, $\mathbf{Pk}(\mathbf{Ks})$ is decrypted using the recipient's private key to obtain $\mathbf{Ks}$, (the 128bit Twofish session key). This serves two purposes;
   (i) Secure exchange of the session key; as only the corresponding private key of the intended recipient decrypts the encrypted key; thus the session key is obtained.
   (ii) It affords authentication of the recipient. It is a confirmation that the intended recipients' public key matches the corresponding private key that would be used to recover the session key and subsequently be used to obtain the original message.

5. Then **Ks**(**M**) is decrypted using **Ks** obtained from step 4 above, to obtain the original message **M**. The confidentiality of the SMS is preserved, as only the intended receiver can read the original message **M**. Figure 3.5 is the flow chart diagram for receiving secure SMS.

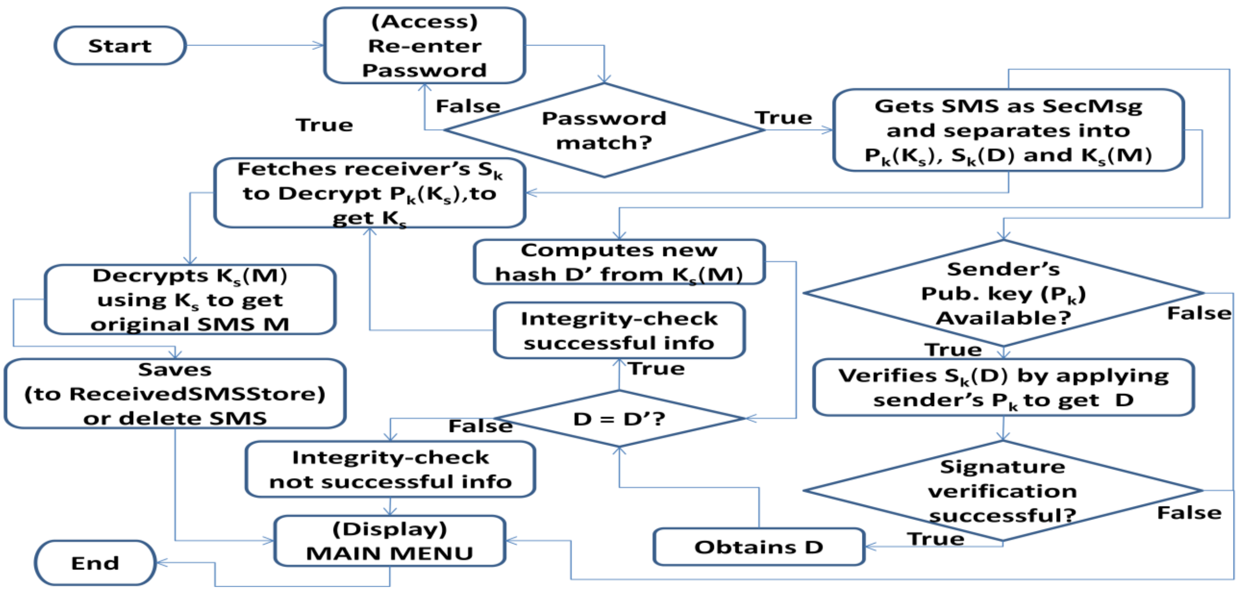


Figure 3.5: ProSMS Receiving SMS Flow Chart Diagram

3.3.5 The Proposed SB-PKC PKV Scheme

Any PKC-based system requires a form of management structure for validating the genuineness or otherwise of users' public keys and their associated claimed identity. Also a means of checking that known to be genuine keys are still valid as the time of use is equally essential in a PKC-based system. The public keys used in secure messaging in the previous section are not authenticated in anyway, neither is their framework for checking their revocation status. To address this, SMS-based Public Key Cryptography (SB-PKC) PKV scheme is proposed.

SB-PKC scheme is designed similar to PKI. However, its model for key validation/revocation-check is reworked to avoid the problems in PKI. It has been observed that, the regular request (by RPs) and consequent response (by CA) for certificate status checking in PKI account for the high overhead in the scheme (see Figure 3.6). Therefore, designing a revocation check mechanism that minimizes this would help minimize the overheads and improve efficiency. This is particularly important for resource constrained environments like mobile phones.

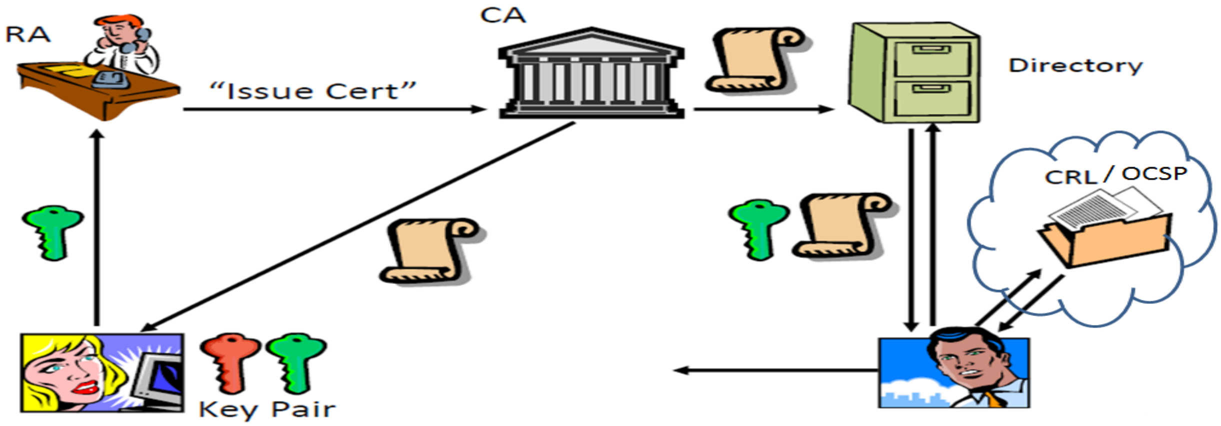


Figure 3.6: Location of Bottleneck in PKI

The SB-PKC can be summarized in the following steps (see Figure 3.7):

1. Users generate own pair of PKC keys.
2. Afterwards, they (users) can also exchange only the public key component of the key pair (and other identity information) with others via traditional SMS. At this level, because the key exchange is not protected, the exchanged key is stored with "UNVERIFIED" status. With this status, secure communication using the developed application is not possible (i.e it will be blocked) until a PKV response is received from the Public Key Authenticator (PKA) affirming the genuineness of a public key and its claimed identity.

The PKA is the TTP in SB-PKC similar to the CA in PKI. The role of the PKA is assigned to the network provider of each of mobile phone subscriber. In many parts of the globe, SIM registration is a norm. Network providers currently maintain database/servers that store their subscribers' personal identity information. This work intends to leverage on this resource, so that what users of this solution (ProSMS) need to do is to add their public key info to their existing data with their network providers.

3. Afterwards, they can register (and exchange) the public key component of their key pair with the PKA via DH key agreement protocol. This step serves two purposes; (i) RPs can henceforth ask about the genuineness of a user's public key from the PKA since the PKA now has the public key information of registered users, and (ii) a user and the PKA can now exchange authenticated messages as this key exchange enables them to be able sign and verify digital signatures (thereby ensuring communications are secured).

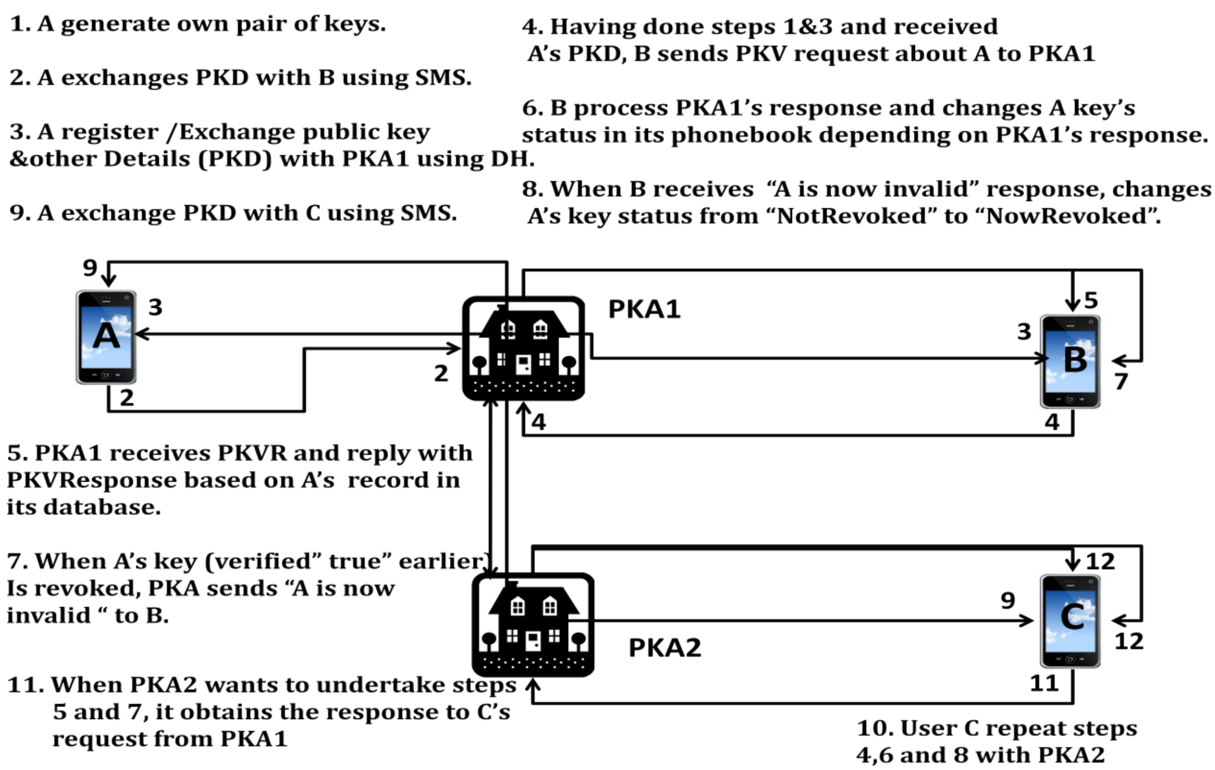


Figure 3.7: Interactions between Parties of the Proposed SBC-PKC

So far, steps 1 through 3 are similar steps that would have taken place even in a typical PKI. However, PKAs only need to maintain the record/database; a certificate or any item is not issued by the PKA to the registered users.

4. When a user (say B) comes across another user's (A) public key and associated identity for the first time (step 2), it sends A's details to its PKA via an authenticated (A can now sign

messages with the PKA, since both have exchanged keys in step 3) or signed SMS by using its private key.

5. When the verification request is received at the PKA, the PKA searches its record for the availability and correctness of the sent request in its database using some criteria as shown in the flow chart of Figure 3.8 and returns a signed response to the user; stating that the "requested public key and the attached identity" is valid (exist and not revoked, thus can be relied upon) or invalid (never exist or has been revoked and should never be trusted or used).

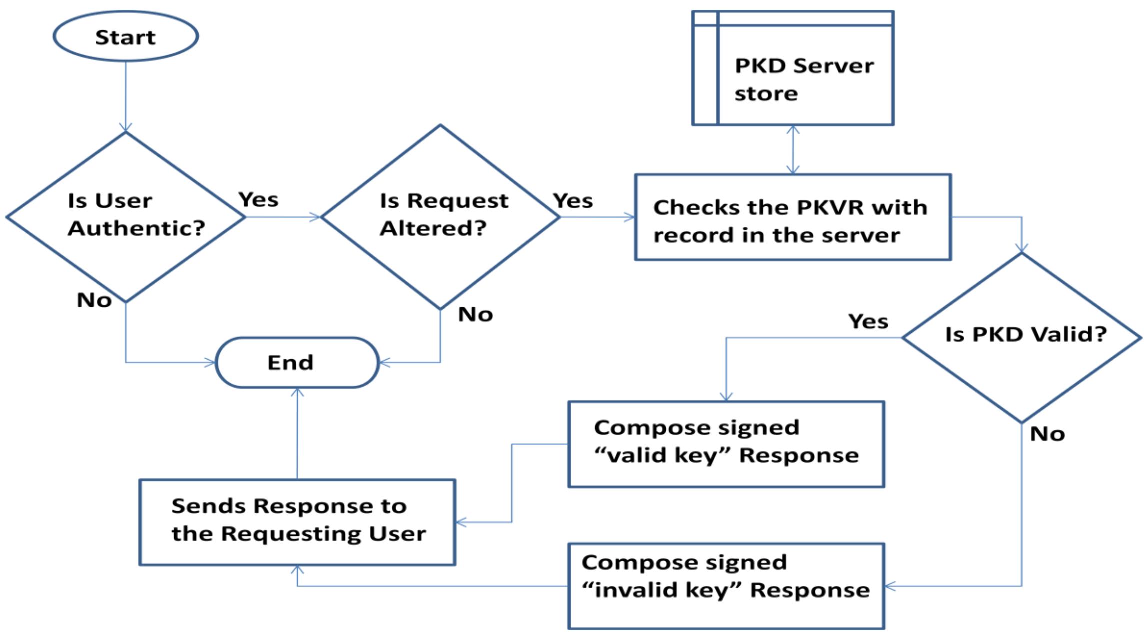


Figure 3.8: PKA Processing of a PKV Request

6. At the users' end, the ProSMS mobile app treats the PKA's response to a verification request in either of two ways depending on whether the response is positive or negative. If the public key and claimed identity is true, then the "UNVERIFIED" status of the key is changed to "NOTREVOKED". When an attempt is made at using the key, the application allows its use for secure communication. But if the key is found to be false, then the public

key status is changed to “SUSPICIOUS”. When an attempt is made at using the key, the application blocks it use and informs the user about the status of the key.

7. For the purpose of key revocation (invalidation) of a once verified (and validated) key, a PKA can also send “NOWREVOKED” information anytime a key is no longer valid for use.
8. At the users’ end, the ProSMS mobile app treat’s a PKA’s “NOWREVOKED” information as follows; the formerly “NOTREVOKED” status of a key is changed to “NOWREVOKED”. When an attempt to use the key is made, the applications blocks it use and informs the user about the status of the key. J2ME codes for implementing varying key status in the developed mobile application are as follows:

```
 if(c == compose){
int getSelInd = searchChoice.getSelectedIndex();
 ContactDetails          getConInfo          =          (ContactDetails)
 vectorArray.elementAt(getSelInd);
publicExpField21.setString(getConInfo.getContactKeySta());
if (publicExpField21.getString().equals("Unverified")){
keyStatusForm.deleteAll();
keyStatusForm.append("This user Identity Details has not been verified! "
 + "Please contact Your PKA about this User");
 keyStatusForm.append (" Do you want to send a Public Validation Request
 Now?");
keyStatusForm.addCommand(yes2);
keyStatusForm.addCommand(back2);
display.setCurrent(keyStatusForm);

}
else if (publicExpField21.getString().equals("Nowrevoked")){
keyStatusForm.deleteAll();
 keyStatusForm.append("This user's public key has been revoked and thus SMS
 " + "commuinication with the user is not SECURED! You may need to inform
 the user about this!");
keyStatusForm.removeCommand(yes2);
keyStatusForm.removeCommand(back2);
keyStatusForm.addCommand(ok);
display.setCurrent(keyStatusForm);
}
else {
ContactName2.setString(getConInfo.getContactName());
ContactUserName2.setString(getConInfo.getContactUserName());
PhoneNo2.setString(getConInfo.getContactNumber());
publicExpField21.setString(getConInfo.getContactKeySta());
publickeyField2.setString(getConInfo.getContactPublicKey());
```

```
publicExpField2.setString(getConInfo.getContactPublicExp());
String UserMod = publickeyField2.getString();
BigInteger RSAmod1 = new BigInteger(UserMod);
String UserPubExp = publicExpField2.getString();
BigInteger RSAPubExp1 = new BigInteger(UserPubExp);
newRSApubKey = new RSAKeyParameters(false, RSAmod1, RSAPubExp1);
display.setCurrent(mainForm2);
}}
```

Steps 9 to 11 as detailed in Figure 3.7 mimics earlier steps (4 to 6) in the instance where the network providers of the communicating partners are not the same.

It is important to note that the "NOWREVOKED" information is only required once per the entire life cycle of a particular public key. In the event that a public key is deemed valid (Step 5), it is the responsibility of the PKA to send any subsequent change in the status of the public key. The user assumes that a once valid public key remains valid until such a time it receives another SMS suggesting otherwise (Step 8). The implication of this is that, resources at RPs' end are conserved. At the user's end, it does not have to repeatedly ask the same request every time, because whenever the validity status of a once valid key changes, it will be communicated by the PKA automatically. This is the main difference of public key status confirmation between PKI and the proposed SB-PKC. In PKI, users assume a key is no longer valid, except the TTP states otherwise via OCSP or CRL when users make request, but in SBC-PKC, users assume once validated public keys remain valid, until the TTP states otherwise via SMS. This way, the cause of delay and its associated consequences during key revocation-check is minimized. Furthermore, SB-PKC does this without having to store (and manage certificates either at users' or TTP end) and still retains the key escrow-free feature of PKI. Figure 3.9 shows the interaction between parties in SB-PKC and how it improves key validation by eliminating the request-response-based chatty protocol used in PKI.

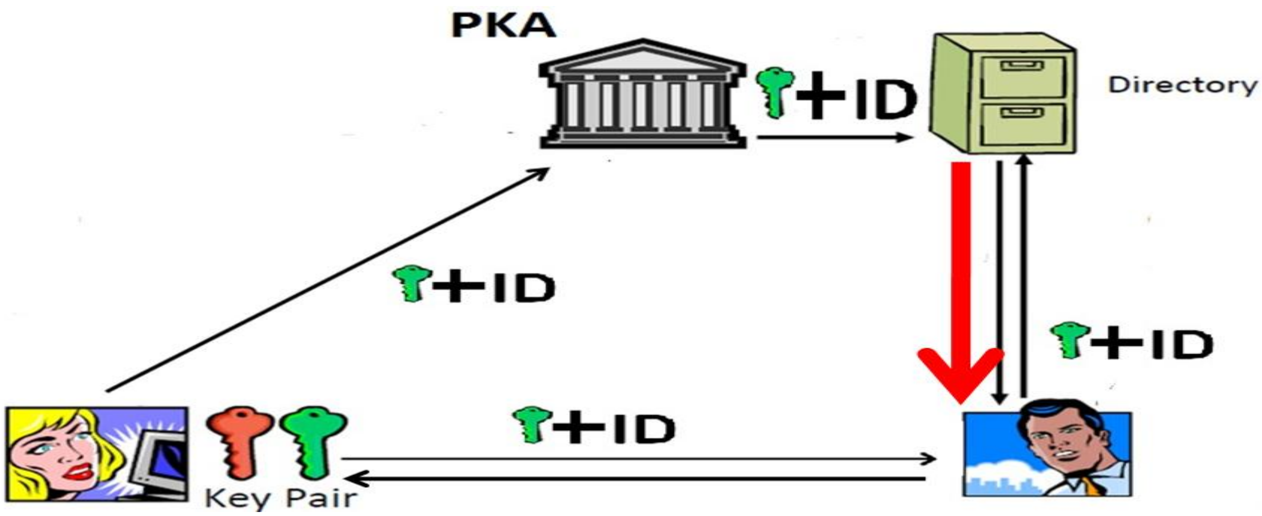


Figure 3.9: SB-PKC Components

However, it is important to note that, this approach to key revocation checking (SB-PKC) is dependent on the probability of true delivery (reliability) of the revocation data(via SMS), since the revocation is sent by the TTP (PKA). It implies that, in case of failed delivery(s), a user might not get revocation information and this will hamper the security of the scheme. SMS is however, known to have as high as 95% true delivery (Meng *et al.*, 2007). It turns out to mean that, it is only in 5% of cases where a revocation SMS is sent to users in this scheme, that they may not receive the revocation SMS and thus the security of the scheme in these instances is compromised.

3.3.6 Implementation of Pertinent Parts of the SB-PKC Scheme

Here the development of pertinent aspect of the mobile application responsible for handling key verification and validation are presented.

3.3.6.1 *Development of the Codes to Enable Key Registration with PKA*

J2ME codes are written to add secure key exchange between the user and the PKA. The entire security of SB-PKC lies on this process. This is because, since RPs rely on the PKA's response to other users' public key status, the process of obtaining the PKA's public key must be made to meet minimum security requirements. Therefore a mechanism like the DH key exchange protocol which is known to be secure against eavesdropping and MITM attacks is implemented following the protocols in section 2.1.1.2. The process is summarized as follows;

1. During the registration, a user generates a large (256 bits) random value A and a primitive modulus b. Under the influence of a random “salt” a user generates own DH key pair (public key (DHPuser) and private key (DHSuser). It sends A, b and DHPuser to the PKA via conventional SMS.
2. The PKA receives A, b and DHPuser from the PKA. It also generates its own DH key pair (public key (DHPuser) and private key (DHSuser) using A and b under the influence of randomly generated salt. It further generates a symmetric key PKAk using its private key (DHSpka) and the received public key of the user (DHPuser).
3. The PKA sends its own public key (DHPpka) value to the user.
4. The user computes its own symmetric key value USERk using its private key (DHSuser) and the received public key (DHPpka) from the PKA.

It has been shown in section 2.1.1.2 that PKAk and USERk will be equal to each other.

5. The user uses USERk as the symmetric key under the influence of Twofish symmetric cipher to encrypt its own RSA public key. The encrypted public key is sent to the PKA via SMS. The PKA uses PKAk to decrypt the user’s encrypted public key and saves it.
6. The PKA does same (as in 5) so that the user now has the PKA’s public key stored on its mobile device.

This way, the PKA and the user can successfully exchange their RSA public keys in a secure manner. J2ME code for arriving at same symmetric key (DH Key Agreement) is shown immediately underneath.

```
public BigInteger generateDHKeyPair () throws Exception {//user genertate DH parameters
          SecureRandom sr = new SecureRandom();// random generator
          p = BigInteger.probablePrime(256,sr);// parameter A, random value
          int h = 5;// parameter b
           BigInteger g = new BigInteger(Integer.toString(h));
           dhParams = new DHParameters(p, g);// A and b combined into DH Parameters
           DHKeyGenerationParameters DHKeyGenPara =
           new DHKeyGenerationParameters(sr, dhParams);
           DHKeyPairGenerator DHKeyPairGen = new DHKeyPairGenerator();
           DHKeyPairGen.init(DHKeyGenPara);
           AsymmetricCipherKeyPair dhPair = DHKeyPairGen.generateKeyPair();
            this.privateValueUser =
            ((DHPrivateKeyParameters)dhPair.getPrivate()).getX(); //User’s DH private
            key value
           if (((DHPublicKeyParameters)dhPair.getPublic()).getY() == null ||
           privateValueUser1==null ){
           throw new Exception("Generate DH keys first!" );
           }else {
           }
            publicValueUser = ((DHPublicKeyParameters)dhPair.getPublic()).getY();
            //user’s DH public key value
            return ((DHPublicKeyParameters)dhPair.getPublic()).getY();
          }
        // publicValueUser, p and g are sent to the PKA via SMS

            public BigInteger generateDHKeyPair2 () throws Exception { // PKA
            generates key pair   based on parameters received from the user
           SecureRandom sr = new SecureRandom();// random generator
           BigInteger bg = p ; // bg equated to the value of pp received
           int h = 5;
           BigInteger g = new BigInteger(Integer.toString(h));
           dhParams = new DHParameters(bg, g); same dh parameters
           DHKeyGenerationParameters DHKeyGenPara =
           new DHKeyGenerationParameters(sr, dhParams);
           DHKeyPairGenerator DHKeyPairGen = new DHKeyPairGenerator();
           DHKeyPairGen.init(DHKeyGenPara);
           AsymmetricCipherKeyPair dhPair = DHKeyPairGen.generateKeyPair();
            this.privateValueUser2 =
            ((DHPrivateKeyParameters)dhPair.getPrivate()).getX();

      publicValueUser2 = ((DHPublicKeyParameters)dhPair.getPublic()).getY();
      return ((DHPublicKeyParameters)dhPair.getPublic()).getY();
           }
      public BigInteger calculateAgreementPKA() // PKA uses this to calculate symmetric
      key
      {
      BigInteger pGF = dhParams.getP();
        EncryptionKey = publicValuePKA1.modPow(privateValueUser2, pGF);
      System.out.println (" Symmetric Key A = " + EncryptionKey);
      return publicValuePKA1.modPow(privateValueUser2, pGF);
     }
```

3.3.6.2 *Development of the codes to Enable Public Key Verification Request (PKVR)*

Users (say A) are required to send PKV request to the PKA whenever they first come in contact with another user's (B) public key via authenticated SMS; the SMS is enabled for integrity and authentication purpose. While the integrity is achieved with SHA2 hash function, the authentication is achieved via signature using the user's private key (see page 97).

3.3.6.3 *Development of the Codes for Receiving Response to PKVR*

Figure 3.10 is the flow chart diagram for receiving response to verification request. It also represents the flow of action when a PKA sends an SMS for revoking a once verified key (see page 100).

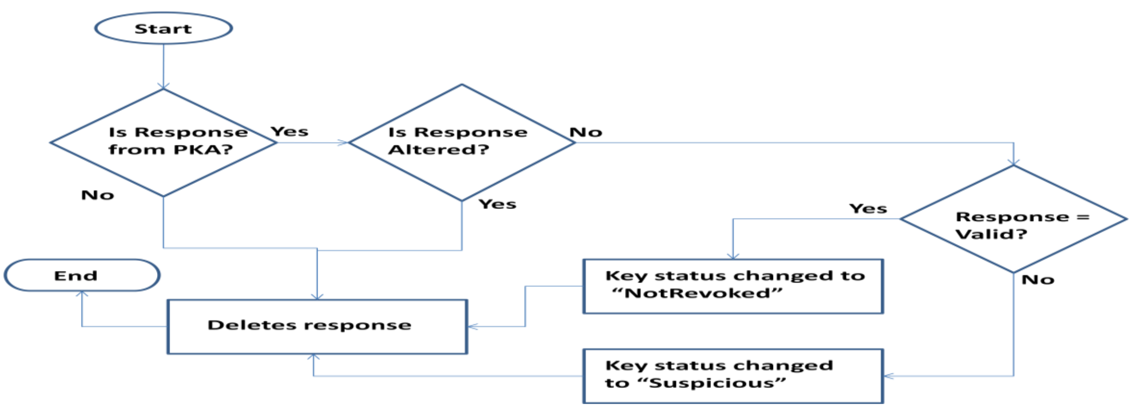


Figure 3.10: Users' Processing of a PKV Response to a PKVR

The runtime requirement for prcossesing the PKV Response data is obtained. The data obtained is compared with similar data obtained in OCSP test (see section 3.3.9).

3.3.7 Testing (Simulation) of the Application's Functionalities

The Netbeans IDE has a simulation environment that can be used to observe the functionality of J2ME applications. The simulation is done to test the behaviour of the application before it is transferred to a real environment. Plate 3.2 shows the final stage of arriving at same symmetric keys between two devices; one representing the user and the other representing the PKA on the Netbeans IDE.

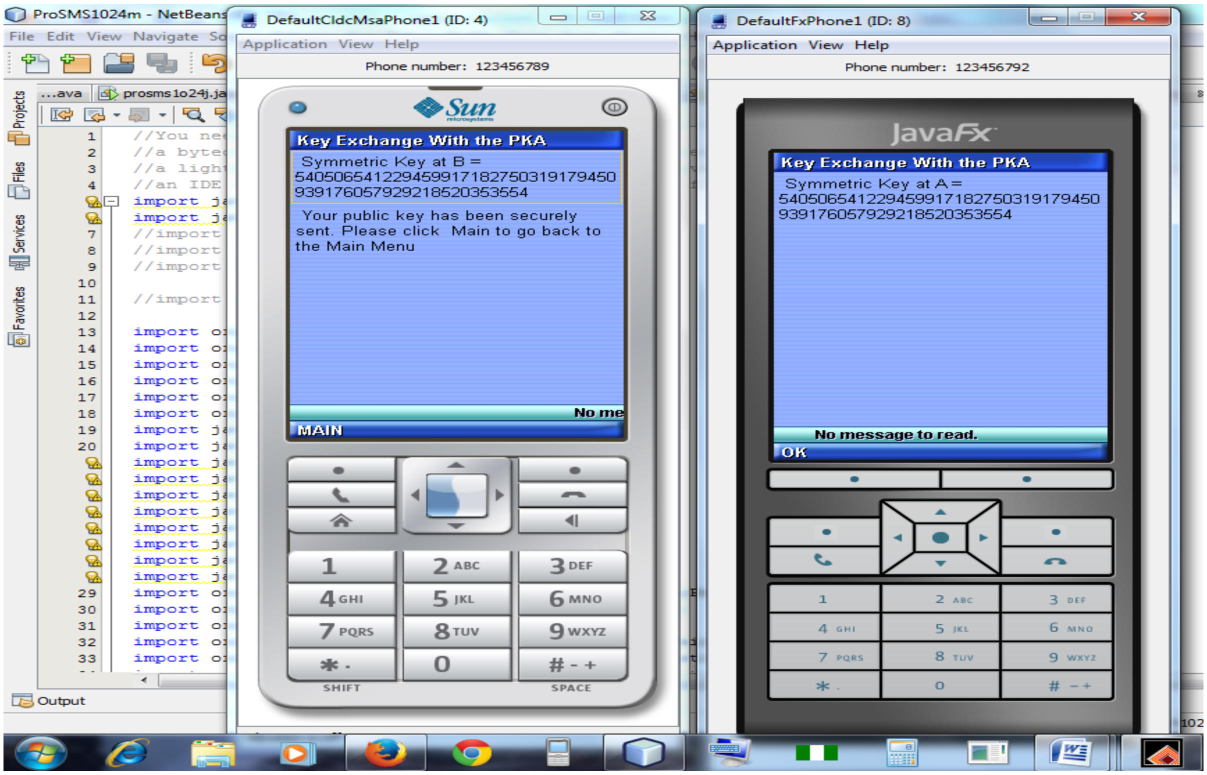


Plate 3.2: DH Symmetric Key Agreement on Netbeans IDE

3.3.8 Testing (on Real Device) of the Application's Functionalities

For testing on real devices; Tecno (T9 model) phone is used as the user device (Plate 3.1).

The performance tests are divided into two; test on secure SMS messaging and test on SB-PKC Scheme.

3.3.8.1 *Test on Secure SMS Messaging*

These tests here are carried out with two intents; (1) to ascertain that the developed application functions as required and (2) to obtain the run time requirements of the cryptographic processes. Cryptosystems are often evaluated on the basis of their cost (resource) implication. Thus, the run time requirement of each of the cryptographic process and the data size used to arrive at the obtained values are as shown in Table 4.1 in chapter four.

Tests conducted include run time values for Twofish key generation, Twofish encryption and decryption of SMS, RSA key generation, RSA encryption and decryption of Twofish key, RSA signing and verification of digital signatures. A snapshot of the test device after RSA key generation is shown in Plate 3.3.

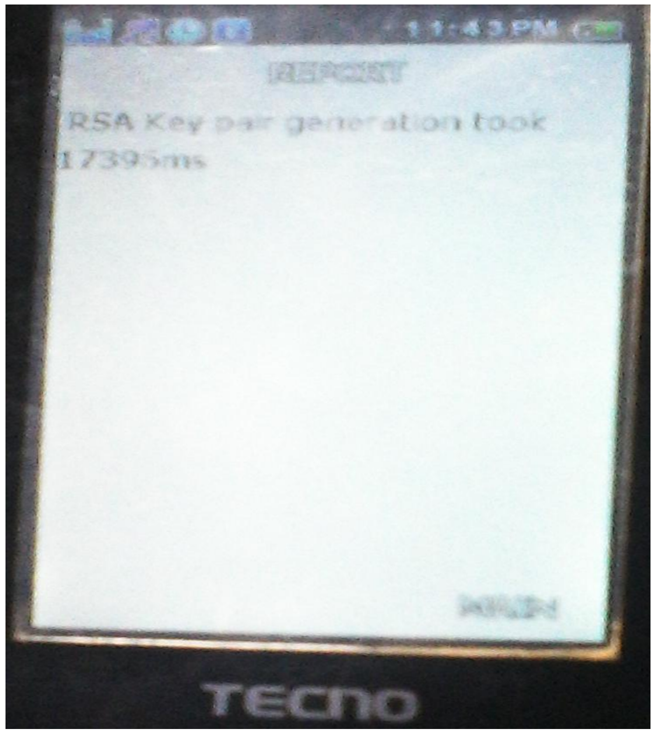


Plate 3.3: RSA Key Genertion

3.3.8.2 *Test on SB-PKC Scheme*

The tests here are also carried out with two intents; (1) to ascertain that the developed J2ME codes for handling PKV (SB-PKC) functions as detailed in Figure 3.7 and (2) to obtain overheads requirements of the SB-PKC that can be subsequently compared with similar measurements in PKI (section 3.3.9).

1. **DH Key Parameter Exchange**

Test for exchanging DH key parameters (via traditional SMS) used for generating a symmetric key that is subsequently used to exchange public key between the user and the PKA during registration of users of the scheme. The test was carried on two mobile devices. The test was to ascertain that the developed application can be used to perform DH key agreement/exchange.

2. RSA Key Exchange using DH-derived Symmetric key

The application was also used to test that the generated symmetric key from the DH key parameter exchange can be used as an encryption key to encrypt RSA public keys. The test was also carried out on two mobile devices.

3. Sending of PKV Request

Test for PKV request between two mobile phones was carried out. The test was to ascertain the developed application can be used to compose a PKV request and securely (using signature) exchange it via SMS. The data size used for this test consists of user's data and a signature.

a. User's data (total equals 378 bytes) consisting of the following;
   i. Name of the User. A maximum of 20 characters is used.
   ii. Username of the User. A maximum of 30 characters is used.
   iii. User's Phone number (international format). A maximum of 14 characters is used.
   iv. Public key modulus and Exponent values. For 1024 bytes RSA key, the key modulus is a large integer of 309 digits and the exponent is a 5 digit prime integer.
b. A signature of the entire data (in a above) is 128 bytes.

Thus, the entire PKV request data is about 506 bytes (≈ 5 SMS payloads).

4. Receiving of PKV Response

Test for PKV response between two mobile phones was carried out. A signed PKV response was sent from a mobile device to the user device (Tecno). The test was to ascertain the developed application can be used to receive response (key verification and key validation) from the PKA via SMS. The average run time to process (confirm for authentication and integrity) the response is taken for twenty (20) instances. The average time it took to process to PKV Response is 40ms. The data size used for this test consists of user's data, public key status data, and a signature;

a. Users data (total equals to 64 bytes); name, username and phone number.
b. Another component of the PKV response is the key status data: "NOTREVOKED", "SUSPICIOUS" or "NOWREVOKED" each equal to 10 bytes.
c. Then a signature (128 bytes) of the combination of (a) and (b) above, using the PKA's private key.

Thus, the entire PKV Response data is about 202 bytes (≈2 SMS payload).

3.3.9 Development of Codes for Public Key validation using PKI

The application is developed further to accommodate public key validation using PKI. The aim of this is to enable a comparison between data obtained for SB-PKC earlier (section3.3.4) and PKI.

3.3.9.1 *PKI (OCSP) Test*

As detailed in section 2.2.2.1**,** key validation in PKI can be obtained via OCSP or CRL. Thus, public key validation is perfomed by sending OCSP request to an OCSP responder and receiving a response from it.

1. **OCSP Request data size**

OCSP request data size varies depending on the parameters used. The RFC standards stipulate the "GET http" method should be used for OCSP request data less than or equal to 255 bytes and the "POST http" method be used for request data greater than 255 bytes. The GET method is used in this work as the OCSP request data is 143 bytes. The OCSP request data used is taken from CryptoSys (2004). Twenty (20) OCSP request instances are sent to Comodo CA OCSP responder/server. It took an average of 5000ms to receive a response from the OCSP server.

2. OCSP Response data size

OCSP response data size also varies depending on the OCSP responder. The OCSP response data size obtained from the test (from Commodo's OSCP responder) is 472 bytes (Plate 3.4).

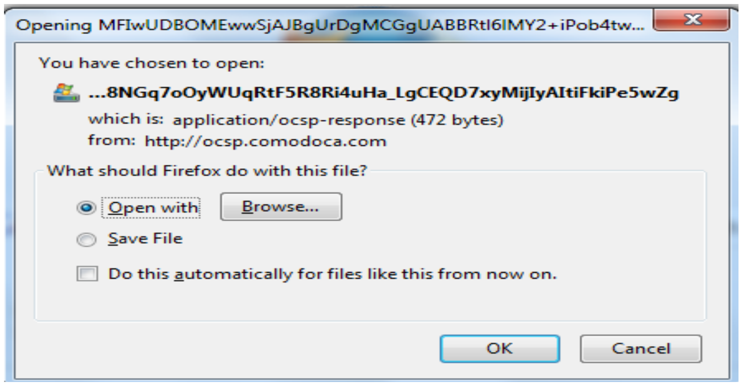


Plate 3.4: OCSP Response Data

3.3.10 Validation by Comparison between PKI and SB-PKC

In this section, the obtained data for key validation in SBC-PKC is compared with that of the PKI. Thus, the obtained result in this work is compared to PKI's OCSP data.

3.3.10.1 *Basis for Comparison*

Both the developed SB-PKC and PKI delegate the fuction of key validation/revocation to TTP entity; the decision whether a key is still valid or not is done by the TTP on behalf of the RPs. RPs only make use of the result or information provided by the TTP in determining whether a public key status is "valid" or "revocked" at any particular time.

Since PKI has been in existence long before now (advent of SB-PKC), SBC-PKC must also be made to use similar standards as used in PKI so that appropriate comparison can be made.

RFC 2650 and RFC 5019 standards stipulates conditions that need to be met for valid key status request and response. They also included features that must be included for the security of OCSP request and response information. In particular, they stated that signature and nonce/timestamp are two important components that may be used to prevent against attacks on OCSP requests and Responses. While at the request end, signature and nonce are optional and not recommended, at the responder end, signature is a must but nonce/timestamp is optional. Signature is meant to prevent impersonation (attack) of TTP. Nonce/timestamp is meant to prevent replay attacks; in which case a previous “valid key” response is replayed to a request of a key’s status that has been revoked afterwards. It is important to note at this point that for the purpose of key revocation, in SBC-PKC, a request is not required. Furthermore, the key status or revocation response can only take place once; thus, replay attack does not apply. Thus, for the two schemes to be on equal footing, their revocation status data must be equipped with digital signature capability.

Twenty instance of OCSP request were sent on the same mobile device (Tecno T9) used for obtaining key status data in SB-PKC (3.3.8.2.). The average time it takes the test device to obtain key status response using PKI (via OCSP) is 5000ms.

CHAPTER FOUR

RESULTS AND DISCUSSIONS

4.1 Introduction

In this section, the results obtained in steps 4, 6 and 7 of the methodology are presented and analyzed. The entire code of the developed application can be found in Appendix A on page 85.

4.2 Developed Application's Performance Based on Objective One

The first objective of this work is to add secure communication requirements to SMS via cryptography. However, cryptographic operations come at a price; they are generally considered to be computationally intensive. Thus, it is usual to test the performance of cryptographic operations as a function of their execution or run time values, hence the values in Table 4.1. Table 4.1 shows the run time (in ms) for cryptographic-related operations of the messaging application. These values represent a measure of *extra* work that the device (Tecno T9) running the developed application would undergo to add secure communication functionalities to traditional SMS messaging.

Table 4.1: Runtime Average of ProSMS' Cryptographic Operations

ProSMS Operation	Runtime (ms)
Twofish key (128 bits) Generation	16.3
Twofish Encryption (120 bytes data size)	71.9
Twofish Decryption (120 bytes data size)	69.3
RSA (1024 bits) Key Generation	**26033**
RSA Encryption (33 bytes data size)	3.3
RSA Decryption (128 bytes data size)	1.9
RSA Signing (120 bytes data size)	68.8
RSA Verification (128 bytes data size)	48

From Table 4.1, it is important to note that, with the exception of RSA Key Generation, all operations on the test device took less than 1s. This implies the developed application's cryptograhic-related operational processes are mild on the test device as it takes less than a second to perform them. RSA key generation is comparatively high (26033ms). However, the fact that in a PKC-based cryptosystem, asymmetric algorithms are usually generated once and saved for subsequent use. Thus, the long execution duration (and likely high consumption of other mobile phone's resources like battery, etc) will happen only once. Thus, despite this "high" execution time for RSA key generation, the solution is still acceptable for use on mobile devices.

4.3 Developed Application's Performance Based on Objective Three

Here, the performance of the SB-PKC scheme is compared with that of PKI. A very important metric of performance is the time it takes to arrive at a public key's validity status, as this time represents latency/delay experienced in the two schemes. Another important metric of performance

Table 4.2: Comparison between SBC-PKC and PKI

Parameter	PKI	SBC-PKC
Latency (during key validation)	5000ms (Recurrent)	40ms (once)
Size of Request Data	143 bytes (Recurrent)	-
Size of Response Data	472 bytes (by User)	202 bytes (by TTP)
Storage space Required per User Key	4325 bytes (Belshe, 2011)	-
Explicit Validation	4325 bytes (Belshe, 2011)	506 bytes

is the size of data used in both schemes as this informs the amount of storage that will be required if the data are to be stored. And if the data required is to be moved from one point to another, the size is equally important as it points to both the financial and resource-cost (overhead) implications of the scheme. These information are contained in Table 4.2

The 40ms run time required for the test device to process a validation response from the PKA is the time it takes the device to "wait" or "suspend other operations" in other to attend to a user's public key validation operation. This value for key revocation confirmation in SB-PKC scheme is the latency a user experience when a revocation data is delivered to his device. This latency is compared with the latency of an average of 5000ms required to obtain OCSP response in PKI. The most significant implication of this is the fact that, while in SBC-PKC, this revocation checking exercise will only occur once per public key life cycle (when a public key is finally revoked), revocation checking will be required as many times a RP intends to use another user's public key for secure communication to the owner of the key in PKI. Other indirect implications is that system resources like mobile device battery energy, memory and CPU operations would be expended on those "many" occasions in PKI and only once in SBC-PKC.

Secondly, both schemes are compared on the basis of the size of validation request data. It can be seen from Table 4.2 that for the purpose of key status confirmation; while PKI requires that a user sends some 143 bytes of data to the OSCP server before validation response is received, in SBC-PKC a user is not required to send request data. Thus, the SBC-PKC reduced to the minimum, the amount of time RPs' devices have to "hit the wire" for the purpose of key revocation checking; which is one of the main design objectives of RFC 5019 (to handle OCSP on mobile devices) as an improvement on RFC 2560.

Thirdly, the size of the response data of the two schemes is compared. Here too, SBC-PKC has a lower response data size. More importantly, in SBC-PKC, the TTP is responsible for sending this response data, unlike in PKI where it is the user that pays for the download of the response data.

Table 4.3: Comparison of PKV Schemes based on limitations.

Limitation	**PKI**	**WoT**	**ID-PKC**	**CL-PKC**	**SC-PKC**	**SGC-PKC**	**CB-PKC**	**SB-PKC**
Key escrow	×	×	✓	×	×	×	×	×
No Revocation	×	✓	✓	✓	✓	✓	×	×
Third Party	✓	×	✓	✓	✓	✓	✓	✓
Storage Space	✓ (at users&CA)	✓ (at users)	×	×	×	✓ (at users)	✓ (at CA)	×
Latency (during key status-check)	✓ (at sender)	×	×	×	×	×	✓ (at receiver)	×
Explicit Verification	✓	✓	×	×	×	×	×	✓
Cumulative	4	3	3	2	2	3	3	2

Fourthly, the storage requirements of the two schemes are compared. In PKI, a proof of ownership of a public key (certificates) is required to be stored on users' devices. The size of these certificates varies from vendors but increases significantly with the "depth" of the certificate used. Thus, certificates with lower depths would have been preferred, but they are more expensive (Belshe, 2011). SBC-PKC like many other PKV schemes does not use certificates for binding identities and public keys. Thus, significant storage space is conserved.

Fifthly, the cost of performing key verification is compared. Both schemes are explicit PKV schemes, but they go about key verification in different ways. While in PKI, certificates are downloaded (from TTP server) and checked for authenticity, in SB-PKC PKV requests are sent to

TTP. Comparing with Belshe (2011) example, while an internet data of 4325 bytes is required to be expended (at the users' end) in PKI, in SBC-PKC a 5-SMS payload data (PKV request) is required.

Finally, the proposed scheme is compared with earlier PKV schemes, to show its comparative advantages over other existing PKV schemes. Thus, the design objectives for robust PKV management scheme as presented in section 2.2.3 is used to create a table (Table 4.3) showing the limitations of all the reviewed PKV schemes and the proposed SB-PKC scheme.

From Table 4.3, it can be seen that, the limitations of SBC-PKC is 2 while in PKI, it is 4. It is however, noteworthy that some of the limitations carry more weight than the other. For example ID-PKC has 2 limitations while PKI has 4. But the limitations of ID-PKC on key escrow and absence of key revocation checking mechanism are so great that PKI would still be preferred to it. Also some of the limitation are inter-related; for example latency during revocation check are not present in schemes like ID-PKC, CL-PKC etc. because there are no revocation checking mechanism in these schemes in the first place.

4.4 Summary

The results obtained from the test experiments of steps 4, 6 and 7 are presented. The results on hand shows that secure communication requirements are added to SMS messaging with very little extra overhead as indicated by the <1s runtime values of the cryptographic processes. On the other hand, the result also shows the advantages of developed SB-PKC over the existing PKI PKV scheme.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATION

5.1 Introduction

This chapter presents the conclusion and limitations encountered during the course of this research. Areas of future research have also been recommended.

5.2 Conclusions

This research developed a secure messaging application for mobile phones using hybrid cryptography. The four secure communication requirements of confidentiality, integrity, authentication and non-repudiation have been added to SMS messaging using a combination of symmetric (Twofish), asymmetric (RSA) and hash function (SHA256) cryptographic ciphers. An improved approach (SB-PKC) to public key verification (SB-PKC), where a TTP (PKA) furnishes RPs with other users' public key status (for verification and validation) information via authenticated SMS was also implemented. Thus, the aim of this research which is to develop a secure SMS messaging application suitable for mobile phones has been achieved. The developed solution has been shown to implement all its security additions with minimal overhead as indicated by the very low time (<1s) it requires for its cryptographic operations. Finally, the SB-PKC PKV scheme is shown to have comparatively lower latency and overhead-cost requirement for public key validation when compared with PKI. SB-PKC has latency of 40ms against 5000ms in PKI. Furthermore, SB-PKC reduced to the minimum (0), number of internet roundtrips requirements for key validation purpose, thus conserving bandwidth and other associated resources like battery consumption, etc.

5.3 Limitations

This work is limited in the following ways;

1. The probability of compromise of the proposed SB-PKC PKV scheme, which is dependent on the probability of true SMS delivery, is 95%. This implies that the SB-PKC can be compromised in 5% of instance of failed delivery of revocation messages sent as SMS.
2. This work only considers a comparison of resource-cost requirements at users' ends and not at the TTP ends.

5.4 Recommendations for Further Works

Further works should consider the following areas:

1. The probability of true delivery can be improved upon with a view to minimize the probability of compromise of revocation information. Other options that can be exploited include the use other push-based communication protocols like Session Initiation Protocol (SIP) etc rather than SMS.
2. Further works can also consider comparison of resource requirements at the TTP ends between SB-PKC and PKI with a view to see which of the schemes more efficient at the TTP end.

REFERENCES

Al-Bakri S. H. and Mat Kiah M. L. (2010) "A Novel Peer-to-Peer SMS Security Solution using a Hybrid Technique of NTRU and AES-Rijndael." Scientific Research and Essays **5**(22): 3455-3466.

Alexander U., Ralph H., Peter H. and Georg C. (2011). Investigating the OpenPGP Web of Trust. Computer Security – ESORICS 2011 (pp. 489 – 507). Springer Berlin Heiderberg.

Anuar N. B., Kuen L. N., Zakaria O., Gani A and Wahab A. W. A. (2008). GSM mobile SMS/MMS using public key infrastructure: m-PKI. WSEAS Trans. Comput., 7: 1219-1229.

Anuja R (2011). How to Send and Receive SMS in J2ME. http://anujarosha.wordpress.com/2011/07/20/how-to-send-and-receive-sms-in-j2me/. Retrieved 04-06-2014.

Azubuike C. and Obiefuna O. (2014). Wireless Communication: The Impact of GSM on the Economic Lives of the Nigerian Rural Users. Journal of Educational and Social Research, Vol. 4**,** issue 79.

Balachandra M., Krishna P. and Shashank S. (2012). Wireless Public Key Infrastructure for Mobile Phones. International Journal of Network Security & Its Applications (IJNSA), Vol.4, No.6.

Belshe M. (2011). Certificate Validation Example: Facebook. "https://www.belshe.com/2011/04/20/ certificate-validation-example-facebook/" Retrieved 04-06-2014.

Berggren V. S. and Winnem Z. (n.d). ECSMSMIDlet. http://people.dsv.su.se/~it03_vso/umt/project. Retrieved 04-12-2014.

CA/Browser Forum (2013). Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates, v.1.1.6. https://cabforum.org/wp-content/uploads/Baseline_Requirements_V1_1_6.pdf. Retrieved 04-06-2014.

Croft, N. J. and Olivier M. S. (2005). Using an Approximated One Time Pad to Secure Short Messaging Service (SMS). Proceedings of the Southern African Telecommunication Networks and Applications Conference (SATNAC).

CryptoSys (2004). PKI OCSP_MakeRequest. **"**http://www.cryptosys.net/pki/manpki/pki_OCSP_MakeRequest.html**".** Retrieved 04-06-2014.

De Santis A., Castiglione A., Cattaneo G., Cembalo M., Petagna F. and Petrillo U. F. (2010). An Extensible Framework for Efficient Secure SMS. Complex, Intelligent and Software Intensive Systems (CISIS). 2010 International Conference on IEEE. 843-850

Diffie W. and Hellman M. E. (1976). "New Directions in Cryptography." Information Theory, IEEE Transactions on 22(6): 644-654.

Enrique C. O. (2003). The MIDP 2.0 Push Registry. http://www.oracle.com/technetwork/systems/index-156665.html Retrieved 04-06-2014

Eric L. (2013). Alternatives. http://proguard.sourceforge.net/index.html#/alternatives.html. Retrieved 04-06-2014.

Fujishiro T., Akane S., Yoko K., Tadashi K. and Kenichi O.(2010). Development of Hi-Speed X. 509 Certification Path Validation System. 2010 IEEE 24th Inrenational Conference on Advanced Information Networking and Applications Workshops (pp 269 – 274).

Gentry C. (2003). Certificate-based Encryption and the Certificate Revocation Problem. Advances in Cryptology-EUROCRYPT 2003, Springer Journal: 272-293.

Gupta P. (n.d). Short Message Service: What, How and Where? Wireless Developer Network. http://www.wirelessdevnet.com/channels/sms/features/sms.html/ Retrieved 02-12-2014

Gutmann P. (2013). Engineering Security. 2013. Book draft. "http://www. cs. auckland. ac. nz/~ pgut001/pubs/book.pdf". Retrieved 04-06-2014

Helme S. 2014. Enabling Certificate Revocation Checks in Google Chrome. https://scotthelme.co.uk/certificate-revocation-google-chrome/ Retrieved 02-12-2014

Hou M., Qiuliang X. and Fengbo L. (2012). An Efficient Certificate Revocation and Verification Scheme from Multi-Hashing. Journal of Computers **7**(6): 1437-1444.

http://bouncycastle.org. Retrieved 01-04-2013.

http://netbeans.org. Retrieved 01-04-2013.

http://proguard.com. Retrieved 01-04-2013.

Isabona J. (2014). Maximising Coverage and Capacity With Qos Guarantee in GSM Network by Means of Cell Cluster Optimization. International Journal of Advanced Research in Physical Science, **1,** 44-55.

Ireland D. (2010). "RSA Algorithm." http://di-mgt.com/rsa_alg.html. Retrieved 04-06-2014.

Kessler G. C. (2014). An Overview of Cryptography. "http://www.garykessler.net/library/crypto.html." Retrieved 02-12-2014

Khiyal M. S., Khan A. and Shehzadi, E (2009). SMS Based Wireless Home Appliance Control System (HACS) for Automating Appliances and Security. Issues in Informing Science and Information Technology, **6,** 887-894.

Kumar A., Antony J., Gaurav K., Hari V., Hoa L., Ning C., and Rick A. (2013). White Paper: Elliptic Curve Cryptography (ECC) Certificates Performance Analysis. http://symantec.com/wp_ecc.pdf. Retrieved 04-06-2014.

Lai J., Weidong K. and Kefei C. (2011). Self-Generated-Certificate Public Key Encryption Without Pairing and its Application. Information Sciences 181 (2011) 2422–2435.

Langley A. (2012). Revocation Checking and Chrome's CRL. "http://www.imperialviolet.org/2012/02/05/crlsets.html." Retrieved 24-02-2014.

Lee, C. C., Hwang M. S. and Liao I. E. (2011).. "On the Security of Self-Certified Public Keys." International Journal of Information Security and Privacy (IJISP) **5**(2): 54-60.

Lisonek D. and Drahansky M. (2008). SMS Encryption for Mobile Communication. IEEE International Conference Security Technology (SECTECH), pp 198-201.

Lu Y. (2011). Further Observations on Certificate-base Encryption and its Generic Construction from Certificateless Public Key Encryption. Scientific Research and Essays Vol. 6(18), pp. 3900-3913.

Medani, A., A. Gani, O. Zakaria, A. A. Zaidan and B. B. Zaidan (2011). Review of Mobile Short Message Service Security Issues and Techniques Towards the Solution. Scientific Research and Essays 6(6): 1147-1165.

Meng, X., P. Zerfos and V. Samanta (2007). Analysis of the Reliability of a Nationwide Short Message Service. 26th IEEE International Conference on Computer Communications (INFOCOM), pp. 1811-1819.

Mutton P. (2014). Heartbleed: Revoke! The Time is Nigh! "http://netcraft.com/archives/2014/04/15/revoke-the-time-is-nigh.html" Retrieved 04-12-2014.

Noroozi E., Salwani M. D. and Sabouhi A. (2014). Enhancing Secured Data Hiding Using Dynamic Digital Signature for Authentication Purpose. Jurnal Teknologi (Sciences & Engineering) 68(2): 39–44.

Pereira G. C. C. F., Mateus A. S. S., Bruno T. O., Marcos A. S., Paulo S.L.M.B., Díntia B. M., and Wilson V. R. (2013). SMSCrypto: A lightweight cryptographic framework for secure SMS transmission. The Journal of Systems and Software. Elsevier 86(3), 693 – 706.

Pujitha S. and Mallu B. V. (2013). SMS Based Mobile Banking. International Journal of Engineering Trends and Technology (IJETT), 4(4), 1211 - 1220.

Rao M. Mobile Africa Report (2011): Regional Hubs of Excellence and Innovation. MobileMonday. “http://www.mobilemonday.org” Retrieved 11-08-2013.

Ray S. and Biswas G. P. (2011). Design of Mobile-PKI for Using Mobile Phones in Various Applications. IEEE International Conference on Recent Trends in Information Systems (ReTIS), pp. 297 – 302.

Ray S. and Biswas G. P. (2013). Design of Mobile Public Key Infrastructure (M-PKI) Using Elliptic Curve Cryptography. International Journal on Cryptography and Information Security (IJCIS), 3(1), 25 – 37.

Saxena, N. and Payal A. (2011). "Enhancing Security System of Short Message Service for M-Commerce in GSM." International Journal of Computer Science & Engineering Technology (IJCSET) 2(4).

Schneier B. (1996). "Applied cryptography." 2nd ed., John Wiley and Sons, Inc., New York.

Schneier B., Kelsey J., Whiting D., Wagner D., Hall C., Ferguson N., Kohno T. & Stay M. (2000). The Twofish Team’s Final Comments on AES Selection. *AES round*, Vol. 2.

Sharma P. K. (2009). Short-Lived Certificates as a Mobile Authentication Method. “http://nordsecmob.aalto.fi/en/publications/theses_2009/_sharma.pdf” Retrieved 04-12-2014.

Stallings W. (2004). Computer Networking with Internet Protocols and Technology. Pearson Prentice Hall, Upper saddle River, NJ 07458.

Stark E, Huang L. S., Israni D., Jackson C. and Boneh D. (2012). The Case for Prefetching and Prevalidating TLS Server Certificates. Proceedings of the 19th Network and Distributed System Security Symposium.

Statista (2015). http://www.statista.com/ Retrieved 04-02-2015.

Stoner E. D (2013). PGP Messaging; Elliptic Curve PGP implementation for SMS. Cryptography and Computer Network Security (ECE 646), George Mason University.

Suresh K., Venkataramana K. and Padmavathamma M. A. (2014). Threshold Public Key Cryptosystem to Secure SMS in the Mobile Devices. International Journal of Engineering Trends and Technology (IJETT) 9 (7): 333 – 341.

Toorani M. (2011). Certificateless PublicKey Cryptography. “http://folk.uib.no/mto079/P02.pdf” Retrieved 04-12-2014.

Toorani M. and Beheshti A. A. (2008). A Secure SMS Messaging Protocol for the M-Payment Systems. Proceedings of the 13th IEEE Symposium on Computers and Communications(ISCC), pp.700-705.

Topalovic, E., B. Saeta, L. Huang, C. Jackson and D. Boneh (2012). Towards Short-lived certificates. Web 2.0 Security and Privacy.

Voarse Zachary (2009). OpenPGP for Complete Beginners. http://zacharyvoase.com/2009/08/20/openpgp/ Retrieved 04-12-2014.

Wiesner, S. (2013). Simple PKI Innovative Internet Technologies and Mobile Communications (IITM) 83.

Yuan Michael Juntao, 2004. Enterprise J2ME: Developing Mobile Java Applications. Pearson Prentice Hall, Upper saddle River, NJ 07458.

Zhao S., Aggarwal A., and Liu S. (2008). Building Secure User-to-User Messaging in Mobile Telecommunication Networks. Wireless Telecommunication Symposium, WTS, 2008.

APPENDIX A

ProSMS Source Code

```
//You need the folowing to run this J2ME code
//a bytecode obfuscateor like proguard download free at http//:www.proguard.org
//a lightweight crypto package like bouncycastel available free at http//www.bouncycastele.org
//an IDE environment like Netbeans 7.0 available free at http//www.netbeans.org
//png images for prosmslogo1 and accesslogo
//@ durojaiyeluqman@yahoo.com +2348062568159

import org.bouncycastle.crypto.*;
import org.bouncycastle.crypto.params.*;
import org.bouncycastle.crypto.engines.*;
import org.bouncycastle.crypto.encodings.*;
import org.bouncycastle.crypto.modes.*;
import org.bouncycastle.crypto.paddings.*;
import java.math.BigInteger;
import java.security.SecureRandom;
import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
import org.bouncycastle.crypto.digests.SHA256Digest;
import org.bouncycastle.crypto.engines.RSAEngine;
import org.bouncycastle.crypto.generators.RSAKeyPairGenerator;
import org.bouncycastle.crypto.params.RSAKeyGenerationParameters;
import org.bouncycastle.crypto.params.RSAKeyParameters;
import org.bouncycastle.crypto.params.RSAPrivateCrtKeyParameters;
import org.bouncycastle.crypto.signers.PSSSigner;
import org.bouncycastle.util.encoders.Hex;
import org.bouncycastle.util.encoders.Base64;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.wireless.messaging.*;
import javax.microedition.rms.*;
import java.io.*;
import java.util.Vector;
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.lcdui.Choice;
import javax.microedition.lcdui.ChoiceGroup;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Item;
import javax.microedition.lcdui.ItemCommandListener;
import javax.microedition.lcdui.ItemStateListener;
import javax.microedition.lcdui.TextField;
import javax.microedition.midlet.MIDlet;
import javax.microedition.rms.RecordComparator;
import javax.microedition.rms.RecordEnumeration;
import javax.microedition.rms.RecordFilter;
import javax.microedition.rms.RecordStore;
import javax.microedition.rms.RecordStoreException;
import javax.microedition.rms.RecordStoreNotOpenException;
import org.bouncycastle.crypto.generators.DHKeyPairGenerator;

public class prosms_1 extends MIDlet implements RecordComparator, RecordFilter,
ItemCommandListener, ItemStateListener, CommandListener, MessageListener {
```

```
private Form f1 = new Form("GENERATE NEW RSA KEY PAIR?");
private Form f2 = new Form("GENERATING RSA...");
private Form f3 = new Form("ABOUT");
private Form f4 = new Form ("AES GENERATED");
private Form f5 = new Form ("RSA GENERATED");
private Form f7 = new Form ("Secure SMS Components");
private Form signedRequestForm = new Form ("SIGNED PKV REQUEST");
private Form signedResponseForm = new Form ("SIGNED PKV RESPONSE");
private Form f8 = new Form ("RESULT DECRYPTION");
private Form f9 = new Form ("CONFIRM IDENTITY FROM PKA");
private Form f11 = new Form ("GENERATING RSA...");
private Form f15 = new Form ("FINAL MESSAGE");
private Form RUNTIME_MESSAGING  = new Form ("REPORT");
private Form reportForm2 = new Form ("REPORT");
private Form mainForm = new Form("KEY EXCHANGE");
private Form mainForm2 = new Form("SEND SMS");
private Form sendPKVRForm = new Form("Send PKV Request");
private Form RevocationStatusForm = new Form("Revocation Status");
private Form mainForm3 = new Form("Got a User Public Key Details");
private Form mainForm3a = new Form("Got a Revocatiom mesage");
private Form keyXchangeForm3 = new Form("Key Exchange With the PKA");
private Form mainForm4 = new Form("Got a Secured SMS");
private Form mainForm4e = new Form("Got Revocation Response");
private Form mainForm4e4 = new Form("Check status");
private Form mainForm4f = new Form("Got PKV Response");
private Form mainForm9 = new Form("Got a REVOCATION SMS");
private Form mainForm5 = new Form("SEND SECURE SMS");
private Form keyStatusForm = new Form("User's Key Status");
private Form mainForm7 = new Form("The real Message");
private Form testOCSPform = new Form("OCSP Check");
private Form OCSP_MEASURE  = new Form("OCSP Result");
private Form SavedSMSForm  = new Form("SAVED SMS");
private Form setPasswordForm;
private Form checkPasswordForm;
private Form changeContactForm;
private Form searchForm;
private Form searchForm2;
private Form addContactForm;

Command Cont1 = new Command("Continue", Command.STOP, 3);
Command Cont2 = new Command("Continue", Command.STOP, 3);
Command Cont3 = new Command("Continue", Command.STOP, 3);
Command Cont4 = new Command("OK", Command.STOP, 3);
Command Cont6 = new Command("OK", Command.STOP, 3);
Command Cont5 = new Command("DecryptKeyString", Command.STOP, 3);
Command Cont7 = new Command("TestHash", Command.STOP, 3);
Command cmPOST = new Command("OCSP_POST", Command.STOP, 3);
Command cmGET = new Command("OCSP_GET", Command.SCREEN, 1);
Command yes2 = new Command("Yes", Command.OK,0);
Command ok = new Command("Okay", Command.OK,0);
Command ok4 = new Command("Continue", Command.OK,0);
Command choiceChoose = new Command("Go", Command.OK,0);
Command rsaSave= new Command("SaveKeys", Command.HELP,0);
Command saveSMS2 = new Command("SaveSMS", Command.HELP,0);
Command rsaprivRetrieve= new Command("ReqRSABoth", Command.OK,1);
Command rsapubRetrieve= new Command("ReqRSAPub", Command.EXIT,0);
Command back = new Command ("MAIN", Command.BACK,0);
Command back4 = new Command ("Go back", Command.BACK,0);
Command discard = new Command ("DISCARD", Command.BACK,0);
Command send2 = new Command ("SEND SMS", Command.BACK,0);
```

```
Command back2 = new Command ("NO", Command.BACK,0);
Command genaes = new Command ("YES", Command.ITEM,0);
Command genrsa = new Command ("GENRSA", Command.SCREEN,0);
Command goagain = new Command ("MOVE", Command.HELP,0);
Command encrypt = new Command ("ENCRYPT", Command.SCREEN, 0);
Command encryptSMS = new Command ("Add Security", Command.SCREEN, 0);
Command encryptSMS1 = new Command ("Check DHKey Encrypt", Command.SCREEN, 0);
Command signPKVRSMS = new Command ("SignRequest", Command.SCREEN, 0);
Command signRSRSMS = new Command ("SignResponse", Command.SCREEN, 0);
Command decrypt = new Command ("DECRYPT",Command.HELP,0);
Command verifyPKVR = new Command ("Continue",Command.HELP,0);
Command verifyPKVRv= new Command ("Continue",Command.HELP,0);
Command send = new Command ("SEND", Command.OK,0);
Command send5 = new Command ("SEND", Command.OK,0);
Command post4 = new Command ("POST", Command.OK,0);
Command send6 = new Command ("SEND", Command.OK,0);
Command sign1 = new Command ("SIGN", Command.SCREEN, 0);
Command verify1 = new Command ("VERIFY", Command.OK, 0);
Command verify2 = new Command ("CONFIRM PKA1", Command.OK, 0);
Command verify3 = new Command ("CONFIRM PKA2", Command.OK, 0);
Command confirmUser = new Command ("FindSender", Command.OK, 0);
Command verify4 = new Command ("Verify4", Command.OK, 0);
Command read = new Command ("READ", Command.OK,0);
Command doIt = new Command ("Do it", Command.OK,0);
Command doGauge = new Command ("Gauge", Command.OK,0);
Command SaveUsercmd = new Command ("Add User?", Command.BACK,0);
Command ignore = new Command ("IGNORE", Command.BACK,0);
Command next = new Command ("NEXT", Command.BACK,0);
Command signpkvr4 = new Command ("SignResponse", Command.BACK,0);
Command savesmstest = new Command ("SaveSMS", Command.BACK,0);
private Command exitCommand;
private Command addContactCmd;
private Command backCmd;
private Command backCmd2;
private Command back3;
private Command saveCmd;
private Command viewCmd;
private Command compose;
private Command compose2;
private Command notRevoked;
private Command revoked;
private Command bindidentity;
private Command changeBackCmd;
private Command changeUpdateCmd;
private Command changeDelCmd;
private Command checkPasswordCommand;
private Command sendCommand;
private Command setPasswordCommand;
//private Command enterMessageCommand;
private Command backCommand;
private Command nextCommand;

TextField OCSPfinalRequest = new TextField("OCSP Request","",600,TextField.ANY);
TextField OCSPRequestString = new TextField("OCSP
Request","MFIwUDBOMEwwSjAJBgUrDgMCGgUABBRtl6lMY2%2BiPob4twryIF%2BFfgUdvwQUK8NGq7oOyWUqRtF5R8Ri4uHa%2FLgCEQD7x
yMijIyAItiFkiPe5wZg",300,TextField.ANY);
TextField OCSPserverString = new TextField("OCSP Server URL","",20,TextField.ANY);
TextField ReceivedKeyDetails = new TextField("AESKey","",200,TextField.ANY);
TextField smsText = new TextField("Pub.Key Details", null, 900, TextField.PLAIN);
TextField PKAPhoneCont = new TextField("PKA Phone Number", null, 15, TextField.PHONENUMBER);
TextField ReceivedPKAKeyDetails = new TextField("PKA's Key Details", null, 900, TextField.PLAIN);
```

```
TextField smsNumber = new TextField ("Send to: Number",null, 12, TextField.PHONENUMBER);
TextField smsPubExp = new TextField ("User Contact",null, 12, TextField.PLAIN);
TextField AESKeyField = new TextField("AESKey","",200,TextField.ANY);
TextField AESinitVField = new TextField("AESKey","",200,TextField.ANY);
TextField TwofishCombinedField = new TextField("TwofishKey","",200,TextField.ANY);
TextField PhoneNo = new TextField ("My Phone No","", 20, TextField.PHONENUMBER);
TextField PhoneNo2 = new TextField ("Send to:","", 20, TextField.PHONENUMBER);
TextField PhoneNo2PKVR = new TextField ("User Contact:","", 20, TextField.PHONENUMBER);
TextField PhoneNo2RS = new TextField ("User Contact:","", 20, TextField.PHONENUMBER);
TextField PhoneNo4PKA = new TextField ("PKA's Contact:","", 20, TextField.PHONENUMBER);
TextField PhoneNoRP = new TextField ("Send to RP's Contact:","", 20, TextField.PHONENUMBER);
TextField ContactName = new TextField ("My Name", "", 30, TextField.ANY);
TextField ContactName2PKVR = new TextField ("User Real Name", "", 30, TextField.ANY);
TextField ContactName2RS = new TextField ("User Real Name", "", 30, TextField.ANY);
TextField ContactName2 = new TextField ("Receipient Name", "", 30, TextField.ANY);
TextField ContactUserName = new TextField ("My User Name", "", 30, TextField.ANY);
TextField ContactUserName2 = new TextField ("Receipient UserName", "", 30, TextField.ANY);
TextField ContactUserName2PKVR = new TextField ("User UserName", "", 30, TextField.ANY);
TextField ContactUserName2RS = new TextField ("User UserName", "", 30, TextField.ANY);
TextField publickeyField1 = new TextField ("My Pub.Key Mod","",500, TextField.ANY);
TextField PKVRfinal = new TextField ("Signed PKVR SMS","",500, TextField.ANY);
TextField PKVResponsefinal = new TextField ("Signed PKVR SMS","",900, TextField.ANY);
TextField publickeyField2 = new TextField ("Receipient Pub.Key Mod", "", 500, TextField.ANY);
TextField publickeyField2PKVR = new TextField ("Receipient Pub.Key Mod", "", 500, TextField.ANY);
TextField publicExpField1 = new TextField ("My Pub.Key Exp","",12, TextField.ANY);
TextField publicExpField2 = new TextField ("Receipient Pub.Key Exp","",7, TextField.ANY);
TextField publicExpField21 = new TextField ("Receipient key Status","",12, TextField.ANY);
TextField publicExpField2PKVR = new TextField ("Receipient Pub.Key Exp","",7, TextField.ANY);
TextField contactSMS = new TextField ("Contact","",12, TextField.ANY);
TextField SMSCompo = new TextField ("SMS Message","",900, TextField.ANY);
TextField smsText2 = new TextField ("Type Message","",110, TextField.ANY);
TextField smsText2PKVR = new TextField ("PKV Request SMS","",110, TextField.ANY);
TextField smsText2RS = new TextField ("PKV Response SMS","",800, TextField.ANY);
TextField newMessg = new TextField ("MESSAGE","", 96, TextField.ANY);
TextField PhoneNoPKA = new TextField ("Phone No","", 20, TextField.ANY);
TextField PhoneNoForPKA = new TextField ("Send to PKA:","08077777", 20, TextField.ANY);
TextField ContactNamePKA = new TextField ("Contact Name", "", 80, TextField.ANY);
TextField publickeyField1PKA = new TextField ("Public Key","",300, TextField.ANY);
TextField newMessgPKA = new TextField ("REQUEST","Please confirm the Accuracy of this User's details", 96,
TextField.ANY);
TextField AnotherMessg = new TextField ("MESSAGE","", 160, TextField.ANY);
TextField newMessgtwo = new TextField ("Encrypted Message","", 900, TextField.ANY);
TextField newMessgthree = new TextField ("UNDONE","", 300, TextField.ANY);
TextField newMessgthree2 = new TextField ("Real Message","", 300, TextField.ANY);
TextField newMessgtwoa = new TextField ("Signed Messsage","", 300,TextField.ANY);
TextField newMessgtwob = new TextField ("Encrypted Session Key:","", 300,TextField.ANY);
TextField RSAKeyPubField = new TextField("RSAPubKey","",300,TextField.ANY);
TextField RSAKeyPrivField = new TextField("RSAPrivKey","",300,TextField.ANY);
TextField HelpField = new TextField("About","",200,TextField.ANY);
TextField ReceivedKeyDetails5 = new TextField("User Public Key Details:","",900,TextField.ANY);
TextField ReceivedKeyDetails5a = new TextField("User Public Key Details:","",900,TextField.ANY);
TextField ReceivedKeyDetails6 = new TextField("Secure SMS:","",900,TextField.ANY);
TextField ReceivedKeyDetails6e = new TextField("Revocation Message:","",300,TextField.ANY);
TextField ReceivedKeyDetails6e4 = new TextField("Revocation Message:","",300,TextField.ANY);
TextField ReceivedKeyDetails6f = new TextField("PKV Response Message:","",300,TextField.ANY);
TextField ReceivedKeyDetails7 = new TextField("Session Key:","",900,TextField.ANY);
TextField ReceivedKeyDetails7e = new TextField("Session Key:","",900,TextField.ANY);
TextField CombinedMessagefield = new TextField ("Secure SMS","", 900, TextField.ANY);
TextField ReceivedKeyDetails4 = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails4e = new TextField("Revocation Signature: ", "", 300, TextField.ANY);
TextField ReceivedKeyDetails4f = new TextField("PKV Response Signature: ", "", 600, TextField.ANY);
```

```
TextField ReceivedKeyDetails49 = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails69 = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails79 = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails7a9 = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails4a = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails4ae = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails4a9 = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails7a = new TextField("Message Authenticator:", "", 900, TextField.ANY);
TextField ReceivedKeyDetails7ae = new TextField("Message Authenticator:", "", 900, TextField.ANY);
private TextField searchTxtField;
private TextField searchTxtField2;
private TextField publickeyStatus;
private TextField changepublickeyField;
private TextField publickeyField;
private TextField publickeyFieldPubExp;
private TextField nameTxtField;
private TextField usernameTxtField;
private TextField numberTxtField;
private TextField changeNameTxtField;
private TextField changeNumberTxtField;
private TextField changeUserNameTxtField;
private TextField changepublicExpField;
private TextField changepublicKeySta;
private TextField enterNumberField;
private TextField setPasswordTextField1;
private TextField setPasswordTextField2;
private TextField checkPasswordTextField;

private final Image prosmsLogo1;
private final Image accessLogo;

private RecordStore RSAKeyPairStore = null;
private RecordStore PasswordStore;
private RecordStore PhoneBookStore;
private RecordStore ReceivedSMSStore;

private RSAPrivateCrtKeyParameters RSAprivKey;
private RSAKeyParameters RSApubKey;
private RSAPrivateCrtKeyParameters newRSAprivKey;
private RSAKeyParameters newRSApubKey;
private BigInteger           p;
private DHParameters           dhParams;
private BigInteger              privateValueUser1;
private BigInteger              privateValueUser2;
private BigInteger              publicValueUser2;
private BigInteger              publicValueUser1;
private BigInteger              publicValuePKA1;
private BigInteger              publicValuePKA2;
private BigInteger              gfhfh1;
private BigInteger              gfhfhTwo;
private SecureRandom            random;
private boolean done;
private boolean setPassword = false;
private byte [] Twofishkey;
private byte [] TwofishinitV;
private byte [] Twofishkey4;
private byte [] TwofishinitV4;
private byte [] passwordtriesbytes;
private byte [] passwordbytes;
private byte [] bnmjhg;
private int port;
```

```
private int passwordtries;
private String DHValues;
private String nameStr;
private String numberStr;
private String publickeyStr;
private String userNameStr;
private String publicExpStr;
private String nameStrg;
private String numberStrg;
private String publickeyStrg;
private String userNameStrg;
private String publicExpStrg;
private String password;
private String passwordtriesstring;
private String checkPassword;
private String keyPassword;
private String receivedfrom2;
private String receivedfrom1;
private String filterStr = "";
private String filterStr2 = "";
private String nmbh;
private String errorMsg = null;
private ChoiceGroup searchChoice;
private ChoiceGroup searchChoice2;
private Display display;
private List menuList;
private Vector vectorArray;
private Alert alert;
private Alert alertAbout;
private Alert addAlert;
private Alert warningAlert;
private Alert alError;
private StringItem siBalance;
private MessageConnection conn;
private Reader Msgreader;
private Integer passwordtriesInteger;
private Ticker ticker;
static final String REC_STORE = "db_1";

public prosms_1() {
try
{
port = Integer.parseInt(getAppProperty("port"));
}
catch (Exception e)
{
port = 5000;
}
prosmsLogo1 = makeImage("/ProSMS_logo1.png");
accessLogo = makeImage("/Access_logo.png");
SavedSMSForm.append(SMSCompo);
SavedSMSForm.addCommand (back);
SavedSMSForm.setCommandListener(this);
f1.addCommand(back2);
f1.addCommand(genaes);
f1.setCommandListener(this);
f2.addCommand(back2);
f2.addCommand(genrsa);
f2.setCommandListener(this);
keyXchangeForm3.addCommand(Cont1);
keyXchangeForm3.setCommandListener(this);
```

```
f3.append(HelpField);
f3.addCommand(back);
f3.setCommandListener(this);
f4.append(AESKeyField);
f4.append(AESinitVField);
f4.append (TwofishCombinedField);
f4.addCommand(back);
f4.setCommandListener(this);
siBalance = new StringItem("", "");
f5.append(RSAKeyPubField);
f5.append(RSAKeyPrivField);
f5.addCommand(back);
f5.addCommand(read);
f5.setCommandListener(this);
mainForm5.append(CombinedMessagefield);
mainForm5.addCommand(send2);
mainForm5.addCommand(back);
mainForm5.setCommandListener(this);
RUNTIME_MESSAGING.addCommand(back);
RUNTIME_MESSAGING.setCommandListener(this);
reportForm2.append(" Please wait, Key generation may take some time!");
reportForm2.setCommandListener(this);
sendPKVRForm.append(ContactName2PKVR);
sendPKVRForm.append(ContactUserName2PKVR);
sendPKVRForm.append(PhoneNo2PKVR);
sendPKVRForm.append(PhoneNo4PKA);
sendPKVRForm.append(smsText2PKVR);
sendPKVRForm.addCommand(signPKVRSMS);
sendPKVRForm.addCommand(back);
sendPKVRForm.setCommandListener(this);
RevocationStatusForm.append(ContactName2RS);
RevocationStatusForm.append(ContactUserName2RS);
RevocationStatusForm.append(PhoneNo2RS);
RevocationStatusForm.append(PhoneNoRP);
RevocationStatusForm.append(publickeyField2PKVR);
RevocationStatusForm.append(publicExpField2PKVR);
RevocationStatusForm.append(smsText2RS);
RevocationStatusForm.addCommand(signRSRSMS);
RevocationStatusForm.addCommand(back);
RevocationStatusForm.setCommandListener(this);
mainForm2.append(ContactName2);
mainForm2.append(ContactUserName2);
mainForm2.append(PhoneNo2);
mainForm2.append(publickeyField2);
mainForm2.append(publicExpField2);
mainForm2.append(smsText2);
mainForm2.append(publicExpField21);
mainForm2.addCommand(encryptSMS);
mainForm2.addCommand(encryptSMS1);
mainForm2.addCommand(back);
mainForm2.setCommandListener(this);
mainForm3.addCommand(back);
mainForm3.addCommand(SaveUsercmd);
mainForm3.addCommand(ignore);
mainForm3.setCommandListener(this);
mainForm3a.addCommand(back);
mainForm3a.addCommand(SaveUsercmd);
mainForm3a.addCommand(ignore);
mainForm3a.setCommandListener(this);
mainForm7.setCommandListener(this);
mainForm.append(ContactName);
```

```
mainForm.append(ContactUserName);
mainForm.append(PhoneNo);
mainForm.append(smsNumber);
mainForm.append(publickeyField1);
mainForm.append(publicExpField1);
mainForm.append(smsText);
mainForm.addCommand(send);
mainForm.addCommand(signpkvr4);
mainForm.addCommand(back);
mainForm.setCommandListener(this);
keyStatusForm.setCommandListener(this);
mainForm4.addCommand(confirmUser);
mainForm4.addCommand(discard);
mainForm4.setCommandListener(this);
mainForm4e.addCommand(confirmUser);
mainForm4e.addCommand(verify4);
mainForm4e.setCommandListener(this);
mainForm4e4.addCommand(back);
mainForm4e4.addCommand(back4);
mainForm4e4.setCommandListener(this);
OCSP_MEASURE.addCommand (back);
OCSP_MEASURE.setCommandListener(this);
testOCSPform.append(OCSPserverString);
testOCSPform.append(OCSPRequestString);
testOCSPform.append(OCSPfinalRequest);
testOCSPform.addCommand(cmPOST);
testOCSPform.addCommand(cmGET);
testOCSPform.setCommandListener(this);
mainForm4f.addCommand(confirmUser);
mainForm4f.setCommandListener(this);
mainForm9.addCommand(confirmUser);
mainForm9.addCommand(discard);
mainForm9.setCommandListener(this);
signedRequestForm.append (PKVRfinal) ;
signedRequestForm.addCommand(send5);
signedRequestForm.addCommand(post4);
signedRequestForm.addCommand(back);
signedRequestForm.setCommandListener(this);
signedRequestForm.setCommandListener(this);
signedResponseForm.append (PKVResponsefinal) ;
signedResponseForm.addCommand(send6);
signedResponseForm.addCommand(back);
signedResponseForm.setCommandListener(this);
f7.append(newMessgtwo);
f7.append(newMessgtwoa);
f7.append(newMessgtwob);
f7.addCommand(next);
f7.addCommand(Cont5);
f7.addCommand(back);
f7.setCommandListener(this);
f8.append(newMessgthree);
f8.addCommand(back);
f8.setCommandListener(this);
f9.append(ContactNamePKA);
f9.append(PhoneNoPKA);
f9.append(publickeyField1PKA);
f9.append(PhoneNoForPKA);
f9.append(newMessgPKA);
f9.addCommand(back);
f9.addCommand(encrypt);
f9.addCommand(sign1);
```

```
f9.addCommand(send);
f9.setCommandListener(this);
f15.addCommand(back);
f15.append(newMessgthree2);
f15.addCommand (savesmstest);
f15.setCommandListener(this);
RUNTIME_MESSAGING.setCommandListener(this);
f11.addCommand(back);
f11.setCommandListener(this);
ticker = new Ticker("");
done = false;
try {
PushRegistry.registerConnection("sms://:5000","prosms_1", "*");
} catch (Exception cnfe) {}
enterNumberField = new TextField("Phone number:","",12,TextField.PHONENUMBER);
exitCommand = new Command("Exit", Command.EXIT, 0);
sendCommand = new Command("Send", Command.SCREEN, 0);
String[] menu = {"Create/Reset Keys","Register/Exchange Public Key with PKA",
"Exchange Public Key with Users","Confirm Identity","Send SMS", "Read SMS","About",
"Reset Password", "OCSP Test"};
menuList = new List("MENU", List.IMPLICIT, menu, null);
menuList.addCommand(exitCommand);
menuList.setCommandListener(this);
backCommand = new Command("Back", Command.SCREEN, 0);
nextCommand = new Command("Next", Command.SCREEN, 0);
setPasswordForm = new Form("Register Password");
setPasswordCommand = new Command("Set password", Command.SCREEN, 0);
setPasswordTextField1 = new TextField("Password:","",10,TextField.PASSWORD);
setPasswordTextField2 = new TextField("Confirm:","",10,TextField.PASSWORD);
setPasswordForm.append(setPasswordTextField1);
setPasswordForm.append(setPasswordTextField2);
setPasswordForm.append(prosmsLogo1);
setPasswordForm.addCommand(exitCommand);
setPasswordForm.addCommand(setPasswordCommand);
setPasswordForm.setCommandListener(this);
checkPasswordForm = new Form("ACCESS");
checkPasswordCommand = new Command("OK", Command.SCREEN, 0);
checkPasswordForm.append(accessLogo);
checkPasswordTextField = new TextField("Password:","",10,TextField.PASSWORD);
checkPasswordForm.append(checkPasswordTextField);
checkPasswordForm.addCommand(exitCommand);
checkPasswordForm.addCommand(checkPasswordCommand);
checkPasswordForm.setCommandListener(this);
display = Display.getDisplay(this);
searchForm = new Form("Search/Add Contact");
searchTxtField = new TextField("Search Here", "", 30, TextField.ANY);
searchForm.append(searchTxtField);
searchForm.addCommand(back);
searchChoice = new ChoiceGroup("", Choice.EXCLUSIVE);
searchForm.append(searchChoice);
searchForm.addCommand(exitCommand);
addContactCmd = new Command("Add Contact", Command.OK, 4);
searchForm.addCommand(addContactCmd);
viewCmd = new Command("Edit Contact", Command.OK, 4);
searchChoice.addCommand(viewCmd);
compose = new Command("Compose SMS", Command.OK, 4);
compose2 = new Command("Compose SMS2", Command.OK, 4);
backCmd2 = new Command("TestKey Agreement", Command.OK, 4);
notRevoked = new Command("Send Valid Response", Command.OK, 4);
revoked = new Command("Send Revoked Response", Command.OK, 4);
bindidentity = new Command("Send PKV Request", Command.OK, 5);
```

```
searchChoice.addCommand(compose);
searchChoice.addCommand(compose2);
searchChoice.addCommand(Cont7);
searchChoice.addCommand(backCmd2);
searchChoice.addCommand(bindidentity);
searchChoice.addCommand(notRevoked);
searchChoice.addCommand(revoked);
searchChoice.setItemCommandListener(this);
searchForm.setItemStateListener(this);
searchForm.setCommandListener(this);
searchForm2 = new Form("Search Sender's Key");
searchTxtField2 = new TextField("Search Here", "", 30, TextField.ANY);
searchForm2.append(searchTxtField2);
searchChoice2 = new ChoiceGroup("", Choice.EXCLUSIVE);
searchForm2.append(searchChoice2);
back3 = new Command("BACK", Command.BACK, 4);
searchForm2.addCommand(back3);
searchChoice2.addCommand(verify1);
searchChoice2.addCommand(verify2);
searchChoice2.addCommand(verify3);
searchChoice2.setItemCommandListener(this);
searchForm2.setItemStateListener(this);
searchForm2.setCommandListener(this);
addContactForm = new Form("Add Contacts");
nameTxtField = new TextField("Name", "", 30, TextField.ANY);
addContactForm.append(nameTxtField);
usernameTxtField = new TextField("User Name", "", 30, TextField.ANY);
addContactForm.append(usernameTxtField);
numberTxtField = new TextField("Number", "", 20, TextField.PHONENUMBER);
addContactForm.append(numberTxtField);
publickeyField = new TextField ("Public Key", "", 500, TextField.ANY);
addContactForm.append(publickeyField);
publickeyFieldPubExp = new TextField ("Public Key", "", 7, TextField.ANY);
addContactForm.append(publickeyFieldPubExp);
publickeyStatus = new TextField ("User/Key Status", "Unverified", 12, TextField.ANY);
addContactForm.append(publickeyStatus);
backCmd = new Command("Back", Command.BACK, 2);
addContactForm.addCommand(backCmd);
saveCmd = new Command("Save", Command.OK, 4);
addContactForm.addCommand(saveCmd);
addContactForm.setCommandListener(this);
changeContactForm = new Form("Change Contact");
changeNameTxtField = new TextField("EDIT NAME", "", 30, TextField.ANY);
changeContactForm.append(changeNameTxtField);
changeUserNameTxtField = new TextField("EDIT USER NAME", "", 30, TextField.ANY);
changeContactForm.append(changeUserNameTxtField);
changeNumberTxtField = new TextField("EDIT NUMBER", "", 20, TextField.PHONENUMBER);
changeContactForm.append(changeNumberTxtField);
changepublickeyField = new TextField ("Public Key","",500, TextField.ANY);
changeContactForm.append(changepublickeyField);
changepublicExpField = new TextField ("Public Key","",7, TextField.ANY);
changeContactForm.append(changepublicExpField);
changepublicKeySta = new TextField ("Public Key Status","",12, TextField.ANY);
changeContactForm.append(changepublicKeySta);
changeBackCmd = new Command("Back", Command.BACK, 2);
changeContactForm.addCommand(changeBackCmd);
changeUpdateCmd = new Command("Update", Command.OK, 4);
changeContactForm.addCommand(changeUpdateCmd);
changeDelCmd = new Command("Delete", Command.OK, 4);
changeContactForm.addCommand(changeDelCmd);
changeContactForm.setCommandListener(this);
```

```
try {
PhoneBookStore = RecordStore.openRecordStore("Contacts", true);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
}

public void startApp() {
String connection = "sms://:" + 5000;
try {
conn = (MessageConnection) Connector.open(connection);
conn.setMessageListener(this);
done = false;
Msgreader = new Reader(this);
new Thread(Msgreader).start();
} catch(java.io.IOException ie) {}
String error = "";
try{
PasswordStore = RecordStore.openRecordStore("FirstTime", false);
passwordbytes = PasswordStore.getRecord(1);
passwordtriesbytes = PasswordStore.getRecord(2);
try {
password = new String(passwordbytes,"UTF8");
passwordbytes = PasswordStore.getRecord(3);
keyPassword = new String(passwordbytes,"UTF8");
passwordtriesstring = new String(passwordtriesbytes,"UTF8");
}catch(UnsupportedEncodingException uee) {}
passwordtriesInteger = new Integer(0);
passwordtries = passwordtriesInteger.parseInt(passwordtriesstring);
PasswordStore.closeRecordStore();
} catch(RecordStoreNotFoundException eNotFound){
try {
PasswordStore = RecordStore.openRecordStore("FirstTime",true);
setPassword = true;
PasswordStore.closeRecordStore();
} catch(RecordStoreException eRSE){
}
} catch(RecordStoreException eRSE2){
enterNumberField.setString(error+" FEL ");
}
if(setPassword == true){
alertAbout = new Alert("General Info", "This application is meant for secure SMS "
+ "communication. To prevent unauthorised access to he application, you are required"
+ "to use a STRONG" + " but easily remembered password. Always change this password "
+ "anytime you feel the password has been compromised.", null, AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,setPasswordForm);
} else {
display.setCurrent(checkPasswordForm);
}
}
void showMessage(String message)
{
}
static Image makeImage(String filename)
{
Image image = null;
try
{
image = Image.createImage(filename);
```

```
}
catch (Exception e)
{
}
return image;
}
private void handleSendUserKeyDetails() {
// Sends User public key details to other users
Thread smsThread = new Thread() {
private MessageConnection connection;
private TextMessage message;
public void run() {
try {
String number;
number =  "sms://" + smsNumber.getString() + ":" + port;
String text = smsText.getString();
connection = (MessageConnection)Connector.open(number);
message = (TextMessage)connection.newMessage(MessageConnection.TEXT_MESSAGE);
message.setPayloadText(text);
connection.send(message);
connection.close ();
}
catch (java.io.IOException ie) {
}
}
};
smsThread.start();
}
private void handleSendPKAKeyExchange() {
// Sends User public key details to other users
Thread smsThread = new Thread() {
private MessageConnection connection;
private TextMessage message;
public void run() {
try {
String number;
number =  "sms://" + PKAPhoneCont.getString() + ":" + port;
String text = DHValues;
connection = (MessageConnection)Connector.open(number);
message = (TextMessage)connection.newMessage(MessageConnection.TEXT_MESSAGE);
message.setPayloadText(text);
connection.send(message);
connection.close ();
}
catch (java.io.IOException ie) {
}
}
};
smsThread.start();
}
private void handleSendPKAEncryptedKey() {
// Sends User encrypted public key details to the PKA
Thread smsThread = new Thread() {
private MessageConnection connection;
private TextMessage message;
public void run() {
try {
String number;
number =  "sms://" + PKAPhoneCont.getString() + ":" + port;
String text = "6666" + newMessgtwo.getString();
connection = (MessageConnection)Connector.open(number);
```

```
message = (TextMessage)connection.newMessage(MessageConnection.TEXT_MESSAGE);
message.setPayloadText(text);
connection.send(message);
connection.send (null);
connection.close ();
}
catch (java.io.IOException ie) {
}
}
};
smsThread.start();
}
private void handleSendPKAKeyExchange2() {
// Sends User DH key parameters to PKA
Thread smsThread = new Thread() {
private MessageConnection connection;
private TextMessage message;
public void run() {
try {
String number;
number =  "sms://" + PKAPhoneCont.getString() + ":" + port;
String text = "5555" + publicValueUser2.toString();
connection = (MessageConnection)Connector.open(number);
message = (TextMessage)connection.newMessage(MessageConnection.TEXT_MESSAGE);
message.setPayloadText(text);
connection.send(message);
connection.close ();
}
catch (java.io.IOException ie) {
}
}
};
smsThread.start();
}
private void handleSendCommand3() {
// sends PKV Request to the PKA
Thread smsThread = new Thread() {
private MessageConnection connection;
private BinaryMessage message;
public void run() {
try {
String number;
number =  "sms://" + PhoneNo4PKA.getString() + ":" + port;
byte [] text = PKVRfinal.getString().getBytes();
connection = (MessageConnection)Connector.open(number);
message = (BinaryMessage)connection.newMessage(MessageConnection.BINARY_MESSAGE);
message.setPayloadData(text);
connection.send(message);
connection.close ();
}
catch (IOException ex) {
}
}
};
smsThread.start();
}
private void handleSendCommand4() {
//Sends PVResponse/Revocation info to RPs
Thread smsThread = new Thread() {
private MessageConnection connection;
private BinaryMessage messages;
```

```
public void run() {
try {
String number;
number =  "sms://" + PhoneNoRP.getString() + ":" + port;
byte [] text = PKVResponsefinal.getString().getBytes();
connection = (MessageConnection)Connector.open(number);
messages = (BinaryMessage)connection.newMessage(MessageConnection.BINARY_MESSAGE);
messages.setPayloadData(text);
connection.send(messages);
connection.send (null);
connection.close ();
}
catch (IOException ex) {
}
}
};
smsThread.start();
}
private void handleSendCommand2() {
// sends secure messages to others
Thread smsThread = new Thread() {
private MessageConnection connection2;
private BinaryMessage message;
public void run() {
try {
String number;
number =  "sms://" + PhoneNo2.getString() + ":" + port;
String text1 = CombinedMessagefield.getString();
byte[] text2 = text1.getBytes();
connection2 = (MessageConnection)Connector.open(number);
message = (BinaryMessage)connection2.newMessage(MessageConnection.BINARY_MESSAGE);
message.setPayloadData(text2);
connection2.send(message);
connection2.close ();
}
catch (IOException ex) {
}
}
};
smsThread.start();
}
public void notifyIncomingMessage(final MessageConnection conn) {
Thread smsThread = new Thread() {
public void run() {
try {
Message message = conn.receive();
if (message != null) {
processMessage(message);
}
} catch (IOException ex) {
}
}
};
smsThread.start();
}
private void processMessage(Message message) {
if (message instanceof TextMessage) {
processTextMessage((TextMessage)message);
} else if (message instanceof BinaryMessage) {
processBinaryMessage((BinaryMessage)message);
}
```

```
}
private void processTextMessage(TextMessage message) {
//Recceives User Public Key Details via traditional SMS
String msgReceived = null;
msgReceived = ((TextMessage)message).getPayloadText();
if (msgReceived.startsWith("0000", 0)){
StringItem textItem = new StringItem("Text", msgReceived);
String receivedfroma = message.getAddress();
receivedfrom1 = receivedfroma.substring(6,receivedfroma.length()-2);
mainForm3.deleteAll();
mainForm3.append("A User sent you his/her Public Key details. You want to Add User to your PhoneBook?");
mainForm3.append("Received Public Key Details from: " + receivedfrom1 );
mainForm3.append(ReceivedKeyDetails5);
ReceivedKeyDetails5.setString(msgReceived);
display.setCurrent(mainForm3);
//Receives DH key Agreement parameters via traditional SMS
} else if (msgReceived.startsWith("1111", 0)){
StringItem textItem = new StringItem("Text", msgReceived);
String receivedfroma = message.getAddress();
receivedfrom1 = receivedfroma.substring(6,receivedfroma.length()-2);
keyXchangeForm3.deleteAll();
PKAPhoneCont.setString(receivedfrom1);
keyXchangeForm3.removeCommand(Cont1);
keyXchangeForm3.addCommand(Cont2);
keyXchangeForm3.append("Your PKA sent you some key exchange data, just press continue to finiish the
process");
keyXchangeForm3.append("PKA's contact: " + receivedfrom1 );
keyXchangeForm3.append(ReceivedPKAKeyDetails);
ReceivedPKAKeyDetails.setString(msgReceived);
display.setCurrent(keyXchangeForm3);
//Receives DH public key to compute symmetric key
} else if (msgReceived.startsWith("5555", 0)){
StringItem textItem = new StringItem("Text", msgReceived);
String receivedfroma = message.getAddress();
receivedfrom1 = receivedfroma.substring(6,receivedfroma.length()-2);
keyXchangeForm3.deleteAll();
PKAPhoneCont.setString(receivedfrom1);
keyXchangeForm3.removeCommand(Cont1);
keyXchangeForm3.removeCommand(Cont2);
keyXchangeForm3.addCommand(Cont3);
keyXchangeForm3.addCommand(back);
keyXchangeForm3.append("Just press continue...");
keyXchangeForm3.append("PKA's contact: " + receivedfrom1 );
keyXchangeForm3.append(ReceivedPKAKeyDetails);
ReceivedPKAKeyDetails.setString(msgReceived);
display.setCurrent(keyXchangeForm3);
}
//Receives public key encrypted using DH-computed symmetric key
else if (msgReceived.startsWith("6666", 0)){
StringItem textItem = new StringItem("Text", msgReceived);
String receivedfroma = message.getAddress();
receivedfrom1 = receivedfroma.substring(6,receivedfroma.length()-2);
PKAPhoneCont.setString(receivedfrom1);
keyXchangeForm3.deleteAll();
keyXchangeForm3.removeCommand(Cont1);
keyXchangeForm3.removeCommand(Cont2);
keyXchangeForm3.removeCommand(Cont3);
keyXchangeForm3.removeCommand(Cont4);
keyXchangeForm3.addCommand(Cont6);
keyXchangeForm3.append("You have received an encrypted key. Click OK to complete the key exchnage
process...");
```

```
keyXchangeForm3.append("PKA's contact: " + receivedfrom1 );
keyXchangeForm3.append(ReceivedPKAKeyDetails);
ReceivedPKAKeyDetails.setString(msgReceived);
display.setCurrent(keyXchangeForm3);
}
}
private void processBinaryMessage(BinaryMessage message) {
//Receives encrypted SMS between users
byte [] dataString = null;
dataString = ((BinaryMessage)message).getPayloadData();
String msgReceived = new String(dataString);
if (msgReceived.startsWith("2222", 0)){
String receivedfrom = message.getAddress();
receivedfrom2 = receivedfrom.substring(6,receivedfrom.length()-2);
mainForm4.deleteAll();
mainForm4.append("Received Secure Message from: " + receivedfrom2 );
String nextRecdStr = new String(msgReceived);
String takeSMS = nextRecdStr.substring(4, nextRecdStr.indexOf("****"));
String takeSIGN = nextRecdStr.substring(nextRecdStr.indexOf("****") +4, nextRecdStr.indexOf("####"));
String takeSessionKey =    nextRecdStr.substring(nextRecdStr.indexOf("####") +4, nextRecdStr.length());
mainForm4.append(ReceivedKeyDetails6);
ReceivedKeyDetails6.setString(takeSMS);
mainForm4.append(ReceivedKeyDetails4);
ReceivedKeyDetails4.setString(takeSIGN);
mainForm4.append(ReceivedKeyDetails7);
ReceivedKeyDetails7.setString(takeSessionKey);
display.setCurrent(mainForm4);
}
// Receives Revocation Response from PKA
else if (msgReceived.startsWith("3333", 0)){
String receivedfrom = message.getAddress();
receivedfrom2 = receivedfrom.substring(6,receivedfrom.length()-2);
mainForm4e.deleteAll();
mainForm4e.append("Got REVOCATION Response from PKA: " + receivedfrom2 );
String nextRecdStr = new String(msgReceived);
String takePKVR = nextRecdStr.substring(4, nextRecdStr.indexOf("$$$$"));
String takePKVRSIGN = nextRecdStr.substring(nextRecdStr.indexOf("$$$$") +4, nextRecdStr.length());
mainForm4e.append(ReceivedKeyDetails4e);
ReceivedKeyDetails4e.setString(takePKVRSIGN);
mainForm4e.append(ReceivedKeyDetails6e);
ReceivedKeyDetails6e.setString(takePKVR);
display.setCurrent(mainForm4e);
}
// Receives PKV Response from PKA
else if (msgReceived.startsWith("4444", 0)){
String receivedfrom = message.getAddress();
receivedfrom2 = receivedfrom.substring(6,receivedfrom.length()-2);
mainForm4f.deleteAll();
mainForm4f.append("Received PKV Response from PKA: " + receivedfrom2 );
String nextRecdStr = new String(msgReceived);
String takePKVResponse = nextRecdStr.substring(4, nextRecdStr.indexOf("$$$$"));
String takePKVResponseSIGN = nextRecdStr.substring(nextRecdStr.indexOf("$$$$") +4, nextRecdStr.length());
mainForm4f.append(ReceivedKeyDetails4f);
ReceivedKeyDetails4f.setString(takePKVResponseSIGN);
mainForm4f.append(ReceivedKeyDetails6f);
ReceivedKeyDetails6f.setString(takePKVResponse);
display.setCurrent(mainForm4f);
}
}
public void pauseApp() {
done = true;
```

```
try {
conn.close();
}catch(java.io.IOException ie) {}
}
public void destroyApp(boolean unconditional) {
done = true;
try {
conn.setMessageListener(null);
conn.close();
} catch (java.io.IOException e) {}
}
public void setTicker(String setstring) {
ticker.setString(setstring);
}
public void updateTicker() {
if(Msgreader.unreadMessages()>0) {
ticker.setString("You have message to read.");
} else {
ticker.setString("No message to read.");
}
}
public void commandAction(Command c, Displayable s) {
updateTicker();
s.setTicker(ticker);
if(c == List.SELECT_COMMAND) {
int menuindex = menuList.getSelectedIndex();
if(menuindex == 4) {
updateTicker();
try {
PhoneBookStore = RecordStore.openRecordStore("Contacts", true);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
display.setCurrent(searchForm);
} else if(menuindex == 5) {
updateTicker();
if(Msgreader.unreadMessages()>0) {
} else {
    openRecStore3();
readSMSMessage();
closeRecStore3();
alert = new Alert("INBOX", "Click to read SMS.", null, AlertType.INFO);
//alert.setTimeout(2000);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,SavedSMSForm);
}
} else if(menuindex == 0) {
updateTicker();
alertAbout = new Alert("Crypto Keys", " Use this feature to create/change the cryptographic"
+ "keys used by this application. It is adviceable to change this key ONLY if you fear your "
+ "application has been compromised. Note however, that when you change your keys you will not"
+ "be able to Decrypt Messages encrypted with your previous keys. You will have to exchage your NEW key "
+ "with your communicating partners before you can decrypt messages they encrypt with your "
+ "current key.", null, AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,f1);
}
else if(menuindex == 1) {
updateTicker();
alertAbout = new Alert("REGISTER PUBLIC KEY", "Use this page to Register your Public Key "
```

```
+ "with the Public Key Authenticator (PKA). The PKA is your Network Service Provider."
+ " This step is neccessary so other user can authenticate you at the PKA. Make sure you have"
+ "generated the keys first using the Create/Reset keys link on the main menu before"
+ " attempting to use this feature", null, AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,keyXchangeForm3);
}
else if(menuindex == 2) {
updateTicker();
openRecStore();
ExchangeRSApub();
closeRecStore();
alertAbout = new Alert("EXCHANGE PUBLIC KEY", "Use this page to exchange your Public Key "
+ "with other users. This step is neccessary so that they can encrypt messages sent to you. "
+ "Make sure you have generated the keys first using the Create/Reset keys link on the main menu before"
+ " attempting to use this feature", null, AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,mainForm);
}
else if(menuindex == 3) {
updateTicker();
alertAbout = new Alert("IDENTITY BINDING", "Use this page to confirm the "
+ "identity of a communicating partener IF you are in doubt. It is strongly adviced "
+ "that you do this for every individual when you are meeting them for the first time."
+ " You might be charged for this service by the PKA", null, AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,searchForm);
}else if(menuindex == 6) {
updateTicker();
alertAbout = new Alert("About ProSMS 1.0", "ProSMS Version 1.0 is.....", null, AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,menuList);
}
else if(menuindex == 8) {
updateTicker();
alertAbout = new Alert("OCSP Test", "This is used for testing Regular OCSP Request and Response", null,
AlertType.INFO);
alertAbout.setTimeout(Alert.FOREVER);
display.setCurrent(alertAbout,testOCSPform);
}
else if(menuindex == 7) {
updateTicker();
if(Msgreader.unreadMessages()>0) {
}
alert = new Alert("Reset Password",  "Use this feature to change your password access to the "
+ "application anytime you feel the secrecy of the password has ben compromised. Your previously "
+ "saved data (like saved SMS etc.) would not be affected in anyway.", null, AlertType.INFO);
alert.setTimeout(3000);
try{
PasswordStore.deleteRecordStore("FirstTime");
} catch(RecordStoreException eRSE){}
try{
PasswordStore.deleteRecordStore("Keys");
} catch(RecordStoreException eRSE){}
display.setCurrent(alert);
destroyApp(true);
notifyDestroyed();
}
}
if (c == genaes )
{
```

```
display.setCurrent(reportForm2);
try {
generateRSAKeyPair ();
} catch (Exception ex) {
ex.printStackTrace();
}
}
if (c == back2)
{
display.setCurrent(menuList);
}
if (c == back4)
{
display.setCurrent(mainForm4e);
}
if (c == ok)
{
display.setCurrent(menuList);
}
if (c == yes2)
{
display.setCurrent(sendPKVRForm);
}
if (c == back)
{
display.setCurrent(menuList);
}
if (c == discard)
{
display.setCurrent(menuList);
}
if (c == confirmUser)
{
display.setCurrent(searchForm2);
searchTxtField2.setString(receivedfrom2);
}
if (c == next)
{
String SMSMessage = newMessgtwo.getString();
String SMSSign =    newMessgtwoa.getString();
String Authenticator =   newMessgtwob.getString();
String ComposedSMS = "2222" + SMSMessage + "****" + SMSSign + "####" + Authenticator;
CombinedMessagefield.setString(ComposedSMS);
display.setCurrent(mainForm5);
}
if (c == exitCommand) {
updateTicker();
if(Msgreader.unreadMessages()>0) {
} else {
destroyApp(true);
notifyDestroyed();
}
}
if (c == Cont5)
{
try  {
TwofishDecrypt2 () ;
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(f15);
```

```
}
if (c == encryptSMS1)
{
try {
TwofishEncrypt2 () ;
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(f7);
}
if (c == encryptSMS)
{
try {
generateTwofishKey ();
} catch (Exception ex) {
ex.printStackTrace();
}
openRecStore();
readRSApriv ();
closeRecStore();
try {
RSASign ();
} catch (Exception ex) {
ex.printStackTrace();
}
String UserMod = publickeyField2.getString();
BigInteger RSAmod1 = new BigInteger(UserMod);
String UserPubExp = publicExpField2.getString();
BigInteger RSAPubExp1 = new BigInteger(UserPubExp);
newRSApubKey = new RSAKeyParameters(false, RSAmod1, RSAPubExp1);
try {
RSAEncryptTwofish ();
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(f7);
}
if (c == signPKVRSMS)
{
openRecStore();
readRSApriv ();
closeRecStore();
try {
RSASign2 ();
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(signedRequestForm);
}
if (c == signRSRSMS)
{
openRecStore();
readRSApriv ();
closeRecStore();
try {
RSASign3 ();
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(signedResponseForm);
}
```

```
if (c == encrypt)
{
try {
RSAEncrypt ();
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(f7);
}
if (c == decrypt)
{
openRecStore();
readRSApriv ();
closeRecStore();
try {
RSADecrypt ()   ;
} catch (Exception ex) {
ex.printStackTrace();
}
try {
TwofishDecrypt () ;
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(f15);
}
if (c == rsaSave)
{
display.setCurrent(f7);
try {
openRecStore();
SerializeRSAKeyPair();
closeRecStore();
} catch (Exception ex) {
ex.printStackTrace();
}
}
if (c == rsaprivRetrieve)
{
display.setCurrent(f8);
try {
openRecStore();
readRSApub ();
closeRecStore();
} catch (Exception ex) {
ex.printStackTrace();
}
}
if (c == sign1)
{
openRecStore();
readRSApriv ();
closeRecStore();
try {
RSASign ();
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(f7);
}
else if (c== verify4){
```

```
openRecStore();
readRSApub ();
closeRecStore();
try {
RSAVerify1 ();
} catch (Exception ex) {
ex.printStackTrace();
}
}
else if (c== send){
nameStrg = ContactName.getString();//publickeyFieldPubExp
userNameStrg = ContactUserName.getString();
numberStrg = PhoneNo.getString();
publickeyStrg = publickeyField1.getString();
publicExpStrg = publicExpField1.getString();
String fullUserDetails = "0000" + nameStrg + "*" + userNameStrg + "#" + numberStrg + "&" + publickeyStrg +
"$" + publicExpStrg;
smsText.setString(fullUserDetails);
handleSendUserKeyDetails ();
}
else if (c== Cont1){
try {
generateDHKeyPaira ();
} catch (Exception ex) {
ex.printStackTrace();
}
handleSendPKAKeyExchange ();
display.setCurrent(menuList);
}
else if (c== Cont2){
try {
generateDHKeyPair2a ();
} catch (Exception ex) {
ex.printStackTrace();
}
calculateAgreementa();
handleSendPKAKeyExchange2 ();
keyXchangeForm3.removeCommand(Cont2);
keyXchangeForm3.addCommand(Cont4);
keyXchangeForm3.deleteAll();
keyXchangeForm3.append("Press Continue once more to complete your Registartion");
keyXchangeForm3.deleteAll();
keyXchangeForm3.append(" Symmetric Key at A = " + gfhfh1.toString());
display.setCurrent(keyXchangeForm3);
}
else if (c== Cont6){
try  {
TwofishDecrypt2 () ;
} catch (Exception ex) {
ex.printStackTrace();
String vcbcb = newMessgthree2.getString();
String nextRecdStr = new String(vcbcb);
String takepubkey = nextRecdStr.substring(4, nextRecdStr.indexOf("****"));
String takePubExp = nextRecdStr.substring(nextRecdStr.indexOf("****") +4, nextRecdStr.length());
publickeyField.setString(takepubkey);
publickeyFieldPubExp.setString(takePubExp);
display.setCurrent(addContactForm);
}
}
else if (c== Cont4){
openRecStore();
```

```
ExchangeRSApub();
closeRecStore();
nmbh = publickeyField1.getString() + "****" + publicExpField1.getString ();
System.out.println ("nmbh = " + nmbh);
try {
TwofishEncrypt2 () ;
} catch (Exception ex) {
ex.printStackTrace();
}
keyXchangeForm3.deleteAll();
keyXchangeForm3.removeCommand(Cont4);
keyXchangeForm3.removeCommand(back);
handleSendPKAEncryptedKey ();
keyXchangeForm3.append(" Your public key has been securely sent to your PKA. " + "please click back to go the
Main Menu");
display.setCurrent(keyXchangeForm3);
}
else if (c== Cont3){
calculateAgreement2b ();
openRecStore();
ExchangeRSApub();
closeRecStore();
nmbh = publickeyField1.getString() + "****" + publicExpField1.getString ();
System.out.println ("nmbh = " + nmbh);
try {
TwofishEncrypt2 () ;
} catch (Exception ex) {
ex.printStackTrace();
}
keyXchangeForm3.deleteAll();
handleSendPKAEncryptedKey ();
keyXchangeForm3.deleteAll();
keyXchangeForm3.removeCommand(Cont3);
keyXchangeForm3.addCommand(back);
keyXchangeForm3.append(" Symmetric Key at B = " + gfhfhTwo.toString());
keyXchangeForm3.append(" Your public key has been securely sent. Please click " + " Main to go back to the
Main Menu");
display.setCurrent(keyXchangeForm3);
}
else if (c== signpkvr4){
nameStrg = ContactName.getString();
userNameStrg = ContactUserName.getString();
numberStrg = PhoneNo.getString();
publicExpStrg = publicExpField1.getString();
String fullUserDetails4 = nameStrg + "*" + userNameStrg + "#" + numberStrg + "&" + publicExpStrg;
smsText.setString(fullUserDetails4);
openRecStore();
readRSApriv ();
closeRecStore();
try {
RSASign4 ();
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(signedRequestForm);
}
else if (c== send5){
handleSendCommand3 ();
}
else if (c== post4){
String SignedPKVR = PKVRfinal.getString();
```

```
String takePKVR = SignedPKVR.substring(0, SignedPKVR.indexOf("$$$$"));
String takePKVRSIGN = SignedPKVR.substring(SignedPKVR.indexOf("$$$$") +4, SignedPKVR.length());
mainForm4e.append(ReceivedKeyDetails4e);
ReceivedKeyDetails4e.setString(takePKVRSIGN);
mainForm4e.append(ReceivedKeyDetails6e);
ReceivedKeyDetails6e.setString(takePKVR);
display.setCurrent(mainForm4e);
}
else if (c== send6){
handleSendCommand4 ();
}
else if (c== savesmstest){
try {
openRecStore3();
SaveSMS ();
closeRecStore3();
} catch (Exception ex) {
ex.printStackTrace();
}
display.setCurrent(menuList);
}
else if (c== send2){
handleSendCommand2 ();
}
if (c == SaveUsercmd )
{
String OtherUsersDetails = ReceivedKeyDetails5.getString();
String nextRecdStr = new String(OtherUsersDetails);
String takeNames = nextRecdStr.substring(4, nextRecdStr.indexOf("*"));
String takeUserNames = nextRecdStr.substring(nextRecdStr.indexOf("*") +1, nextRecdStr.indexOf("#"));
String takeNumbers =    nextRecdStr.substring(nextRecdStr.indexOf("#") +1, nextRecdStr.indexOf("&"));
String takePublicKey = nextRecdStr.substring(nextRecdStr.indexOf("&") + 1, nextRecdStr.indexOf("$"));
String takePublicExp = nextRecdStr.substring(nextRecdStr.indexOf("$") + 1, nextRecdStr.length());
nameTxtField.setString(takeNames);
usernameTxtField.setString(takeUserNames);
numberTxtField.setString(takeNumbers);
publickeyField.setString(takePublicKey);
publickeyFieldPubExp.setString(takePublicExp);
display.setCurrent(addContactForm);
}
else if(c== setPasswordCommand){
updateTicker();
String verifyPassword1 = setPasswordTextField1.getString();
String verifyPassword2 = setPasswordTextField2.getString();
if ((verifyPassword1.compareTo(verifyPassword2)) == 0){
try{
PasswordStore = RecordStore.openRecordStore("FirstTime",false);
password = verifyPassword1;
try {
PasswordStore.addRecord(verifyPassword1.getBytes("UTF8"),0,verifyPassword1.getBytes("UTF8").length);
passwordtriesstring = "0";
PasswordStore.addRecord(passwordtriesstring.getBytes("UTF8"),0,passwordtriesstring.getBytes("UTF8").length);
keyPassword ="";
keyPassword = password;
while (keyPassword.length() < 4){
keyPassword = keyPassword + keyPassword;
}
if (keyPassword.length() > 4){
keyPassword = keyPassword.substring(0,5);
}
PasswordStore.addRecord(keyPassword.getBytes("UTF8"),0,keyPassword.getBytes("UTF8").length);
```

```
}catch(UnsupportedEncodingException uee) {}
PasswordStore.closeRecordStore();
} catch(RecordStoreException eRSE){}
alert = new Alert("Password", "Password set and saved ", null, null);
alert.setTimeout(2000);
display.setCurrent(alert,checkPasswordForm);
} else {
alert = new Alert("Password", "Password mismatch", null, null);
alert.setTimeout(2000);
display.setCurrent(alert,setPasswordForm);
}
}
if (c == ignore)
{
display.setCurrent(menuList);
}
if (c == cmGET || c == cmPOST)
{
try
{
if (c == cmGET)
lookupBalance_withGET();
else
lookupBalance_withPOST();
}
catch (Exception e)
{
System.err.println("Msg: " + e.toString());
}
}
if (c == back3)
{
display.setCurrent(mainForm4);
}
else if (c == saveCmd) {
nameStr = nameTxtField.getString();
userNameStr = usernameTxtField.getString();
numberStr = numberTxtField.getString();
publickeyStr = publickeyField.getString();
publicExpStr = publickeyFieldPubExp.getString();
String fullDetails = nameStr + "*" + userNameStr + "#" + numberStr + "&" + publickeyStr + "$" + publicExpStr
+ "!" + publickeyStatus.getString();
addAlert = new Alert("Added to the Contacts");
loadContacts();
display.setCurrent(addAlert, searchForm);
searchTxtField.setString("");
searchTxtField2.setString("");
byte[] byteArray = fullDetails.getBytes();
try {
PhoneBookStore.addRecord(byteArray, 0, fullDetails.length());
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
}
else if(c == changeBackCmd){
loadContacts();
display.setCurrent(searchForm);
}
else if(c == changeUpdateCmd){
int getSelInd = searchChoice.getSelectedIndex();
```

```
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
int recordId = getConInfo.getContactId();
String changeName = changeNameTxtField.getString();
String ChangeUserName = changeUserNameTxtField.getString ();
String changeNumber = changeNumberTxtField.getString();
String changePublicKey = changepublickeyField.getString();
String changePublicExp = changepublicExpField.getString();
String changePublicKeyStatus = changepublicKeySta.getString();
String changeFullDetails = changeName + "*" + ChangeUserName + "#" + changeNumber + "&" + changePublicKey +
"$" + changePublicExp + "!" + changePublicKeyStatus;
byte[] editRec = changeFullDetails.getBytes();
try {
PhoneBookStore.setRecord(recordId, editRec, 0, editRec.length);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
warningAlert = new Alert("Successfully updated", "", null, AlertType.INFO);
display.setCurrent(warningAlert, searchForm);
}
else if (c == addContactCmd) {
nameTxtField.setString("");
numberTxtField.setString("");
display.setCurrent(addContactForm);
}
else if (c == backCmd) {
searchTxtField.setString("");
display.setCurrent(searchForm);
}
else if(c == changeDelCmd){
int getSelInd = searchChoice.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
int recordId = getConInfo.getContactId();
try {
PhoneBookStore.deleteRecord(recordId);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
warningAlert = new Alert("Permanantly Deleted !", "", null, AlertType.CONFIRMATION);
display.setCurrent(warningAlert, searchForm);
}
else if(c==backCommand) {
updateTicker();
display.setCurrent(menuList);
}
else if(c==nextCommand) {
updateTicker();
if(Msgreader.unreadMessages()>0) {
}
else {
alert = new Alert("SMS", "No more SMS", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,menuList);
}
}
else if(c==checkPasswordCommand){
updateTicker();
checkPassword = checkPasswordTextField.getString();
if((checkPassword.compareTo(password))==0) {
try{
```

```
PasswordStore = RecordStore.openRecordStore("FirstTime",false);
passwordtriesstring = "0";
try {
PasswordStore.setRecord(2,passwordtriesstring.getBytes("UTF8"),0,passwordtriesstring.getBytes("UTF8").length)
;
}
catch(UnsupportedEncodingException uee) {}
PasswordStore.closeRecordStore();
}
catch(RecordStoreException eRSE){}
updateTicker();
if(Msgreader.unreadMessages()>0) {
} else {
display.setCurrent(menuList);
}
} else {
updateTicker();
passwordtries = passwordtries+1;
if(passwordtries == 3) {
try{
PasswordStore.deleteRecordStore("FirstTime");
PasswordStore.deleteRecordStore("Keys");
} catch(RecordStoreException eRSE){}
alert = new Alert("Password error", "To many password tries", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);
destroyApp(true);
notifyDestroyed();
} else {
passwordtriesInteger = new Integer(passwordtries);
passwordtriesstring = passwordtriesInteger.toString();
try{
PasswordStore = RecordStore.openRecordStore("FirstTime",true);
try {
PasswordStore.setRecord(2,passwordtriesstring.getBytes("UTF8"),0,passwordtriesstring.getBytes("UTF8").length)
;
}catch(UnsupportedEncodingException uee) {}
PasswordStore.closeRecordStore();
} catch(RecordStoreException eRSE){}
alert = new Alert("Password", "Password mismatch", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,checkPasswordForm);
}
}
}
}
public void SaveSMS ()
{
    String smscomponent = newMessgthree2.getString();

    String [] smsstrings = {smscomponent};
    writesmsstring (smsstrings);
}
public void writesmsstring(String[] smsData)
{
try
{
ByteArrayOutputStream strmBytes =
new ByteArrayOutputStream();
DataOutputStream strmDataType =
new DataOutputStream(strmBytes);
```

```
byte[] record;
for (int i = 0; i < smsData.length; i++)
{
strmDataType.writeUTF(smsData[i]);
strmDataType.flush();
record = strmBytes.toByteArray();
ReceivedSMSStore.addRecord(record, 0, record.length);
strmBytes.reset();
}
strmBytes.close();
strmDataType.close();
}
catch (Exception e)
{
db(e.toString());
}
}
public void readSMSMessage()
{
try
{
byte[] recData = new byte[500];
ByteArrayInputStream strmBytes =
new ByteArrayInputStream(recData);
DataInputStream strmDataType =
new DataInputStream(strmBytes);
ReceivedSMSStore.getRecord(1, recData, 0);
String mod = strmDataType.readUTF();
strmBytes.reset();
SMSCompo.setString(mod);
strmBytes.close();
strmDataType.close();
}
catch (Exception e)
{
db(e.toString());
}
}
public void SerializeRSAKeyPair() // stores generated RSA Key Pairs
{
BigInteger mod = RSAprivKey.getModulus();
String mods = mod.toString();
BigInteger privExp = RSAprivKey.getExponent();
String privExps = privExp.toString();
BigInteger pubExp = RSAprivKey.getPublicExponent();
String pubExps = pubExp.toString();
BigInteger dp = RSAprivKey.getDP();
String dps = dp.toString();
BigInteger dq = RSAprivKey.getDQ();
String dqs = dq.toString();
BigInteger p = RSAprivKey.getP();
String ps = p.toString();
BigInteger q = RSAprivKey.getQ();
String qs = q.toString();
BigInteger qInv = RSAprivKey.getQInv();
String qInvs = qInv.toString();
String [] strings = {mods, privExps,pubExps,dps,dqs, ps,qs,qInvs};
writeStream(strings);
}
public void writeStream(String[] sData)
{
```

```
try
{
ByteArrayOutputStream strmBytes =
new ByteArrayOutputStream();
DataOutputStream strmDataType =
new DataOutputStream(strmBytes);
byte[] record;
for (int i = 0; i < sData.length; i++)
{
strmDataType.writeUTF(sData[i]);
strmDataType.flush();
record = strmBytes.toByteArray();
RSAKeyPairStore.addRecord(record, 0, record.length);
strmBytes.reset();
}
strmBytes.close();
strmDataType.close();
}
catch (Exception e)
{
db(e.toString());
}
}
private void db(String str)
{
System.err.println("Msg: " + str);
}
public void readRSApub()
{
try
{
byte[] recData = new byte[500];
ByteArrayInputStream strmBytes =
new ByteArrayInputStream(recData);
DataInputStream strmDataType =
new DataInputStream(strmBytes);
RSAKeyPairStore.getRecord(1, recData, 0);
String mod = strmDataType.readUTF();
BigInteger RSAmod = new BigInteger(mod);
strmBytes.reset();
RSAKeyPairStore.getRecord(3, recData, 0);
String pubExp = strmDataType.readUTF();
BigInteger RSApubExp = new BigInteger(pubExp);
strmBytes.reset();
newRSApubKey = new RSAKeyParameters(false, RSAmod, RSApubExp);
strmBytes.close();
strmDataType.close();
}
catch (Exception e)
{
db(e.toString());
}
}
public void ExchangeRSApub()
{
try
{
byte[] recData = new byte[500];
ByteArrayInputStream strmBytes =
new ByteArrayInputStream(recData);
DataInputStream strmDataType =
```

```
new DataInputStream(strmBytes);
RSAKeyPairStore.getRecord(1, recData, 0);
String mod = strmDataType.readUTF();
System.out.println ("mod = " + mod);
BigInteger RSAmod = new BigInteger(mod);
System.out.println ("RSAmod = " + RSAmod);
strmBytes.reset();
RSAKeyPairStore.getRecord(3, recData, 0);
String pubExp = strmDataType.readUTF();
System.out.println ("pubExp = " + pubExp);
BigInteger RSApubExp = new BigInteger(pubExp);
System.out.println ("RSApubExp = " + RSApubExp);
strmBytes.reset();
publickeyField1.setString(mod);
publicExpField1.setString(pubExp);
strmBytes.close();
strmDataType.close();
}
catch (Exception e)
{
db(e.toString());
}
}
public void readRSApriv()
{
try
{
byte[] recData = new byte[500];
ByteArrayInputStream strmBytes =
new ByteArrayInputStream(recData);
DataInputStream strmDataType =
new DataInputStream(strmBytes);
RSAKeyPairStore.getRecord(1, recData, 0);
String mod = strmDataType.readUTF();
BigInteger RSAmod = new BigInteger(mod);
strmBytes.reset();
RSAKeyPairStore.getRecord(2, recData, 0);
String privExp = strmDataType.readUTF();
BigInteger RSAprivExp = new BigInteger(privExp);
strmBytes.reset();
RSAKeyPairStore.getRecord(3, recData, 0);
String pubExp = strmDataType.readUTF();
BigInteger RSApubExp = new BigInteger(pubExp);
strmBytes.reset();
RSAKeyPairStore.getRecord(4, recData, 0);
String dps = strmDataType.readUTF();
BigInteger RSAdp = new BigInteger(dps);
strmBytes.reset();
RSAKeyPairStore.getRecord(5, recData, 0);
String dqs = strmDataType.readUTF();
BigInteger RSAdq = new BigInteger(dqs);
strmBytes.reset();
RSAKeyPairStore.getRecord(6, recData, 0);
String p = strmDataType.readUTF();
BigInteger RSAp = new BigInteger(p);
strmBytes.reset();
RSAKeyPairStore.getRecord(7, recData, 0);
String q = strmDataType.readUTF();
BigInteger RSAq = new BigInteger(q);
strmBytes.reset();
RSAKeyPairStore.getRecord(8, recData, 0);
```

```
String qInv = strmDataType.readUTF();
BigInteger RSAqInv = new BigInteger(qInv);
strmBytes.reset();
newRSAprivKey = new RSAPrivateCrtKeyParameters(
RSAmod, RSAprivExp, RSApubExp, RSAp,RSAq, RSAdp, RSAdq, RSAqInv);
strmBytes.close();
strmDataType.close();
}
catch (Exception e)
{
db(e.toString());
}
}
public void openRecStore()
{
try
{
RSAKeyPairStore = RecordStore.openRecordStore(REC_STORE, true );
}
catch (Exception e)
{
db(e.toString());
}
}
public void closeRecStore()
{
try
{
RSAKeyPairStore.closeRecordStore();
}
catch (Exception e)
{
db(e.toString());
}
}
public void deleteRecStore()
{
if (RecordStore.listRecordStores() != null)
{
try
{
RecordStore.deleteRecordStore(REC_STORE);
}
catch (Exception e)
{
db(e.toString());
}
}
}
public void openRecStore3()
{
try
{
ReceivedSMSStore = RecordStore.openRecordStore(REC_STORE, true );
}
catch (Exception e)
{
db(e.toString());
}
}
public void closeRecStore3()
```

```
{
try
{
ReceivedSMSStore.closeRecordStore();
}
catch (Exception e)
{
db(e.toString());
}
}
public void deleteRecStore3()
{
if (RecordStore.listRecordStores() != null)
{
try
{
RecordStore.deleteRecordStore(REC_STORE);
}
catch (Exception e)
{
db(e.toString());
}
}
}
public void openRecStore2()
{
try
{
PhoneBookStore = RecordStore.openRecordStore(REC_STORE, true );
}
catch (Exception e)
{
db(e.toString());
}
}
public void closeRecStore2()
{
try
{
PhoneBookStore.closeRecordStore();
}
catch (Exception e)
{
db(e.toString());
}
}
public void deleteRecStore2()
{
if (RecordStore.listRecordStores() != null)
{
try
{
RecordStore.deleteRecordStore(REC_STORE);
}
catch (Exception e)
{
db(e.toString());
}
}
}
public void generateTwofishKey () throws Exception {
```

```
long difference = System.currentTimeMillis();
SecureRandom sr = new SecureRandom();
Twofishkey = new byte [16];
sr.nextBytes(Twofishkey);
System.out.println ("TwofishinitV = " + TwofishinitV);
TwofishinitV = new byte [16];
System.out.println ("TwofishinitV1 = " + TwofishinitV);
sr.nextBytes(TwofishinitV);
System.out.println ("Twofishkey1 = " + Twofishkey);
System.out.println ("TwofishinitV2 = " + TwofishinitV);
difference = System.currentTimeMillis() - difference;
f7.append(" Twofish generation of 128 bits key size took" +  difference + "ms");
AESKeyField.setString(new String(Hex.encode(Twofishkey)));
AESinitVField.setString(new String(Base64.encode(TwofishinitV)));
System.out.println ("TwofishkeyHexEncode = " + AESKeyField.getString());
System.out.println ("TwofishinitivBase64Encode = " + AESinitVField.getString());
String TwofishKey1 = AESKeyField.getString();
String TwofishinitV1 = AESinitVField.getString();
String TwofishKeyDetails = TwofishKey1 + "*" + TwofishinitV1;
TwofishCombinedField.setString(TwofishKeyDetails);
}
public byte [] TwofishEncrypt () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes = smsText2.getString().getBytes();
BufferedBlockCipher cipher =
new PaddedBufferedBlockCipher(
new CBCBlockCipher(new TwofishEngine()));
ParametersWithIV piv = new ParametersWithIV (
(new KeyParameter(Twofishkey)), TwofishinitV);
cipher.init(true, piv);
byte[] result =
new byte[cipher.getOutputSize(inBytes.length)];
int len = cipher.processBytes(inBytes, 0,
inBytes.length, result, 0);
try {
cipher.doFinal(result, len);
difference = System.currentTimeMillis() - difference;
} catch (CryptoException ce) {
}
newMessgtwo.setString(new String(Base64.encode(result)));
f7.append(" Twofish Encryption of data of size " + inBytes.length +
"bytes" + " took " +  difference + "ms");
return result;
}
public byte [] TwofishEncrypt2 () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes = nmbh.getBytes();
BufferedBlockCipher cipher =
new PaddedBufferedBlockCipher(
new CBCBlockCipher(new TwofishEngine()));
ParametersWithRandom piv = new ParametersWithRandom (
(new KeyParameter(bnmjhg)));
cipher.init(true, piv);
byte[] result =
new byte[cipher.getOutputSize(inBytes.length)];
int len = cipher.processBytes(inBytes, 0,
inBytes.length, result, 0);
try {
cipher.doFinal(result, len);
difference = System.currentTimeMillis() - difference;
} catch (CryptoException ce) {
```

```
}
newMessgtwo.setString(new String(Base64.encode(result)));
System.out.println ("encrypted public key = " + newMessgtwo.getString());
System.out.println (" Twofish Encryption of data of size " + inBytes.length +
"bytes" + " took " +  difference + "ms");
return result;
}
public byte [] TwofishDecrypt2 () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes1 = Base64.decode(newMessgtwo.getString().getBytes());
BufferedBlockCipher cipher =
new PaddedBufferedBlockCipher(
new CBCBlockCipher(new TwofishEngine()));
ParametersWithRandom piv = new ParametersWithRandom (
(new KeyParameter(bnmjhg)));
cipher.init(false, piv);
byte[] result = new byte[cipher.getOutputSize(inBytes1.length)];
int len = cipher.processBytes(inBytes1, 0,
inBytes1.length, result, 0);
try {
cipher.doFinal(result, len);
difference = System.currentTimeMillis() - difference;
} catch (CryptoException ce) {
}
newMessgthree2.setString(new String(result).trim());
f15.append(" Twofish Decryption took " +  difference + "ms");
String srtf = result.toString();
return result;
}
public byte [] TwofishDecrypt () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes1 = Base64.decode(ReceivedKeyDetails6.getString().getBytes());
BufferedBlockCipher cipher =
new PaddedBufferedBlockCipher(
new CBCBlockCipher(new TwofishEngine()));
ParametersWithIV piv = new ParametersWithIV (
(new KeyParameter(Twofishkey4)), TwofishinitV4);
cipher.init(false, piv);
byte[] result = new byte[cipher.getOutputSize(inBytes1.length)];
int len = cipher.processBytes(inBytes1, 0,
inBytes1.length, result, 0);
try {
cipher.doFinal(result, len);
difference = System.currentTimeMillis() - difference;
} catch (CryptoException ce) {
}
newMessgthree2.setString(new String(result).trim());
f15.append(" Twofish Decryption took " +  difference + "ms");
String srtf = result.toString();
return result;
}
public void generateRSAKeyPair () throws Exception {
long difference = System.currentTimeMillis();
SecureRandom sr = new SecureRandom();
BigInteger pubExp = new BigInteger("10001", 16);
RSAKeyGenerationParameters RSAKeyGenPara =
new RSAKeyGenerationParameters(pubExp, sr, 1024, 80);
RSAKeyPairGenerator RSAKeyPairGen = new RSAKeyPairGenerator();
RSAKeyPairGen.init(RSAKeyGenPara);
AsymmetricCipherKeyPair keyPair = RSAKeyPairGen.generateKeyPair();
RSAprivKey = (RSAPrivateCrtKeyParameters) keyPair.getPrivate();
```

```
RSApubKey = (RSAKeyParameters) keyPair.getPublic();
difference = System.currentTimeMillis() - difference;
display.setCurrent(RUNTIME_MESSAGING);
RUNTIME_MESSAGING.append(" RSA Key pair generation took " +  difference + "ms");
openRecStore();
SerializeRSAKeyPair ();
closeRecStore();
if (RSApubKey == null && RSAprivKey==null){
throw new Exception("Generate RSA keys first!" );
}else {
}
}
public BigInteger calculateMessage()
{
DHKeyPairGenerator dhGen = new DHKeyPairGenerator();
dhGen.init(new DHKeyGenerationParameters(random, dhParams));
AsymmetricCipherKeyPair dhPair = dhGen.generateKeyPair();
this.privateValueUser1 = ((DHPrivateKeyParameters)dhPair.getPrivate()).getX();
return ((DHPublicKeyParameters)dhPair.getPublic()).getY();
}
public BigInteger generateDHKeyPair () throws Exception {
long difference = System.currentTimeMillis();
SecureRandom sr = new SecureRandom();
p = BigInteger.probablePrime(256,sr);
int h = 5;
BigInteger g = new BigInteger(Integer.toString(h));
System.out.println ("SecurerRandom1 " + sr);
System.out.println ("p " + p);
System.out.println ("g " + g);
dhParams = new DHParameters(p, g);
DHKeyGenerationParameters DHKeyGenPara =
new DHKeyGenerationParameters(sr, dhParams);
System.out.println ("dhParams " + dhParams);
DHKeyPairGenerator DHKeyPairGen = new DHKeyPairGenerator();
DHKeyPairGen.init(DHKeyGenPara);
AsymmetricCipherKeyPair dhPair = DHKeyPairGen.generateKeyPair();
this.privateValueUser1 = ((DHPrivateKeyParameters)dhPair.getPrivate()).getX();
System.out.println ("privateValue1 = " + privateValueUser1);
difference = System.currentTimeMillis() - difference;
display.setCurrent(RUNTIME_MESSAGING);
RUNTIME_MESSAGING.append(" DH Key pair generation took " +  difference + "ms");
if (((DHPublicKeyParameters)dhPair.getPublic()).getY() == null || privateValueUser1==null ){
throw new Exception("Generate DH keys first!" );
}else {
}
publicValueUser1 = ((DHPublicKeyParameters)dhPair.getPublic()).getY();
System.out.println ("publicValueUser1 = " + publicValueUser1);
return ((DHPublicKeyParameters)dhPair.getPublic()).getY();
}
public BigInteger generateDHKeyPair2 () throws Exception {
long difference = System.currentTimeMillis();
SecureRandom sr = new SecureRandom();
BigInteger bg = p ;
int h = 5;
BigInteger g = new BigInteger(Integer.toString(h));
System.out.println ("SecurerRandom1 " + sr);
System.out.println ("bg " + bg);
System.out.println ("g " + g);
dhParams = new DHParameters(bg, g);
DHKeyGenerationParameters DHKeyGenPara =
new DHKeyGenerationParameters(sr, dhParams);
```

```
System.out.println ("dhParams " + dhParams);
DHKeyPairGenerator DHKeyPairGen = new DHKeyPairGenerator();
DHKeyPairGen.init(DHKeyGenPara);
AsymmetricCipherKeyPair dhPair = DHKeyPairGen.generateKeyPair();
this.privateValueUser2 = ((DHPrivateKeyParameters)dhPair.getPrivate()).getX();
System.out.println ("privateValue2 = " + privateValueUser2);
difference = System.currentTimeMillis() - difference;
display.setCurrent(RUNTIME_MESSAGING);
RUNTIME_MESSAGING.append(" DH Key pair generation took " +  difference + "ms");
publicValueUser2 = ((DHPublicKeyParameters)dhPair.getPublic()).getY();
System.out.println ("publicValueUser2 " + publicValueUser2);
return ((DHPublicKeyParameters)dhPair.getPublic()).getY();
}
public BigInteger generateDHKeyPaira () throws Exception {
long difference = System.currentTimeMillis();
SecureRandom sr = new SecureRandom();
p = BigInteger.probablePrime(192,sr);
int h = 5;
BigInteger g = new BigInteger(Integer.toString(h));
System.out.println ("SecurerRandom1 " + sr);
System.out.println ("p " + p);
System.out.println ("g " + g);
dhParams = new DHParameters(p, g);
DHKeyGenerationParameters DHKeyGenPara =
new DHKeyGenerationParameters(sr, dhParams);
System.out.println ("dhParams " + dhParams);
DHKeyPairGenerator DHKeyPairGen = new DHKeyPairGenerator();
DHKeyPairGen.init(DHKeyGenPara);
AsymmetricCipherKeyPair dhPair = DHKeyPairGen.generateKeyPair();
this.privateValueUser1 = ((DHPrivateKeyParameters)dhPair.getPrivate()).getX();
System.out.println ("privateValue1 = " + privateValueUser1);
difference = System.currentTimeMillis() - difference;
display.setCurrent(RUNTIME_MESSAGING);
RUNTIME_MESSAGING.append(" DH Key pair generation took " +  difference + "ms");
if (((DHPublicKeyParameters)dhPair.getPublic()).getY() == null || privateValueUser1==null ){
throw new Exception("Generate DH keys first!" );
}else {
}
publicValueUser1 = ((DHPublicKeyParameters)dhPair.getPublic()).getY();
DHValues = "1111" + p + "****" + publicValueUser1 + "####" + h;
System.out.println ("publicValueUser1 = " + publicValueUser1);
return ((DHPublicKeyParameters)dhPair.getPublic()).getY();
}
public BigInteger generateDHKeyPair2a () throws Exception {
long difference = System.currentTimeMillis();
SecureRandom sr = new SecureRandom();
String PKAKeyDetails = ReceivedPKAKeyDetails.getString();
String nextRecdStr = new String(PKAKeyDetails);
String takep = nextRecdStr.substring(4, nextRecdStr.indexOf("****"));
String takePubKey2 = nextRecdStr.substring(nextRecdStr.indexOf("****") +4, nextRecdStr.indexOf("####"));
String takeh =    nextRecdStr.substring(nextRecdStr.indexOf("####") +4, nextRecdStr.length());
BigInteger bg = new BigInteger (takep);
publicValuePKA1 = new BigInteger (takePubKey2);
BigInteger g = new BigInteger (takeh);
System.out.println ("SecurerRandom1 " + sr);
System.out.println ("bg " + bg);
System.out.println ("g " + g);
dhParams = new DHParameters(bg, g);
DHKeyGenerationParameters DHKeyGenPara =
new DHKeyGenerationParameters(sr, dhParams);
System.out.println ("dhParams " + dhParams);
```

```
DHKeyPairGenerator DHKeyPairGen = new DHKeyPairGenerator();
DHKeyPairGen.init(DHKeyGenPara);
AsymmetricCipherKeyPair dhPair = DHKeyPairGen.generateKeyPair();
this.privateValueUser2 = ((DHPrivateKeyParameters)dhPair.getPrivate()).getX();
System.out.println ("privateValue2 = " + privateValueUser2);
difference = System.currentTimeMillis() - difference;
display.setCurrent(RUNTIME_MESSAGING);
RUNTIME_MESSAGING.append(" DH Key pair generation took " +  difference + "ms");
publicValueUser2 = ((DHPublicKeyParameters)dhPair.getPublic()).getY();
System.out.println ("publicValueUser2 " + publicValueUser2);
return ((DHPublicKeyParameters)dhPair.getPublic()).getY();
}
public byte [] RSAEncryptTwofish () throws Exception {
long difference = System.currentTimeMillis();
String asdfghj = TwofishCombinedField.getString();
byte[] inBytes = TwofishCombinedField.getString().getBytes();
if (newRSApubKey == null){
throw new Exception("Generate RSA keys first!");
}
AsymmetricBlockCipher eng = new RSAEngine();
eng = new PKCS1Encoding(eng);
eng.init(true, newRSApubKey);
difference = System.currentTimeMillis() - difference;
f7.append(" RSA Encryption of Twofish (session) key took " + difference + "ms");
String dftg = eng.processBlock(inBytes, 0, inBytes.length).toString();
byte [] vbnm2 = eng.processBlock(inBytes, 0, inBytes.length);
String asdfgh2 =  new String (vbnm2);
newMessgtwob.setString(new String(asdfgh2));
return eng.processBlock(inBytes, 0, inBytes.length);
}
public byte [] RSAEncrypt () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes = smsText2.getString().getBytes();
if (newRSApubKey == null){
throw new Exception("Generate RSA keys first!");
}
AsymmetricBlockCipher eng = new RSAEngine();
eng = new PKCS1Encoding(eng);
eng.init(true, newRSApubKey);
difference = System.currentTimeMillis() - difference;
f7.append(" RSA Encryption of data of size" + inBytes.length
+ "bytes" + " took " +  difference + "ms");
newMessgtwo.setString(new String(Base64.encode(eng.processBlock(inBytes, 0, inBytes.length))));
String fghj = eng.processBlock(inBytes, 0, inBytes.length).toString();
return eng.processBlock(inBytes, 0, inBytes.length);
}
public byte [] RSADecrypt () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes1 = ReceivedKeyDetails7.getString().getBytes();
if (newRSAprivKey == null){
throw new Exception("Generate RSA keys first!");
}
AsymmetricBlockCipher eng = new RSAEngine();
eng = new PKCS1Encoding(eng);
eng.init(false, newRSAprivKey);
difference = System.currentTimeMillis() - difference;
f8.append(" RSA Decryption of data of size" + inBytes1.length + "bytes"
+ " took " +  difference + "ms");
newMessgthree.setString(new String(eng.processBlock(inBytes1, 0, inBytes1.length)).trim());
String TwofishKeynew = newMessgthree.getString();
String takeKey = TwofishKeynew.substring(0, TwofishKeynew.indexOf("*"));
```

```
String takeinitV = TwofishKeynew.substring(TwofishKeynew.indexOf("*")+1, TwofishKeynew.length());
Twofishkey4 = Hex.decode(takeKey);
TwofishinitV4 = Hex.decode(takeinitV);
return eng.processBlock(inBytes1, 0, inBytes1.length);
}
public byte [] RSASign () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes2 = newMessgtwo.getString().getBytes();
if (newRSAprivKey == null)
throw new Exception("Generate RSA keys first!");
SHA256Digest dig = new SHA256Digest();
RSAEngine eng = new RSAEngine();
PSSSigner signer = new PSSSigner(eng, dig, 64);
signer.init(true, newRSAprivKey);
signer.update(inBytes2, 0, inBytes2.length);
difference = System.currentTimeMillis() - difference;
f7.append("Digital Signature of data of size" + inBytes2.length + "bytes" + " took " +  difference + "ms");
String vbn = signer.generateSignature().toString();
byte [] vbnm = signer.generateSignature();
String asdfgh =  new String (vbnm);
newMessgtwoa.setString(new String(asdfgh));
return signer.generateSignature();
}
public byte [] RSASign2 () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes2 = smsText2PKVR.getString().getBytes();
if (newRSAprivKey == null)
throw new Exception("Generate RSA keys first!");
SHA256Digest dig = new SHA256Digest();
RSAEngine eng = new RSAEngine();
PSSSigner signer = new PSSSigner(eng, dig, 64);
signer.init(true, newRSAprivKey);
signer.update(inBytes2, 0, inBytes2.length);
difference = System.currentTimeMillis() - difference;
byte [] vbnm = signer.generateSignature();
String asdfgh =  new String (vbnm);
String PKVR =   smsText2PKVR.getString();
String CombinedPKVR = "3333" + PKVR + "$$$$" + asdfgh;
PKVRfinal.setString(CombinedPKVR);
return signer.generateSignature();
}
public byte [] RSASign4 () throws Exception {
long difference = System.currentTimeMillis();
byte[] inBytes2 = smsText.getString().getBytes();
if (newRSAprivKey == null)
throw new Exception("Generate RSA keys first!");
SHA256Digest dig = new SHA256Digest();
RSAEngine eng = new RSAEngine();
PSSSigner signer = new PSSSigner(eng, dig, 64);
signer.init(true, newRSAprivKey);
signer.update(inBytes2, 0, inBytes2.length);
difference = System.currentTimeMillis() - difference;
byte [] vbnm = signer.generateSignature();
String asdfgh =  new String (vbnm);
String PKVR4 =   smsText.getString();
String CombinedPKVR4 = PKVR4 + "$$$$" + asdfgh;
PKVRfinal.setString(CombinedPKVR4);
return signer.generateSignature();
}
public byte [] RSASign3 () throws Exception {
long difference = System.currentTimeMillis();
```

```
byte[] inBytes2 = smsText2RS.getString().getBytes();
if (newRSAprivKey == null)
throw new Exception("Generate RSA keys first!");
SHA256Digest dig = new SHA256Digest();
RSAEngine eng = new RSAEngine();
PSSSigner signer = new PSSSigner(eng, dig, 64);
signer.init(true, newRSAprivKey);
signer.update(inBytes2, 0, inBytes2.length);
difference = System.currentTimeMillis() - difference;
byte [] vbnm = signer.generateSignature();
String asdfgh =  new String (vbnm);
String PKVResponse =   smsText2RS.getString();
String CombinedPKVResponse = "4444" + PKVResponse + "$$$$" + asdfgh;
PKVResponsefinal.setString(CombinedPKVResponse);
return signer.generateSignature();
}
public boolean RSAVerify () throws Exception {
long difference = System.currentTimeMillis();
byte[] mesg = ReceivedKeyDetails6.getString().getBytes();
byte [] sig = ReceivedKeyDetails4.getString().getBytes();
if (newRSApubKey == null)
throw new Exception("Generate RSA keys first!");
SHA256Digest dig = new SHA256Digest();
RSAEngine eng = new RSAEngine();
PSSSigner signer = new PSSSigner(eng, dig, 64);
signer.init(false, newRSApubKey);
signer.update(mesg, 0, mesg.length);
difference = System.currentTimeMillis() - difference;
mainForm4.append(" Signature verification message of size" + mesg.length +
"bytes" + "and Signature of size" + sig.length + "bytes " + "took" + difference + "ms");
if (signer.verifySignature(sig) == true){
mainForm4.removeCommand(confirmUser);
mainForm4.addCommand(decrypt);
alert = new Alert("Message Verication", "This message HAS NOT BEEN modified. Click"
+ "decrypt" + "in the menu to obatain the original message", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4);
}
else if (signer.verifySignature(sig) == false){
alert = new Alert("Message Verication", "Oops, This message HAS BEEN modified!. Click"
+ "Discard" + "in the menu to delete it", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4);
}
return signer.verifySignature(sig);
}
public boolean RSAVerify1 () throws Exception {
long difference = System.currentTimeMillis();
byte[] mesg = ReceivedKeyDetails6e.getString().getBytes();
byte [] sig = ReceivedKeyDetails4e.getString().getBytes();
if (newRSApubKey == null)
throw new Exception("Generate RSA keys first!");
SHA256Digest dig = new SHA256Digest();
RSAEngine eng = new RSAEngine();
PSSSigner signer = new PSSSigner(eng, dig, 64);
signer.init(false, newRSApubKey);
signer.update(mesg, 0, mesg.length);
difference = System.currentTimeMillis() - difference;
mainForm4e4.append("PKV Response verification message of size" +  mesg.length
+ "bytes " + " and Signature of size " + sig.length + "bytes " + "took " + difference + "ms");
if (signer.verifySignature(sig) == true){
```

```
mainForm4e.removeCommand(confirmUser);
mainForm4e.addCommand(ok4);
alert = new Alert("PKA's Response Authentication",
"This Response HAS NOT been modified and its truly from the PKA. Click"
+ " continue" + " in the menu to see key's status", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4e4);
}
else if (signer.verifySignature(sig) == false){
mainForm4e.removeCommand(confirmUser);
mainForm4e.addCommand(discard);
alert = new Alert("Revocation Veriffication", "This response is either "
+ "modified in transit or not from your PKA (not genuine). Click" + "Discard"
+ "in the menu to delete it", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4e);
}
return signer.verifySignature(sig);
}
public boolean RSAVerify2 () throws Exception {
long difference = System.currentTimeMillis();
byte[] mesg = ReceivedKeyDetails6f.getString().getBytes();
byte [] sig = ReceivedKeyDetails4f.getString().getBytes();
if (newRSApubKey == null)
throw new Exception("Generate RSA keys first!");
SHA256Digest dig = new SHA256Digest();
RSAEngine eng = new RSAEngine();
PSSSigner signer = new PSSSigner(eng, dig, 64);
signer.init(false, newRSApubKey);
signer.update(mesg, 0, mesg.length);
difference = System.currentTimeMillis() - difference;
mainForm4f.append(" PKV Response verification message of size" + mesg.length
+ "bytes" + "and Signature of size" + sig.length + "bytes "
+ "took" + difference + "ms");
if (signer.verifySignature(sig) == true){
mainForm4f.removeCommand(confirmUser);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4e);
}
else if (signer.verifySignature(sig) == false){
mainForm4e.removeCommand(confirmUser);
mainForm4e.addCommand(discard);
alert = new Alert("PKV Response Veriffication", "This response is either "
+ "modified in transit or not from your PKA (not genuine). Click" + "Discard"
+ "in the menu to delete it", null, AlertType.INFO);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert,mainForm4e);
}
return signer.verifySignature(sig);
}
class Reader implements Runnable {
private int waitingMessages = 0;
private prosms_1 sendback;
private String message;
private String number;
private Vector msgstorage;
private Vector numberstorage;
private int ourcount;
private int getcounter = 0;
private int rns = 6, rne = 18;
```

```
public Reader(prosms_1 main) {
ourcount = 0;
msgstorage = new Vector();
numberstorage = new Vector();
this.sendback = main;
}
public void run() {
while (!done) {
synchronized(this) {
if (waitingMessages == 0) {
try {
wait();
} catch (Exception e) {
}
}
waitingMessages--;
}
try {
Message mess = conn.receive();
number = mess.getAddress();
message = ((TextMessage)mess).getPayloadText();
number = number.substring(rns,rne);
msgstorage.addElement(message);
numberstorage.addElement(number);
sendback.setTicker("You have message to read.");
} catch (java.io.IOException ioe) {
}
}
}
public synchronized void handleMessage() {
waitingMessages++;
ourcount++;
notify();
}
public String getNumber() {
number = (String)numberstorage.elementAt(0);
numberstorage.removeElementAt(0);
return number;
}
public String getMessage() {
ourcount = ourcount-1;
message = (String)msgstorage.elementAt(0);
msgstorage.removeElementAt(0);
return message;
}
public int unreadMessages() {
return ourcount;
}
public int messageCount() {
return waitingMessages;
}
}
private void loadContacts() {
searchChoice.deleteAll();
vectorArray = new Vector();
vectorArray.removeAllElements();
try {
RecordEnumeration enuRec = PhoneBookStore.enumerateRecords(this, this, true);
while (enuRec.hasNextElement()) {
try {
int recordId = enuRec.nextRecordId();
```

```
byte[] nextRec = PhoneBookStore.getRecord(recordId);
String nextRecStr = new String(nextRec);
String takeName = nextRecStr.substring(0, nextRecStr.indexOf("*"));
String takeUserName = nextRecStr.substring(nextRecStr.indexOf("*")
+ 1, nextRecStr.indexOf("#"));
String takeNumber =    nextRecStr.substring(nextRecStr.indexOf("#")
+ 1, nextRecStr.indexOf("&"));
String takePublicKey = nextRecStr.substring(nextRecStr.indexOf("&")
+ 1, nextRecStr.indexOf("$"));
String takePublicExp = nextRecStr.substring(nextRecStr.indexOf("$")
+ 1, nextRecStr.indexOf("!"));
String takePublicKeySta = nextRecStr.substring(nextRecStr.indexOf("!")
+ 1, nextRecStr.length());
searchChoice2.append (takeNumber, null);
searchChoice.append(takeName, null);
ContactDetails conDet = new ContactDetails();
conDet.setContactId(recordId);
conDet.setContactName(takeName);
conDet.setContactUserName(takeUserName);
conDet.setContactNumber(takeNumber);
conDet.setContactPublicKey(takePublicKey);
conDet.setContactPublicExp(takePublicExp );
conDet.setContactKeySta(takePublicKeySta);
vectorArray.addElement(conDet);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
}
} catch (RecordStoreNotOpenException ex) {
ex.printStackTrace();
}
}
public int compare(byte[] rec1, byte[] rec2) {
String data1 = new String(rec1);
String data2 = new String(rec2);
int x = data1.compareTo(data2);
if(x == 0){
return RecordComparator.EQUIVALENT;
}else if(x < 0){
return RecordComparator.PRECEDES;
}else{
return RecordComparator.FOLLOWS;
}
}
public void itemStateChanged(Item item) {
if(item == searchTxtField){
filterStr = searchTxtField.getString();
loadContacts();
}else if (item == searchTxtField2){
filterStr2 = searchTxtField2.getString();
loadContacts();
}
}
public boolean matches(byte[] candidate) {
String candi = new String(candidate);
if (candi.startsWith(filterStr)||candi.startsWith(filterStr2) ) {
return true;
}
return false;
}
public void commandAction(Command c, Item item) {
```

```
if(c == viewCmd){
int getSelInd = searchChoice.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
changeNameTxtField.setString(getConInfo.getContactName());
changeUserNameTxtField.setString(getConInfo.getContactUserName());
changeNumberTxtField.setString(getConInfo.getContactNumber());
changepublickeyField.setString(getConInfo.getContactPublicKey());
changepublickeyField.setString(getConInfo.getContactPublicExp());
changepublicKeySta.setString(getConInfo.getContactKeySta());
display.setCurrent(changeContactForm);
}
if (c == backCmd2) {
System.out.println ("Alhamdulilah!");
try {
generateDHKeyPair ();
generateDHKeyPair2 ();
calculateAgreement ();
calculateAgreement2 ();
} catch (Exception ex) {
ex.printStackTrace();
}
}
if(c == compose){
int getSelInd = searchChoice.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
ContactName2.setString(getConInfo.getContactName());
ContactUserName2.setString(getConInfo.getContactUserName());
PhoneNo2.setString(getConInfo.getContactNumber());
publicExpField21.setString(getConInfo.getContactKeySta());
publickeyField2.setString(getConInfo.getContactPublicKey());
publicExpField2.setString(getConInfo.getContactPublicExp());
String UserMod = publickeyField2.getString();
BigInteger RSAmod1 = new BigInteger(UserMod);
String UserPubExp = publicExpField2.getString();
BigInteger RSAPubExp1 = new BigInteger(UserPubExp);
newRSApubKey = new RSAKeyParameters(false, RSAmod1, RSAPubExp1);
display.setCurrent(mainForm2);
}
if(c == compose2){
int getSelInd = searchChoice.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
publicExpField21.setString(getConInfo.getContactKeySta());
if (publicExpField21.getString().equals("Unverified")){
keyStatusForm.deleteAll();
keyStatusForm.append("This user Identity Details has not been verified! "
+ "Please contact Your PKA about this User");
keyStatusForm.append (" Do you want to send a Public Validation Request Now?");
keyStatusForm.addCommand(yes2);
keyStatusForm.addCommand(back2);
display.setCurrent(keyStatusForm);
}
else if (publicExpField21.getString().equals("Revoked")){
keyStatusForm.deleteAll();
keyStatusForm.append("This user's public key has been revoked and thus SMS "
+ "commuinication with the user is not SECURED! You may need to inform the user about this!");
keyStatusForm.removeCommand(yes2);
keyStatusForm.removeCommand(back2);
keyStatusForm.addCommand(ok);
display.setCurrent(keyStatusForm);
}
else {
```

```
ContactName2.setString(getConInfo.getContactName());
ContactUserName2.setString(getConInfo.getContactUserName());
PhoneNo2.setString(getConInfo.getContactNumber());
publicExpField21.setString(getConInfo.getContactKeySta());
publickeyField2.setString(getConInfo.getContactPublicKey());
publicExpField2.setString(getConInfo.getContactPublicExp());
String UserMod = publickeyField2.getString();
BigInteger RSAmod1 = new BigInteger(UserMod);
String UserPubExp = publicExpField2.getString();
BigInteger RSAPubExp1 = new BigInteger(UserPubExp);
newRSApubKey = new RSAKeyParameters(false, RSAmod1, RSAPubExp1);
display.setCurrent(mainForm2);
}
}
if(c == bindidentity){
int getSelInd = searchChoice.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
ContactName2PKVR.setString(getConInfo.getContactName());
ContactUserName2PKVR.setString(getConInfo.getContactUserName());
PhoneNo2PKVR.setString(getConInfo.getContactNumber());
String PVRname = ContactName2PKVR.getString();
String PKVRusername =    ContactUserName2PKVR.getString();
String PKVRuserccontact =   PhoneNo2PKVR.getString();
String PKVRequestSMS = PVRname + "****" + PKVRusername + "####" + PKVRuserccontact;
smsText2PKVR.setString(PKVRequestSMS);
display.setCurrent(sendPKVRForm);
}
if(c == notRevoked){
int getSelInd = searchChoice.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
ContactName2RS.setString(getConInfo.getContactName());
ContactUserName2RS.setString(getConInfo.getContactUserName());
PhoneNo2RS.setString(getConInfo.getContactNumber());
publickeyField2PKVR.setString(getConInfo.getContactPublicKey());
publicExpField2PKVR.setString(getConInfo.getContactPublicExp());
String PVRname1 = ContactName2RS.getString();
String PKVRusername1 =    ContactName2RS.getString();
String PKVRuserccontact1 =   PhoneNo2RS.getString();
String PKVRpubMod =    publickeyField2PKVR.getString();
String PKVRpubExp =   publicExpField2PKVR.getString();
String PKVResponseSMS = PVRname1 + "****" + PKVRusername1 + "####"
+ PKVRuserccontact1 + "$$$$" + PKVRpubMod + "&&&&" + PKVRpubExp;
smsText2RS.setString(PKVResponseSMS);
display.setCurrent(RevocationStatusForm);
}
if(c == verify1){
try {
PhoneBookStore = RecordStore.openRecordStore("Contacts", true);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
display.setCurrent(searchForm2);
int getSelInd = searchChoice2.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
publickeyField2.setString(getConInfo.getContactPublicKey());
publicExpField2.setString(getConInfo.getContactPublicExp());
String UserMod = publickeyField2.getString();
BigInteger RSAmod1 = new BigInteger(UserMod);
String UserPubExp = publicExpField2.getString();
BigInteger RSAPubExp1 = new BigInteger(UserPubExp);
```

```
newRSApubKey = new RSAKeyParameters(false, RSAmod1, RSAPubExp1);
try {
RSAVerify ();
} catch (Exception ex) {
ex.printStackTrace();
}
}
if(c == verify2){
try {
PhoneBookStore = RecordStore.openRecordStore("Contacts", true);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
display.setCurrent(searchForm2);
int getSelInd = searchChoice2.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
publickeyField2.setString(getConInfo.getContactPublicKey());
publicExpField2.setString(getConInfo.getContactPublicExp());
String UserMod = publickeyField2.getString();
BigInteger RSAmod1 = new BigInteger(UserMod);
String UserPubExp = publicExpField2.getString();
BigInteger RSAPubExp1 = new BigInteger(UserPubExp);
newRSApubKey = new RSAKeyParameters(false, RSAmod1, RSAPubExp1);
try {
RSAVerify1();
} catch (Exception ex) {
ex.printStackTrace();
}
}
if(c == verify2){
try {
PhoneBookStore = RecordStore.openRecordStore("Contacts", true);
} catch (RecordStoreException ex) {
ex.printStackTrace();
}
loadContacts();
display.setCurrent(searchForm2);
int getSelInd = searchChoice2.getSelectedIndex();
ContactDetails getConInfo = (ContactDetails) vectorArray.elementAt(getSelInd);
publickeyField2.setString(getConInfo.getContactPublicKey());
publicExpField2.setString(getConInfo.getContactPublicExp());
String UserMod = publickeyField2.getString();
BigInteger RSAmod1 = new BigInteger(UserMod);
String UserPubExp = publicExpField2.getString();
BigInteger RSAPubExp1 = new BigInteger(UserPubExp);
newRSApubKey = new RSAKeyParameters(false, RSAmod1, RSAPubExp1);
try {
RSAVerify2();
} catch (Exception ex) {
ex.printStackTrace();
}
}
}
class ContactDetails {
private int contactId;
private String contactName;
private String contactUserName;
private String contactNumber;
private String contactPublicKey;
private String contactPubExp;
```

```
private String contactKeyStatus;
public int getContactId() {
return contactId;
}
public void setContactId(int contactId) {
this.contactId = contactId;
}
public String getContactName() {
return contactName;
}
public void setContactName(String contactName) {
this.contactName = contactName;
}
public String getContactUserName() {
return contactUserName;
}
public void setContactUserName(String contactUserName) {
this.contactUserName = contactUserName;
}
public String getContactNumber() {
return contactNumber;
}
public void setContactNumber(String contactNumber) {
this.contactNumber = contactNumber;
}
public String getContactPublicKey() {
return contactPublicKey;
}
public void setContactPublicKey(String contactPublicKey) {
this.contactPublicKey = contactPublicKey;
}
public String getContactPublicExp() {
return contactPubExp;
}
public void setContactPublicExp(String contactPubExp) {
this.contactPubExp = contactPubExp;
}
public String getContactKeySta() {
return contactKeyStatus;
}
public void setContactKeySta(String contactKeyStatus) {
this.contactKeyStatus = contactKeyStatus;
}
}
private void lookupBalance_withGET() throws IOException
{
long difference = System.currentTimeMillis();
HttpConnection http = null;
InputStream iStrm = null;
boolean ret = false;
String url = "http://ocsp.comodoca.com/" + OCSPRequestString.getString();
try
{
http = (HttpConnection) Connector.open(url);
http.setRequestMethod(HttpConnection.GET);
iStrm = http.openInputStream();
ret = processServerResponse(http, iStrm);
}
finally
{
if (iStrm != null)
```

```
iStrm.close();
if (http != null)
http.close();
}
if (ret == false)
showAlert(errorMsg);
difference = System.currentTimeMillis() - difference;
OCSP_MEASURE.append(" OCSP Test took " +  difference + "ms");
}
// this throws up error because the OCSP data used is less than 255 bytes
private void lookupBalance_withPOST() throws IOException
{
HttpConnection http = null;
OutputStream oStrm = null;
InputStream iStrm = null;
boolean ret = false;
String url = "http://ocsp.comodoca.com/";
try
{
http = (HttpConnection) Connector.open(url);
oStrm = http.openOutputStream();
http.setRequestMethod(HttpConnection.POST);
http.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
byte data[] = (OCSPRequestString.getString()).getBytes();
oStrm.write(data);
iStrm = http.openInputStream();
ret = processServerResponse(http, iStrm);
}
finally
{
if (iStrm != null)
iStrm.close();
if (oStrm != null)
oStrm.close();
if (http != null)
http.close();
}
if (ret == false)
showAlert(errorMsg);
}
private boolean processServerResponse(HttpConnection http, InputStream iStrm) throws IOException
{
errorMsg = null;
if (http.getResponseCode() == HttpConnection.HTTP_OK)
{
int length = (int) http.getLength();
String str;
if (length != -1)
{
byte servletData[] = new byte[length];
iStrm.read(servletData);
str = new String(servletData);
}
else
{
ByteArrayOutputStream bStrm = new ByteArrayOutputStream();
int ch;
while ((ch = iStrm.read()) != -1)
bStrm.write(ch);
str = new String(bStrm.toByteArray());
bStrm.close();
```

```
}
OCSP_MEASURE.deleteAll();
OCSP_MEASURE.append(siBalance);
display.setCurrent(OCSP_MEASURE);
siBalance.setText(str);
return true;
}
else
errorMsg = new String( http.getResponseMessage());
return false;
}
private void showAlert(String msg)
{
alError = new Alert("Error", msg, null, AlertType.ERROR);
alError.setTimeout(Alert.FOREVER);
display.setCurrent(alError, menuList);
}
public BigInteger calculateAgreementa()
{
BigInteger pGF = dhParams.getP();
System.out.println ("NEW p = " + pGF);
gfhfh1 = publicValuePKA1.modPow(privateValueUser2, pGF);
bnmjhg = gfhfh1.toByteArray();
System.out.println (" Symmetric Key A = " + gfhfh1);
return publicValuePKA1.modPow(privateValueUser2, pGF);
}
public BigInteger calculateAgreement2()
{
BigInteger pGF = dhParams.getP();
System.out.println ("NEW p = " + pGF);
gfhfhTwo = publicValueUser2.modPow(privateValueUser1, pGF);
System.out.println (" Symmetric Key B = " + gfhfhTwo);
return publicValueUser2.modPow(privateValueUser1, pGF);
}
public BigInteger calculateAgreement()
{
BigInteger pGF = dhParams.getP();
System.out.println ("NEW p = " + pGF);
BigInteger gfhfh = publicValueUser1.modPow(privateValueUser2, pGF);
System.out.println (" Symmetric Key A = " + gfhfh);
return publicValueUser1.modPow(privateValueUser2, pGF);
}
public BigInteger calculateAgreement2b()
{
String PKAKeyDetails = ReceivedPKAKeyDetails.getString();
String nextRecdStr = new String(PKAKeyDetails);
String takepubkey3 = nextRecdStr.substring(4, nextRecdStr.length());
publicValuePKA2 = new BigInteger (takepubkey3);
BigInteger pGF = dhParams.getP();
System.out.println ("NEW p = " + pGF);
gfhfhTwo = publicValuePKA2.modPow(privateValueUser1, pGF);
bnmjhg = gfhfhTwo.toByteArray();
System.out.println (" Symmetric Key B = " + gfhfhTwo);
return publicValuePKA2.modPow(privateValueUser1, pGF);
}
}
```